Richard M. Reese, Jennifer L. Reese,
Alexey Grigorev

Java: Data Science Made Easy

Learning Path

Data Collection, Processing, Analysis and more





Packt>

Java: Data Science Made Easy

Data collection, processing, analysis, and more

A course in two modules

Packt>

BIRMINGHAM - MUMBAI

Java: Data Science Made Easy

Copyright © 2017 Packt Publishing

All rights reserved. No part of this course may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this course is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this course.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Published on: July 2017

Production reference: 1040717

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.
ISBN 978-1-78847-565-5

www.packtpub.com

Credits

Authors
Richard M. Reese
Jennifer L. Reese
Alexey Grigorev

Content Development Editor
Mayur Pawanikar

Reviewers
Walter Molina
Shilpi Saxena
Stanislav Bashkyrtsev
Luca Massaron
Prashant Verma

Production Coordinator
Arvindkumar Gupta

Module 1: Java for Data Science

Chapter 1: Getting Started with Data Science 7

Problems solved using data science 8
Understanding the data science problem - solving approach 9
Using Java to support data science 10
Acquiring data for an application 11
The importance and process of cleaning data 12
Visualizing data to enhance understanding 14
The use of statistical methods in data science 15
Machine learning applied to data science 17
Using neural networks in data science 19
Deep learning approaches 22
Performing text analysis 23
Visual and audio analysis 25
Improving application performance using parallel techniques 27
Assembling the pieces 29
Summary 29

Chapter 2: Data Acquisition 30

Understanding the data formats used in data science applications 31
Overview of CSV data 32
Overview of spreadsheets 32
Overview of databases 33
Overview of PDF files 35
Overview of JSON 36
Overview of XML 36
Overview of streaming data 37
Overview of audio/video/images in Java 38
Data acquisition techniques 39
Using the HttpUrlConnection class 39
Web crawlers in Java 40
Creating your own web crawler 42
Using the crawler4j web crawler 45
Web scraping in Java 48

Using API calls to access common social media sites 52
Using OAuth to authenticate users 52
Handing Twitter 52
Handling Wikipedia 55
Handling Flickr 58
Handling YouTube 61
Searching by keyword 62
Summary 65
Chapter 3: Data Cleaning 66
Handling data formats 67
Handling CSV data 68
Handling spreadsheets 70
Handling Excel spreadsheets 71
Handling PDF files 72
Handling JSON 74
Using JSON streaming API 74
Using the JSON tree API 79
The nitty gritty of cleaning text 80
Using Java tokenizers to extract words 82
Java core tokenizers 83
Third-party tokenizers and libraries 83
Transforming data into a usable form 85
Simple text cleaning 85
Removing stop words 87
Finding words in text 89
Finding and replacing text 90
Data imputation 92
Subsetting data 95
Sorting text 96
Data validation 100
Validating data types 101
Validating dates 102
Validating e-mail addresses 104
Validating ZIP codes 106
Validating names 106
Cleaning images 107
Changing the contrast of an image 108
Smoothing an image 109
Brightening an image 111
Resizing an image 112
Converting images to different formats 113
Summary 114

Chapter 4: Data Visualization 115
Understanding plots and graphs 116
Visual analysis goals 122
Creating index charts 123
Creating bar charts 126
Using country as the category 128
Using decade as the category 130
Creating stacked graphs 133
Creating pie charts 135
Creating scatter charts 138
Creating histograms 140
Creating donut charts 143
Creating bubble charts 145
Summary 148
Chapter 5: Statistical Data Analysis Techniques 149
Working with mean, mode, and median 150
Calculating the mean 150
Using simple Java techniques to find mean 150
Using Java 8 techniques to find mean 151
Using Google Guava to find mean 152
Using Apache Commons to find mean 152
Calculating the median 153
Using simple Java techniques to find median 153
Using Apache Commons to find the median 155
Calculating the mode 155
Using ArrayLists to find multiple modes 157
Using a HashMap to find multiple modes 158
Using a Apache Commons to find multiple modes 159
Standard deviation 159
Sample size determination 162
Hypothesis testing 162
Regression analysis 163
Using simple linear regression 165
Using multiple regression 168
Summary 175
Chapter 6: Machine Learning 176
Supervised learning techniques 177
Decision trees 178
Decision tree types 179
Decision tree libraries 179

Using a decision tree with a book dataset 180
Testing the book decision tree 184
Support vector machines 185
Using an SVM for camping data 188
Testing individual instances 191
Bayesian networks 192
Using a Bayesian network 193
Unsupervised machine learning 196
Association rule learning 196
Using association rule learning to find buying relationships 198
Reinforcement learning 200
Summary 201
Chapter 7: Neural Networks 203
Training a neural network 205
Getting started with neural network architectures 206
Understanding static neural networks 207
A basic Java example 207
Understanding dynamic neural networks 215
Multilayer perceptron networks 215
Building the model 216
Evaluating the model 218
Predicting other values 219
Saving and retrieving the model 220
Learning vector quantization 220
Self-Organizing Maps 221
Using a SOM 221
Displaying the SOM results 222
Additional network architectures and algorithms 226
The k-Nearest Neighbors algorithm 226
Instantaneously trained networks 226
Spiking neural networks 227
Cascading neural networks 227
Holographic associative memory 227
Backpropagation and neural networks 228
Summary 228
Chapter 8: Deep Learning 229
Deeplearning4j architecture 230
Acquiring and manipulating data 231
Reading in a CSV file 231
Configuring and building a model 232
Using hyperparameters in ND4J 233

Instantiating the network model 235
Training a model 235
Testing a model 236
Deep learning and regression analysis 237
Preparing the data 237
Setting up the class 238
Reading and preparing the data 238
Building the model 239
Evaluating the model 240
Restricted Boltzmann Machines 242
Reconstruction in an RBM 243
Configuring an RBM 244
Deep autoencoders 245
Building an autoencoder in DL4J 246
Configuring the network 246
Building and training the network 248
Saving and retrieving a network 248
Specialized autoencoders 248
Convolutional networks 249
Building the model 249
Evaluating the model 252
Recurrent Neural Networks 253
Summary 254
Chapter 9: Text Analysis 255
Implementing named entity recognition 256
Using OpenNLP to perform NER 257
Identifying location entities 258
Classifying text 260
Word2Vec and Doc2Vec 260
Classifying text by labels 260
Classifying text by similarity 263
Understanding tagging and POS 266
Using OpenNLP to identify POS 266
Understanding POS tags 268
Extracting relationships from sentences 269
Using OpenNLP to extract relationships 270
Sentiment analysis 272
Downloading and extracting the Word2Vec model 273
Building our model and classifying text 276
Summary 278

Chapter 10: Visual and Audio Analysis 280
Text-to-speech 281
Using FreeTTS 283
Getting information about voices 285
Gathering voice information 287
Understanding speech recognition 288
Using CMUPhinx to convert speech to text 289
Obtaining more detail about the words 290
Extracting text from an image 292
Using Tess4j to extract text 292
Identifying faces 293
Using OpenCV to detect faces 294
Classifying visual data 296
Creating a Neuroph Studio project for classifying visual images 297
Training the model 304
Summary 309
Chapter 11: Mathematical and Parallel Techniques for Data Analysis 310
Implementing basic matrix operations 311
Using GPUs with DeepLearning4j 313
Using map-reduce 315
Using Apache's Hadoop to perform map-reduce 315
Writing the map method 316
Writing the reduce method 317
Creating and executing a new Hadoop job 318
Various mathematical libraries 320
Using the jblas API 320
Using the Apache Commons math API 321
Using the ND4J API 322
Using OpenCL 324
Using Aparapi 324
Creating an Aparapi application 325
Using Aparapi for matrix multiplication 328
Using Java 8 streams 330
Understanding Java 8 lambda expressions and streams 331
Using Java 8 to perform matrix multiplication 332
Using Java 8 to perform map-reduce 333
Summary 335
Chapter 12: Bringing It All Together 337

Defining the purpose and scope of our application 338
Understanding the application's architecture 338
Data acquisition using Twitter 342
Understanding the TweetHandler class 344
Extracting data for a sentiment analysis model 346
Building the sentiment model 347
Processing the JSON input 348
Cleaning data to improve our results 349
Removing stop words 350
Performing sentiment analysis 351
Analysing the results 351
Other optional enhancements 352
Summary 353

Module 2: Mastering Java for Data Science

Chapter 1: Data Science Using Java 355

Data science 356
Machine learning 357
Supervised learning 358
Unsupervised learning 359
Clustering 359
Dimensionality reduction 359
Natural Language Processing 360
Data science process models 361
CRISP-DM 361
A running example 363
Data science in Java 364
Data science libraries 365
Data processing libraries 365
Math and stats libraries 367
Machine learning and data mining libraries 367
Text processing 368
Summary 369

Chapter 2: Data Processing Toolbox 370

Standard Java library 370
Collections 371
Input/Output 372
Reading input data 372
Writing ouput data 374
Streaming API 375
Extensions to the standard library 378

Apache Commons 378
Commons Lang 378
Commons IO 379
Commons Collections 380
Other commons modules 380
Google Guava 381
AOL Cyclops React 385
Accessing data 386
Text data and CSV 386
Web and HTML 388
JSON 391
Databases 393
DataFrames 396
Search engine - preparing data 398
Summary 402
Chapter 3: Exploratory Data Analysis 403
Exploratory data analysis in Java 403
Search engine datasets 404
Apache Commons Math 405
Joinery 408
Interactive Exploratory Data Analysis in Java 410
JVM languages 410
Interactive Java 411
Joinery shell 412
Summary 417
Chapter 4: Supervised Learning - Classification and Regression 418
Classification 419
Binary classification models 419
Smile 420
JSAT 422
LIBSVM and LIBLINEAR 424
Encog 431
Evaluation 432
Accuracy 433
Precision, recall, and F1 434
ROC and AU ROC (AUC) 436
Result validation 439
K-fold cross-validation 442
Training, validation, and testing 444
Case study - page prediction 446
Regression 450

Machine learning libraries for regression 451
Smile 451
JSAT 452
Other libraries 453
Evaluation 453
MSE 453
MAE 454
Case study - hardware performance 454
Summary 460
Chapter 5: Unsupervised Learning - Clustering and Dimensionality Reduction 461
Dimensionality reduction 462
Unsupervised dimensionality reduction 462
Principal Component Analysis 463
Truncated SVD 467
Truncated SVD for categorical and sparse data 470
Random projection 476
Cluster analysis 479
Hierarchical methods 480
K-means 487
Choosing K in K-Means 488
DBSCAN 490
Clustering for supervised learning 491
Clusters as features 491
Clustering as dimensionality reduction 492
Supervised learning via clustering 495
Evaluation 496
Manual evaluation 496
Supervised evaluation 497
Unsupervised Evaluation 499
Summary 500
Chapter 6: Working with Text - Natural Language Processing and Information Retrieval 501
Natural Language Processing and information retrieval 502
Vector Space Model - Bag of Words and TF-IDF 502
Vector space model implementation 505
Indexing and Apache Lucene 511
Natural Language Processing tools 515
Stanford CoreNLP 515
Customizing Apache Lucene 519
Machine learning for texts 521

Unsupervised learning for texts 521
Latent Semantic Analysis 521
Text clustering 525
Word embeddings 527
Supervised learning for texts 536
Text classification 537
Learning to rank for information retrieval 540
Reranking with Lucene 545
Summary 548
Chapter 7: Extreme Gradient Boosting 549
Gradient Boosting Machines and XGBoost 549
Installing XGBoost 550
XGBoost in practice 553
XGBoost for classification 553
Parameter tuning 560
Text features 562
Feature importance 563
XGBoost for regression 564
XGBoost for learning to rank 568
Summary 573
Chapter 8: Deep Learning with DeepLearning4J 574
Neural Networks and DeepLearning4J 574
ND4J - N-dimensional arrays for Java 575
Neural networks in DeepLearning4J 578
Convolutional Neural Networks 588
Deep learning for cats versus dogs 594
Reading the data 595
Creating the model 598
Monitoring the performance 602
Data augmentation 609
Running DeepLearning4J on GPU 612
Summary 617
Chapter 9: Scaling Data Science 618
Apache Hadoop 619
Hadoop MapReduce 619
Common Crawl 620
Apache Spark 630
Link prediction 633
Reading the DBLP graph 634

Extracting features from the graph 636
Node features 638
Negative sampling 642
Edge features 647
Link Prediction with MLlib and XGBoost 652
Link suggestion 657
Summary 662
Chapter 10: Deploying Data Science Models 663
Microservices 664
Spring Boot 664
Search engine service 665
Online evaluation 675
A/B testing 675
Multi-armed bandits 680
Summary 683
Bibliography 684
Index 685

Preface

Data science is concerned with extracting knowledge and insights from a wide variety of data sources to analyze patterns or predict future behavior. It draws from a wide array of disciplines including statistics, computer science, mathematics, machine learning, and data mining. In this course, we cover the basic as well as advanced data science concepts and how they are implemented using the popular Java tools and libraries.

The course starts with an introduction of data science, followed by the basic data science tasks of data collection, data cleaning, data analysis, and data visualization. This is followed by a discussion of statistical techniques and more advanced topics including machine learning, neural networks, and deep learning. You will examine the major categories of data analysis including text, visual, and audio data, followed by a discussion of resources that support parallel implementation. Throughout this course, the chapters will illustrate a challenging data science problem, and then go on to present a comprehensive, Java-based solution to tackle that problem. You will cover a wide range of topics – from classification and regression, to dimensionality reduction and clustering, deep learning and working with Big Data. Finally, you will see the different ways to deploy the model and evaluate it in production settings.

By the end of this course, you will be up and running with various facets of data science using Java, in no time at all.

What this learning path covers

Module 1, Java for data science, this module takes an expansive yet cursory approach to various aspects of data science. A brief introduction to the field is presented in the first chapter. Subsequent chapters cover significant aspects of data science, such as data cleaning and the application of neural networks. The last chapter combines topics discussed throughout the book to create a comprehensive data science application.

Module 2, *Mastering Java for data science*, in this module we will see how we can utilize Java's toolbox for processing small and large datasets, then look into doing initial exploration data analysis.

Next, we will review the Java libraries that implement common Machine Learning models for classification, regression, clustering, and dimensionality reduction problems. Then we will get into more advanced techniques and discuss Information Retrieval and Natural Language Processing, XGBoost, deep learning, and large scale tools for processing big datasets such as Apache Hadoop and Apache Spark. Finally, we will also have a look at how to evaluate and deploy the produced models such that the other services can use them.

What you need for this learning path

Many of the examples in the book use Java 8 features. There are a number of Java APIs demonstrated, each of which is introduced before it is applied. An IDE is not required but is desirable.

You need to have any latest system with at least 2GB RAM and a Windows 7 /Ubuntu 14.04/Mac OS X operating system. Further, you will need to have Java 1.8.0 or above and Maven 3.0.0 or above installed.

Who this learning path is for

This course is meant for Java developers who are comfortable developing applications in Java, and now want to enter the world of data science or wish to build intelligent applications. Aspiring data scientists with some understanding of the Java programming language will also find this book to be very helpful. If you are willing to build efficient data science applications and bring them in the enterprise environment without changing your existing Java stack, this book is for you!

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Here, we create `SummaryStatistics` objects and add all body content lengths."

A block of code is set as follows:

```
SummaryStatistics statistics = new SummaryStatistics();
data.stream().mapToDouble(RankedPage::getBodyContentLength)
    .forEach(statistics::addValue);
System.out.println(statistics.getSummary());
```

Any command-line input or output is written as follows:

```
mvn dependency:copy-dependencies -DoutputDirectory=lib
mvn compile
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "If, instead, our model outputs some score such that the higher the values of the score the more likely the item is to be positive, then the binary classifier is called a **ranking classifier**."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Mastering-Java-for-Data-Science`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/MasteringJavaforDataScience_ColorImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

Module 1

Java for Data Science

Examine the techniques and Java tools supporting the growing field of data science

1
Getting Started with Data Science

Data science is not a single science as much as it is a collection of various scientific disciplines integrated for the purpose of analyzing data. These disciplines include various statistical and mathematical techniques, including:

- Computer science
- Data engineering
- Visualization
- Domain-specific knowledge and approaches

With the advent of cheaper storage technology, more and more data has been collected and stored permitting previously unfeasible processing and analysis of data. With this analysis came the need for various techniques to make sense of the data. These large sets of data, when used to analyze data and identify trends and patterns, become known as **big data**.

This in turn gave rise to cloud computing and concurrent techniques such as **map-reduce,** which distributed the analysis process across a large number of processors, taking advantage of the power of parallel processing.

The process of analyzing big data is not simple and evolves to the specialization of developers who were known as **data scientists**. Drawing upon a myriad of technologies and expertise, they are able to analyze data to solve problems that previously were either not envisioned or were too difficult to solve.

Early big data applications were typified by the emergence of search engines capable of more powerful and accurate searches than their predecessors. For example, **AltaVista** was an early popular search engine that was eventually superseded by Google. While big data applications were not limited to these search engine functionalities, these applications laid the groundwork for future work in big data.

The term, data science, has been used since 1974 and evolved over time to include statistical analysis of data. The concepts of data mining and data analytics have been associated with data science. Around 2008, the term data scientist appeared and was used to describe a person who performs data analysis. A more in-depth discussion of the history of data science can be found at `http://www.forbes.com/sites/gilpress/2013/05/28/a-very-short-history-of-data-science/#3d9ea08369fd`.

This book aims to take a broad look at data science using Java and will briefly touch on many topics. It is likely that the reader may find topics of interest and pursue these at greater depth independently. The purpose of this book, however, is simply to introduce the reader to the significant data science topics and to illustrate how they can be addressed using Java.

There are many algorithms used in data science. In this book, we do not attempt to explain how they work except at an introductory level. Rather, we are more interested in explaining how they can be used to solve problems. Specifically, we are interested in knowing how they can be used with Java.

Problems solved using data science

The various data science techniques that we will illustrate have been used to solve a variety of problems. Many of these techniques are motivated to achieve some economic gain, but they have also been used to solve many pressing social and environmental problems. Problem domains where these techniques have been used include finance, optimizing business processes, understanding customer needs, performing DNA analysis, foiling terrorist plots, and finding relationships between transactions to detect fraud, among many other data-intensive problems.

Data mining is a popular application area for data science. In this activity, large quantities of data are processed and analyzed to glean information about the dataset, to provide meaningful insights, and to develop meaningful conclusions and predictions. It has been used to analyze customer behavior, detecting relationships between what may appear to be unrelated events, and to make predictions about future behavior.

Machine learning is an important aspect of data science. This technique allows the computer to solve various problems without needing to be explicitly programmed. It has been used in self-driving cars, speech recognition, and in web searches. In data mining, the data is extracted and processed. With machine learning, computers use the data to take some sort of action.

Understanding the data science problem - solving approach

Data science is concerned with the processing and analysis of large quantities of data to create models that can be used to make predictions or otherwise support a specific goal. This process often involves the building and training of models. The specific approach to solve a problem is dependent on the nature of the problem. However, in general, the following are the high-level tasks that are used in the analysis process:

- **Acquiring the data**: Before we can process the data, it must be acquired. The data is frequently stored in a variety of formats and will come from a wide range of data sources.
- **Cleaning the data**: Once the data has been acquired, it often needs to be converted to a different format before it can be used. In addition, the data needs to be processed, or cleaned, so as to remove errors, resolve inconsistencies, and otherwise put it in a form ready for analysis.
- **Analyzing the data**: This can be performed using a number of techniques including:
    - **Statistical analysis**: This uses a multitude of statistical approaches to provide insight into data. It includes simple techniques and more advanced techniques such as regression analysis.
    - **AI analysis**: These can be grouped as machine learning, neural networks, and deep learning techniques:
        - Machine learning approaches are characterized by programs that can learn without being specifically programmed to complete a specific task
        - Neural networks are built around models patterned after the neural connection of the brain
        - Deep learning attempts to identify higher levels of abstraction within a set of data

- **Text analysis**: This is a common form of analysis, which works with natural languages to identify features such as the names of people and places, the relationship between parts of text, and the implied meaning of text.
- **Data visualization**: This is an important analysis tool. By displaying the data in a visual form, a hard-to-understand set of numbers can be more readily understood.
- **Video, image, and audio processing and analysis**: This is a more specialized form of analysis, which is becoming more common as better analysis techniques are discovered and faster processors become available. This is in contrast to the more common text processing and analysis tasks.

Complementing this set of tasks is the need to develop applications that are efficient. The introduction of machines with multiple processors and GPUs contributes significantly to the end result.

While the exact steps used will vary by application, understanding these basic steps provides the basis for constructing solutions to many data science problems.

Using Java to support data science

Java and its associated third-party libraries provide a range of support for the development of data science applications. There are numerous core Java capabilities that can be used, such as the basic string processing methods. The introduction of lambda expressions in Java 8 helps enable more powerful and expressive means of building applications. In many of the examples that follow in subsequent chapters, we will show alternative techniques using lambda expressions.

There is ample support provided for the basic data science tasks. These include multiple ways of acquiring data, libraries for cleaning data, and a wide variety of analysis approaches for tasks such as natural language processing and statistical analysis. There are also myriad of libraries supporting neural network types of analysis.

Java can be a very good choice for data science problems. The language provides both object-oriented and functional support for solving problems. There is a large developer community to draw upon and there exist multiple APIs that support data science tasks. These are but a few reasons as to why Java should be used.

The remainder of this chapter will provide an overview of the data science tasks and Java support demonstrated in the book. Each section is only able to present a brief introduction to the topics and the available support. The subsequent chapter will go into considerably more depth regarding these topics.

Acquiring data for an application

Data acquisition is an important step in the data analysis process. When data is acquired, it is often in a specialized form and its contents may be inconsistent or different from an application's need. There are many sources of data, which are found on the Internet. Several examples will be demonstrated in `Chapter 2`, *Data Acquisition*.

Data may be stored in a variety of formats. Popular formats for text data include HTML, **Comma Separated Values** (**CSV**), **JavaScript Object Notation** (**JSON**), and XML. Image and audio data are stored in a number of formats. However, it is frequently necessary to convert one data format into another format, typically plain text.

For example, JSON (`http://www.JSON.org/`) is stored using blocks of curly braces containing key-value pairs. In the following example, parts of a YouTube result is shown:

```
{
  "kind": "youtube#searchResult",
  "etag": etag,
  "id": {
    "kind": string,
    "videoId": string,
    "channelId": string,
    "playlistId": string
  },
  ...
}
```

Data is acquired using techniques such as **processing live streams**, **downloading compressed files**, and through **screen scraping**, where the information on a web page is extracted. **Web crawling** is a technique where a program examines a series of web pages, moving from one page to another, acquiring the data that it needs.

With many popular media sites, it is necessary to acquire a user ID and password to access data. A commonly used technique is **OAuth,** which is an open standard used to authenticate users to many different websites. The technique delegates access to a server resource and works over HTTPS. Several companies use OAuth 2.0, including PayPal, Facebook, Twitter, and Yelp

The importance and process of cleaning data

Once the data has been acquired, it will need to be cleaned. Frequently, the data will contain errors, duplicate entries, or be inconsistent. It often needs to be converted to a simpler data type such as text. **Data cleaning** is often referred to as **data wrangling**, **reshaping,** or **munging**. They are effectively synonyms.

When data is cleaned, there are several tasks that often need to be performed, including checking its validity, accuracy, completeness, consistency, and uniformity. For example, when the data is incomplete, it may be necessary to provide substitute values.

Consider CSV data. It can be handled in one of several ways. We can use simple Java techniques such as the `String` class' `split` method. In the following sequence, a string array, `csvArray`, is assumed to hold comma-delimited data. The `split` method populates a second array, `tokenArray`.

```
for(int i=0; i<csvArray.length; i++) {
    tokenArray[i] = csvArray[i].split(",");
}
```

More complex data types require APIs to retrieve the data. For example, in `Chapter 3`, *Data Cleaning*, we will use the Jackson Project (`https://github.com/FasterXML/jackson`) to retrieve fields from a JSON file. The example uses a file containing a JSON-formatted presentation of a person, as shown next:

```
{
 "firstname":"Smith",
 "lastname":"Peter",
 "phone":8475552222,
 "address":["100 Main Street","Corpus","Oklahoma"]
}
```

The code sequence that follows shows how to extract the values for fields of a person. A parser is created, which uses `getCurrentName` to retrieve a field name. If the name is `firstname`, then the `getText` method returns the value for that field. The other fields are handled in a similar manner.

```
try {
    JsonFactory jsonfactory = new JsonFactory();
    JsonParser parser = jsonfactory.createParser(
        new File("Person.json"));
    while (parser.nextToken() != JsonToken.END_OBJECT) {
        String token = parser.getCurrentName();
        if ("firstname".equals(token)) {
            parser.nextToken();
            String fname = parser.getText();
            out.println("firstname : " + fname);
        }
        ...
    }
    parser.close();
} catch (IOException ex) {
    // Handle exceptions
}
```

The output of this example is as follows:

```
firstname : Smith
```

Simple data cleaning may involve converting the text to lowercase, replacing certain text with blanks, and removing multiple whitespace characters with a single blank. One way of doing this is shown next, where a combination of the `String` class' `toLowerCase`, `replaceAll`, and `trim` methods are used. Here, a string containing dirty text is processed:

```
dirtyText = dirtyText
    .toLowerCase()
    .replaceAll("[\\d[^\\w\\s]]+", "
    .trim();
while(dirtyText.contains("  ")){
      dirtyText = dirtyText.replaceAll("  ", " ");
}
```

Stop words are words such as *the*, *and*, or *but* that do not always contribute to the analysis of text. Removing these stop words can often improve the results and speed up the processing.

The **LingPipe API** can be used to remove stop words. In the next code sequence, a `TokenizerFactory` class instance is used to tokenize text. Tokenization is the process of returning individual words. The `EnglishStopTokenizerFactory` class is a special tokenizer that removes common English stop words.

```
text = text.toLowerCase().trim();
TokenizerFactory fact = IndoEuropeanTokenizerFactory.INSTANCE;
fact = new EnglishStopTokenizerFactory(fact);
Tokenizer tok = fact.tokenizer(
    text.toCharArray(), 0, text.length());
for(String word : tok){
      out.print(word + " ");
}
```

Consider the following text, which was pulled from the book, Moby Dick:

```
Call me Ishmael. Some years ago- never mind how long precisely - having
little or no money in my purse, and nothing particular to interest me
on shore, I thought I would sail about a little and see the watery part
of the world.
```

The output will be as follows:

```
call me ishmael . years ago - never mind how long precisely - having little
money my purse , nothing particular interest me shore , i thought i sail
little see watery part world .
```

These are just a couple of the data cleaning tasks discussed in `Chapter 3`, *Data Cleaning*.

Visualizing data to enhance understanding

The analysis of data often results in a series of numbers representing the results of the analysis. However, for most people, this way of expressing results is not always intuitive. A better way to understand the results is to create graphs and charts to depict the results and the relationship between the elements of the result.

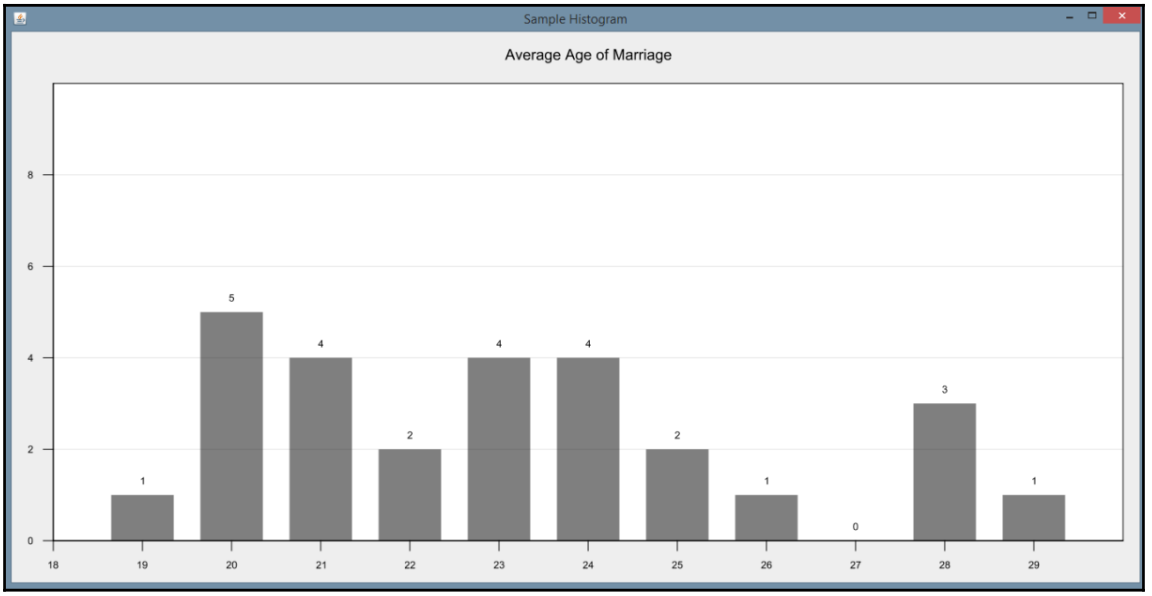
The human mind is often good at seeing patterns, trends, and outliers in visual representation. The large amount of data present in many data science problems can be analyzed using visualization techniques. Visualization is appropriate for a wide range of audiences ranging from analysts to upper-level management to clientele. In this chapter, we present various visualization techniques and demonstrate how they are supported in Java.

In `Chapter 4`, *Data Visualization*, we illustrate how to create different types of graphs, plots, and charts. These examples use JavaFX using a free library called **GRAL** (`http://trac.erichseifert.de/gral/`).

Visualization allows users to examine large datasets in ways that provide insights that are not present in the mass of the data. Visualization tools helps us identify potential problems or unexpected data results and develop meaningful interpretations of the data.

For example, outliers, which are values that lie outside of the normal range of values, can be hard to spot from a sea of numbers. Creating a graph based on the data allows users to quickly see outliers. It can also help spot errors quickly and more easily classify data.

For example, the following chart might suggest that the upper two values should be outliers that need to be dealt with:



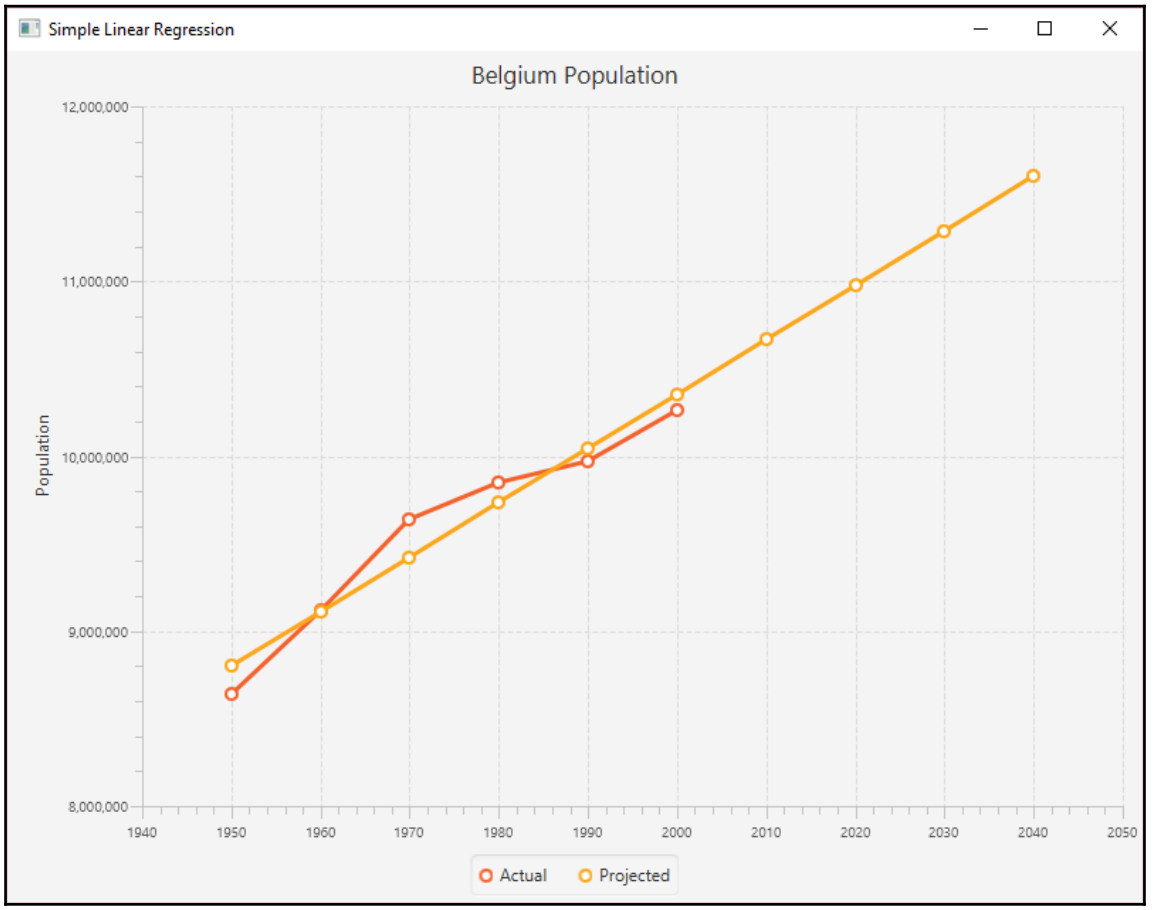
The use of statistical methods in data science

Statistical analysis is the key to many data science tasks. It is used for many types of analysis ranging from the computation of simple mean and medium to complex multiple regression analysis. `Chapter 5`, *Statistical Data Analysis Techniques*, introduces this type of analysis and the Java support available.

Statistical analysis is not always an easy task. In addition, advanced statistical techniques often require a particular mindset to fully comprehend, which can be difficult to learn. Fortunately, many techniques are not that difficult to use and various libraries mitigate some of these techniques' inherent complexity.

Regression analysis, in particular, is an important technique for analyzing data. The technique attempts to draw a line that matches a set of data. An equation representing the line is calculated and can be used to predict future behavior. There are several types of regression analysis, including simple and multiple regression. They vary by the number of variables being considered.

The following graph shows the straight line that closely matches a set of data points representing the population of Belgium over several decades:



Simple statistical techniques, such as mean and standard deviation, can be computed using basic Java. They can also be handled by libraries such as Apache Commons. For example, to calculate the median, we can use the Apache Commons `DescriptiveStatistics` class. This is illustrated next where the median of an array of doubles is calculated. The numbers are added to an instance of this class, as shown here:

```
double[] testData = {12.5, 18.3, 11.2, 19.0, 22.1, 14.3, 16.2,
    12.5, 17.8, 16.5, 12.5};
DescriptiveStatistics statTest =
    new SynchronizedDescriptiveStatistics();
for(double num : testData){
   statTest.addValue(num);
}
```

The `getPercentile` method returns the value stored at the percentile specified in its argument. To find the median, we use the value of `50`.

```
out.println("The median is " + statTest.getPercentile(50));
```

Our output is as follows:

```
The median is 16.2
```

In `Chapter 5`, *Statistical Data Analysis Techniques*, we will demonstrate how to perform regression analysis using the Apache Commons `SimpleRegression` class.

Machine learning applied to data science

Machine learning has become increasingly important for data science analysis as it has been for a multitude of other fields. A defining characteristic of machine learning is the ability of a model to be trained on a set of representative data and then later used to solve similar problems. There is no need to explicitly program an application to solve the problem. A model is a representation of the real-world object.

For example, customer purchases can be used to train a model. Subsequently, predictions can be made about the types of purchases a customer might subsequently make. This allows an organization to tailor ads and coupons for a customer and potentially providing a better customer experience.

Training can be performed in one of several different approaches:

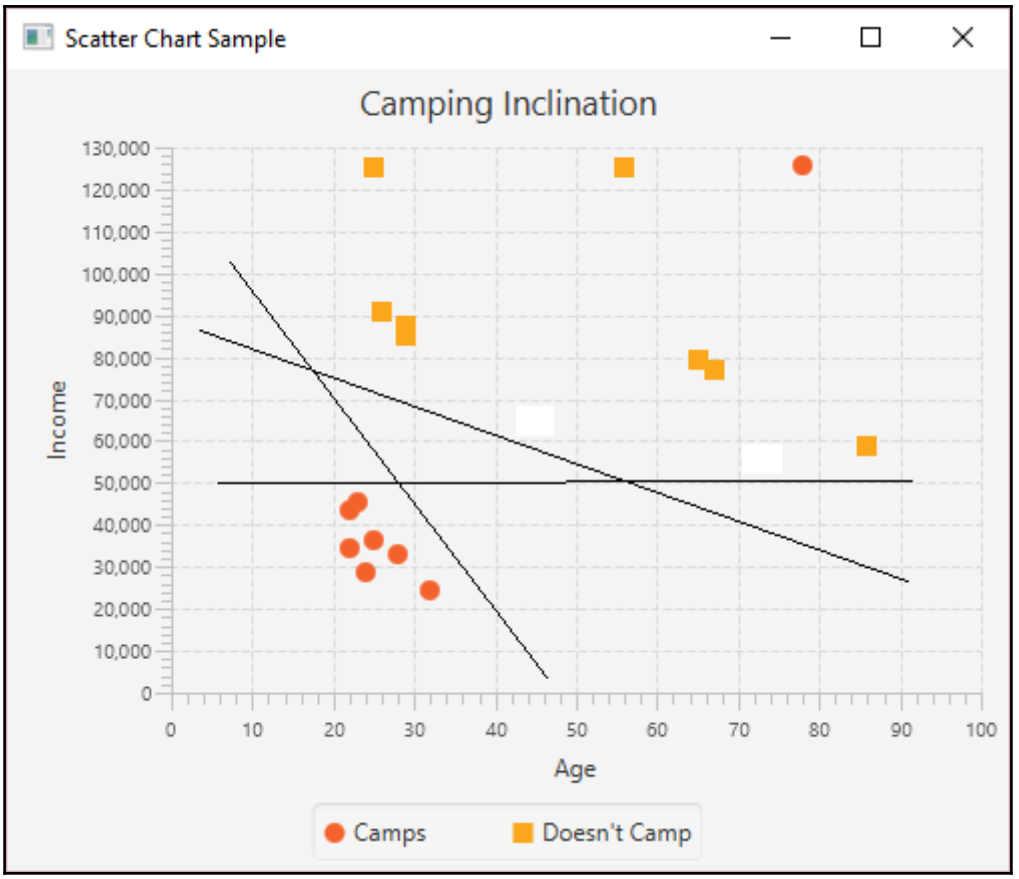
- **Supervised learning**: The model is trained with annotated, labeled, data showing corresponding correct results
- **Unsupervised learning**: The data does not contain results, but the model is expected to find relationships on its own
- **Semi-supervised**: A small amount of labeled data is combined with a larger amount of unlabeled data
- **Reinforcement learning**: This is similar to supervised learning, but a reward is provided for good results

There are several approaches that support machine learning. In `Chapter 6`, *Machine Learning*, we will illustrate three techniques:

- **Decision trees**: A tree is constructed using features of the problem as internal nodes and the results as leaves
- **Support vector machines**: This is used for classification by creating a hyperplane that partitions the dataset and then makes predictions
- **Bayesian networks**: This is used to depict probabilistic relationships between events

A **Support Vector Machine** (**SVM**) is used primarily for classification type problems. The approach creates a hyperplane to categorize data, which can be envisioned as a **geometric plane** that separates two regions. In a two-dimensional space, it will be a line. In a three-dimensional space, it will be a two-dimensional plane. In `Chapter 6`, *Machine Learning*, we will demonstrate how to use the approach using a set of data relating to the propensity of individuals to camp. We will use the Weka class, `SMO`, to demonstrate this type of analysis.

The following figure depicts a hyperplane using a distribution of two types of data points. The lines represent possible hyperplanes that separate these points. The lines clearly separate the data points except for one outlier.



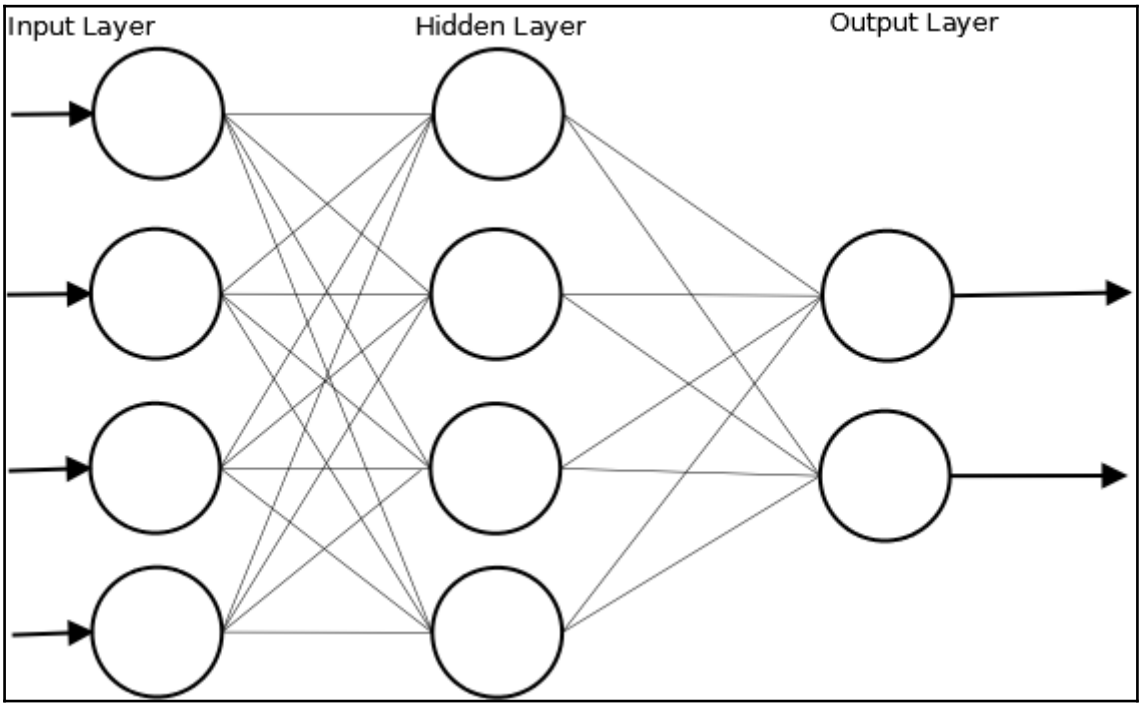
Once the model has been trained, the possible hyperplanes are considered and predictions can then be made using similar data.

Using neural networks in data science

An **Artificial Neural Network** (**ANN**), which we will call a **neural network**, is based on the neuron found in the brain. A **neuron** is a cell that has **dendrites** connecting it to input sources and other neurons. Depending on the input source, a weight allocated to a source, the neuron is activated, and then **fires** a signal down a dendrite to another neuron. A collection of neurons can be trained to respond to a set of input signals.

An artificial neuron is a node that has one or more inputs and a single output. Each input has a **weight** assigned to it that can change over time. A neural network can learn by feeding an input into a network, invoking an **activation function**, and comparing the results. This function combines the inputs and creates an output. If outputs of multiple neurons match the expected result, then the network has been trained correctly. If they don't match, then the network is modified.

A neural network can be visualized as shown in the following figure, where **Hidden Layer** is used to augment the process:



In `Chapter 7`, *Neural Networks*, we will use the `Weka` class, `MultilayerPerceptron`, to illustrate the creation and use of a **Multi Layer Perceptron** (**MLP**) network. As we will explain, this type of network is a feedforward neural network with multiple layers. The network uses supervised learning with backpropagation. The example uses a dataset called `dermatology.arff` that contains *366* instances that are used to diagnose erythemato-squamous diseases. It uses *34* attributes to classify the disease into one of the five different categories.

The dataset is split into a training set and a testing set. Once the data has been read, the MLP instance is created and initialized using the method to configure the attributes of the model, including how quickly the model is to learn and the amount of time spent training the model.

```
String trainingFileName = "dermatologyTrainingSet.arff";
String testingFileName = "dermatologyTestingSet.arff";

try (FileReader trainingReader = new FileReader(trainingFileName);
        FileReader testingReader =
            new FileReader(testingFileName)) {
    Instances trainingInstances = new Instances(trainingReader);
    trainingInstances.setClassIndex(
        trainingInstances.numAttributes() - 1);
    Instances testingInstances = new Instances(testingReader);
    testingInstances.setClassIndex(
        testingInstances.numAttributes() - 1);

    MultilayerPerceptron mlp = new MultilayerPerceptron();
    mlp.setLearningRate(0.1);
    mlp.setMomentum(0.2);
    mlp.setTrainingTime(2000);
    mlp.setHiddenLayers("3");
          mlp.buildClassifier(trainingInstances);
       ...
} catch (Exception ex) {
    // Handle exceptions
}
```

The model is then evaluated using the testing data:

```
Evaluation evaluation = new Evaluation(trainingInstances);
evaluation.evaluateModel(mlp, testingInstances);
```

The results can then be displayed:

```
System.out.println(evaluation.toSummaryString());
```

The truncated output of this example is shown here where the number of correctly and incorrectly identified diseases are listed:

```
Correctly Classified Instances 73 98.6486 %
Incorrectly Classified Instances 1 1.3514 %
```

The various attributes of the model can be tweaked to improve the model. In `Chapter 7`, *Neural Networks,* we will discuss this and other techniques in more depth.

Deep learning approaches

Deep learning networks are often described as neural networks that use multiple intermediate layers. Each layer will train on the outputs of a previous layer potentially identifying features and subfeatures of a dataset. The features refer to those aspects of the data that may be of interest. In `Chapter 8`, *Deep Learning*, we will examine these types of networks and how they can support several different data science tasks.

These networks often work with unstructured and unlabeled datasets, which is the vast majority of the data available today. A typical approach is to take the data, identify features, and then use these features and their corresponding layers to reconstruct the original dataset, thus validating the network. The **Restricted Boltzmann Machines** (**RBM**) is a good example of the application of this approach.

The deep learning network needs to ensure that the results are accurate and minimizes any error that can creep into the process. This is accomplished by adjusting the internal weights assigned to neurons based on what is known as **gradient descent**. This represents the slope of the weight changes. The approach modifies the weight so as to minimize the error and also speeds up the learning process.

There are several types of networks that have been classified as a deep learning network. One of these is an **autoencoder** network. In this network, the layers are symmetrical where the number of input values is the same as the number of output values and the intermediate layers effectively compress the data to a single smaller internal layer. Each layer of the autoencoder is a RBM.

This structure is reflected in the following example, which will extract the numbers found in a set of images containing hand-written numbers. The details of the complete example are not shown here, but notice that 1,000 input and output values are used along with internal layers consisting of RBMs. The size of the layers are specified in the `nOut` and `nIn` methods.

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .seed(seed)
        .iterations(numberOfIterations)
        .optimizationAlgo(
           OptimizationAlgorithm.LINE_GRADIENT_DESCENT)
        .list()
        .layer(0, new RBM.Builder()
            .nIn(numberOfRows * numberOfColumns).nOut(1000)
            .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
            .build())
        .layer(1, new RBM.Builder().nIn(1000).nOut(500)
            .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
```

```
        .build())
    .layer(2, new RBM.Builder().nIn(500).nOut(250)
        .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
        .build())
    .layer(3, new RBM.Builder().nIn(250).nOut(100)
        .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
        .build())
    .layer(4, new RBM.Builder().nIn(100).nOut(30)
        .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
        .build()) //encoding stops
    .layer(5, new RBM.Builder().nIn(30).nOut(100)
        .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
        .build()) //decoding starts
    .layer(6, new RBM.Builder().nIn(100).nOut(250)
        .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
        .build())
    .layer(7, new RBM.Builder().nIn(250).nOut(500)
        .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
        .build())
    .layer(8, new RBM.Builder().nIn(500).nOut(1000)
        .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
        .build())
    .layer(9, new OutputLayer.Builder(
            LossFunctions.LossFunction.RMSE_XENT).nIn(1000)
            .nOut(numberOfRows * numberOfColumns).build())
    .pretrain(true).backprop(true)
    .build();
```

Once the model has been trained, it can be used for predictive and searching tasks. With a search, the compressed middle layer can be used to match other compressed images that need to be classified.

Performing text analysis

The field of **Natural Language Processing** (**NLP**) is used for many different tasks including text searching, language translation, sentiment analysis, speech recognition, and classification to mention a few. Processing text is difficult due to a number of reasons, including the inherent ambiguity of natural languages.

There are several different types of processing that can be performed such as:

- **Identifying Stop words**: These are words that are common and may not be necessary for processing
- **Name Entity Recognition** (**NER**): This is the process of identifying elements of text such as people's names, location, or things
- **Parts of Speech** (**POS**): This identifies the grammatical parts of a sentence such as noun, verb, adjective, and so on
- **Relationships**: Here we are concerned with identifying how parts of text are related to each other, such as the subject and object of a sentence

As with most data science problems, it is important to preprocess and clean text. In `Chapter 9`, *Text Analysis*, we examine the support Java provides for this area of data science.

For example, we will use Apache's OpenNLP (`https://opennlp.apache.org/`) library to find the parts of speech. This is just one of the several NLP APIs that we could have used including LingPipe (`http://alias-i.com/lingpipe/`), Apache UIMA (`https://uima.apache.org/`), and Standford NLP (`http://nlp.stanford.edu/`). We chose OpenNLP because it is easy to use for this example.

In the following example, a model used to identify POS elements is found in the `en-pos-maxent.bin` file. An array of words is initialized and the POS model is created:

```
try (InputStream input = new FileInputStream(
        new File("en-pos-maxent.bin"));) {
    String sentence = "Let's parse this sentence.";
    ...
    String[] words;
    ...
    list.toArray(words);
    POSModel posModel = new POSModel(input);
    ...
} catch (IOException ex) {
    // Handle exceptions
}
```

The `tag` method is passed an array of `words` and returns an array of tags. The words and tags are then displayed.

```
String[] posTags = posTagger.tag(words);
for(int i=0; i<posTags.length; i++) {
    out.println(words[i] + " - " + posTags[i]);
}
```

The output for this example is as follows

```
Let's - NNP
parse - NN
this - DT
sentence. - NN
```

The abbreviations `NNP` and `DT` stand for a singular proper noun and determiner respectively. We examine several other NLP techniques in `Chapter 9`, *Text Analysis*.

Visual and audio analysis

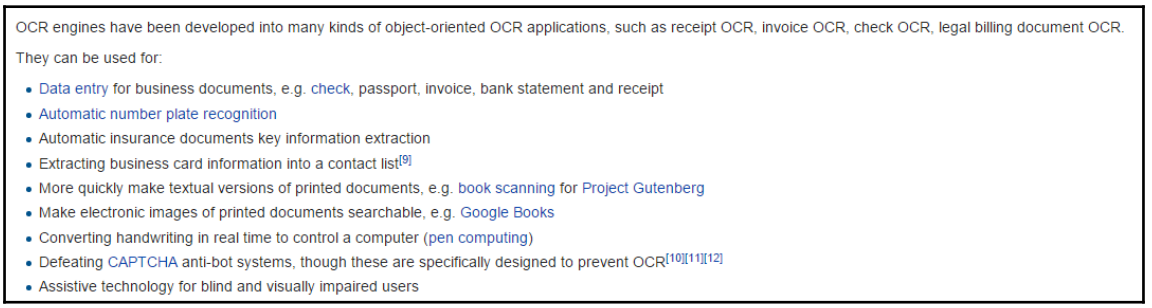
In `Chapter 10`, *Visual and Audio Analysis*, we demonstrate several Java techniques for processing sounds and images. We begin by demonstrating techniques for sound processing, including speech recognition and text-to-speech APIs. Specifically, we will use the FreeTTS (`http://freetts.sourceforge.net/docs/index.php`) API to convert text to speech. We also include a demonstration of the **CMU Sphinx** toolkit for speech recognition.

The **Java Speech API** (**JSAPI**) (`http://www.oracle.com/technetwork/java/index-140170.html`) supports speech technology. This API, created by third-party vendors, supports speech recognition and speech synthesizers. FreeTTS and Festival (`http://www.cstr.ed.ac.uk/projects/festival/`) are examples of vendors supporting JSAPI.

In the second part of the chapter, we examine image processing techniques such as facial recognition. This demonstration involves identifying faces within an image and is easy to accomplish using OpenCV (`http://opencv.org/`).

Also, in `Chapter 10`, *Visual and Audio Analysis,* we demonstrate how to extract text from images, a process known as **OCR**. A common data science problem involves extracting and analyzing text embedded in an image. For example, the information contained in license plate, road signs, and directions can be significant.

In the following example, explained in more detail in `Chapter 11`, *Mathematical and Parallel Techniques for Data Analysis* accomplishes OCR using Tess4j (`http://tess4j.sourceforge.net/`) a Java JNA wrapper for Tesseract OCR API. We perform OCR on an image captured from the Wikipedia article on OCR (`https://en.wikipedia.org/wiki/Optical_character_recognition#Applications`), shown here



OCR engines have been developed into many kinds of object-oriented OCR applications, such as receipt OCR, invoice OCR, check OCR, legal billing document OCR.

They can be used for:

- Data entry for business documents, e.g. check, passport, invoice, bank statement and receipt
- Automatic number plate recognition
- Automatic insurance documents key information extraction
- Extracting business card information into a contact list[9]
- More quickly make textual versions of printed documents, e.g. book scanning for Project Gutenberg
- Make electronic images of printed documents searchable, e.g. Google Books
- Converting handwriting in real time to control a computer (pen computing)
- Defeating CAPTCHA anti-bot systems, though these are specifically designed to prevent OCR[10][11][12]
- Assistive technology for blind and visually impaired users

The `ITesseract` interface provides numerous OCR methods. The `doOCR` method takes a file and returns a string containing the words found in the file as shown here:

```
ITesseract instance = new Tesseract();
try {
    String result = instance.doOCR(new File("OCRExample.png"));
    System.out.println(result);
} catch (TesseractException e) {
    System.err.println(e.getMessage());
}
```

A part of the output is shown next:

```
OCR engines nave been developed into many lunds oiobiectorlented OCR
applicatlons, sucn as reoeipt OCR, involoe OCR, check OCR, legal billing
document OCR
They can be used ior
- Data entry ior business documents, e g check, passport, involoe, bank
statement and receipt
- Automatic number plate recognnlon
```

As you can see, there are numerous errors in this example that need to be addressed. We build upon this example in `Chapter 11`, *Mathematical and Parallel Techniques for Data Analysis*, with a discussion of enhancements and considerations to ensure the OCR process is as effective as possible.

We will conclude the chapter with a discussion of NeurophStudio, a neural network Java-based editor, to classify images and perform image recognition. We train a neural network to recognize and classify faces in this section.

Improving application performance using parallel techniques

In `Chapter 11`, *Mathematical and Parallel Techniques for Data Analysis*, we consider some of the parallel techniques available for data science applications. Concurrent execution of a program can significantly improve performance. In relation to data science, these techniques range from low-level mathematical calculations to higher-level API-specific options.

This chapter includes a discussion of basic performance enhancement considerations. Algorithms and application architecture matter as much as enhanced code, and this should be considered when attempting to integrate parallel techniques. If an application does not behave in the expected or desired manner, any gains from parallel optimizing are irrelevant.

Matrix operations are essential to many data applications and supporting APIs. We will include a discussion in this chapter about matrix multiplication and how it is handled using a variety of approaches. Even though these operations are often hidden within the API, it can be useful to understand how they are supported.

One approach we demonstrate utilizes the Apache Commons Math API (`http://commons.apache.org/proper/commons-math/`). This API supports a large number of mathematical and statistical operations, including matrix multiplication. The following example illustrates how to perform matrix multiplication.

We first declare and initialize matrices `A` and `B`:

```
double[][] A = {
    {0.1950, 0.0311},
    {0.3588, 0.2203},
    {0.1716, 0.5931},
    {0.2105, 0.3242}};

double[][] B = {
    {0.0502, 0.9823, 0.9472},
    {0.5732, 0.2694, 0.916}};
```

Apache Commons uses the `RealMatrix` class to store a matrix. Next, we use the `Array2DRowRealMatrix` constructor to create the corresponding matrices for `A` and `B`:

```
RealMatrix aRealMatrix = new Array2DRowRealMatrix(A);
RealMatrix bRealMatrix = new Array2DRowRealMatrix(B);
```

We perform multiplication simply using the `multiply` method:

```
RealMatrix cRealMatrix = aRealMatrix.multiply(bRealMatrix);
```

Finally, we use a `for` loop to display the results:

```
for (int i = 0; i < cRealMatrix.getRowDimension(); i++) {
    System.out.println(cRealMatrix.getRowVector(i));
}
```

The output is as follows:

```
{0.02761552; 0.19992684; 0.2131916}
{0.14428772; 0.41179806; 0.54165016}
{0.34857924; 0.32834382; 0.70581912}
{0.19639854; 0.29411363; 0.4963528}
```

Another approach to concurrent processing involves the use of Java threads. Threads are used by APIs such as Aparapi when multiple CPUs or GPUs are not available.

Data science applications often take advantage of the map-reduce algorithm. We will demonstrate parallel processing by using Apache's Hadoop to perform map-reduce. Designed specifically for large datasets, Hadoop reduces processing time for large scale data science projects. We demonstrate a technique for calculating the average value of a large dataset.

We also include examples of APIs that support multiple processors, including CUDA and OpenCL. CUDA is supported using Java bindings for CUDA (JCuda) (`http://jcuda.org/`). We also discuss OpenCL and its Java support. The Aparapi API provides high-level support for using multiple CPUs or GPUs and we include a demonstration of Aparapi in support of matrix multiplication.

Assembling the pieces

In the final chapter of this book, we will tie together many of the techniques explored in the previous chapters. We will create a simple console-based application for acquiring data from Twitter and performing various types of data manipulation and analysis. Our goal in this chapter is to demonstrate a simple project exploring a variety of data science concepts and provide insights and considerations for future projects.

Specifically, the application developed in the final chapter performs several high-level tasks, including data acquisition, data cleaning, sentiment analysis, and basic statistical collection. We demonstrate these techniques using Java 8 Streams and focus on Java 8 approaches whenever possible.

Summary

Data science is a broad, diverse field of study and it would be impossible to explore exhaustively within this book. We hope to provide a solid understanding of important data science concepts and equip the reader for further study. In particular, this book will provide concrete examples of different techniques for all stages of data science related inquiries. This ranges from data acquisition and cleaning to detailed statistical analysis.

So let's start with a discussion of data acquisition and how Java supports it as illustrated in the next chapter.

2
Data Acquisition

It is never much fun to work with code that is not formatted properly or uses variable names that do not convey their intended purpose. The same can be said of data, except that bad data can result in inaccurate results. Thus, data acquisition is an important step in the analysis of data. Data is available from a number of sources but must be retrieved and ultimately processed before it can be useful. It is available from a variety of sources. We can find it in numerous public data sources as simple files, or it may be found in more complex forms across the Internet. In this chapter, we will demonstrate how to acquire data from several of these, including various Internet sites and several social media sites.

We can access data from the Internet by downloading specific files or through a process known as **web scraping**, which involves extracting the contents of a web page. We also explore a related topic known as **web crawling**, which involves applications that examine a web site to determine whether it is of interest and then follows embedded links to identify other potentially relevant pages.

We can also extract data from social media sites. These types of sites often hold a treasure trove of data that is readily available if we know how to access it. In this chapter, we will demonstrate how to extract data from several sites, including:

- Twitter
- Wikipedia
- Flickr
- YouTube

When extracting data from a site, many different data formats may be encountered. We will examine three basic types: text, audio, and video. However, even within text, audio, and video data, many formats exist. For audio data alone, there are 45 audio coding formats compared at `https://en.wikipedia.org/wiki/Comparison_of_audio_coding_formats`. For textual data, there are almost 300 formats listed at `http://fileinfo.com/filetypes/text`. In this chapter, we will focus on how to download and extract these types of text as plain text for eventual processing.

We will briefly examine different data formats, followed by an examination of possible data sources. We need this knowledge to demonstrate how to obtain data using different data acquisition techniques.

Understanding the data formats used in data science applications

When we discuss data formats, we are referring to content format, as opposed to the underlying file format, which may not even be visible to most developers. We cannot examine all available formats due to the vast number of formats available. Instead, we will tackle several of the more common formats, providing adequate examples to address the most common data retrieval needs. Specifically, we will demonstrate how to retrieve data stored in the following formats:

- HTML
- PDF
- CSV/TSV
- Spreadsheets
- Databases
- JSON
- XML

Some of these formats are well supported and documented elsewhere. For example, XML has been in use for years and there are several well-established techniques for accessing XML data in Java. For these types of data, we will outline the major techniques available and show a few examples to illustrate how they work. This will provide those readers who are not familiar with the technology some insight into their nature.

The most common data format is binary files. For example, Word, Excel, and PDF documents are all stored in binary. These require special software to extract information from them. Text data is also very common.

Overview of CSV data

Comma Separated Values (**CSV**) files, contain tabular data organized in a row-column format. The data, stored as plaintext, is stored in rows, also called **records**. Each record contains fields separated by commas. These files are also closely related to other delimited files, most notably **Tab-Separated Values** (**TSV**) files. The following is a part of a simple CSV file, and these numbers are not intended to represent any specific type of data:

```
JURISDICTION NAME,COUNT PARTICIPANTS,COUNT FEMALE,PERCENT FEMALE
10001,44,22,0.5
10002,35,19,0.54
10003,1,1,1
```

Notice that the first row contains header data to describe the subsequent records. Each value is separated by a comma and corresponds to the header in the same position. In `Chapter 3`, *Data Cleaning*, we will discuss CSV files in more depth and examine the support available for different types of delimiters.

Overview of spreadsheets

Spreadsheets are a form of tabular data where information is stored in rows and columns, much like a two-dimensional array. They typically contain numeric and textual information and use formulas to summarize and analyze their contents. Most people are familiar with Excel spreadsheets, but they are also found as part of other product suites, such as OpenOffice.

Spreadsheets are an important data source because they have been used for the past several decades to store information in many industries and applications. Their tabular nature makes them easy to process and analyze. It is important to know how to extract data from this ubiquitous data source so that we can take advantage of the wealth of information that is stored in them.

For some of our examples, we will use a simple Excel spreadsheet that consists of a series of rows containing an ID, along with minimum, maximum, and average values. These numbers are not intended to represent any specific type of data. The spreadsheet looks like this:

ID	Minimum	Maximum	Average
12345	45	89	65.55
23456	78	96	86.75
34567	56	89	67.44
45678	86	99	95.67

In `Chapter 3`, *Data Cleaning*, we will learn how to extract data from spreadsheets.

Overview of databases

Data can be found in **Database Management Systems** (**DBMS**), which, like spreadsheets, are ubiquitous. Java provides a rich set of options for accessing and processing data in a DBMS. The intent of this section is to provide a basic introduction to database access using Java.

We will demonstrate the essence of connecting to a database, storing information, and retrieving information using JDBC. For this example, we used the MySQL DBMS. However, it will work for other DBMSes as well with a change in the database driver. We created a database called `example` and a table called `URLTABLE` using the following command within the **MySQL Workbench**. There are other tools that can achieve the same results:

```
CREATE TABLE IF NOT EXISTS `URLTABLE` (
  `RecordID` INT(11) NOT NULL AUTO_INCREMENT,
  `URL` text NOT NULL,
  PRIMARY KEY (`RecordID`)
);
```

We start with a `try` block to handle exceptions. A driver is needed to connect to the DBMS. In this example, we used `com.mysql.jdbc.Driver`. To connect to the database, the `getConnection` method is used, where the database server location, user ID, and password are passed. These values depend on the DBMS used:

```
try {
    Class.forName("com.mysql.jdbc.Driver");
    String url = "jdbc:mysql://localhost:3306/example";
```

```
        connection = DriverManager.getConnection(url, "user ID",
            "password");
            ...
    } catch (SQLException | ClassNotFoundException ex) {
        // Handle exceptions
    }
```

Next, we will illustrate how to add information to the database and then how to read it. The SQL `INSERT` command is constructed in a string. The name of the MySQL database is `example`. This command will insert values into the `URLTABLE` table in the database where the question mark is a placeholder for the value to be inserted:

```
String insertSQL = "INSERT INTO  `example`.`URLTABLE` "
    + "(`url`) VALUES " + "(?);";
```

The `PreparedStatement` class represents an SQL statement to execute. The `prepareStatement` method creates an instance of the class using the `INSERT` SQL statement:

```
PreparedStatement stmt = connection.prepareStatement(insertSQL);
```

We then add URLs to the table using the `setString` method and the `execute` method. The `setString` method possesses two arguments. The first specifies the column index to insert the data and the second is the value to be inserted. The `execute` method does the actual insertion. We add two URLs in the next sequence:

```
stmt.setString(1, "https://en.wikipedia.org/wiki/Data_science");
stmt.execute();
stmt.setString(1,
  "https://en.wikipedia.org/wiki/Bishop_Rock,_Isles_of_Scilly");
stmt.execute();
```

To read the data, we use a SQL `SELECT` statement as declared in the `selectSQL` string. This will return all the rows and columns from the `URLTABLE` table. The `createStatement` method creates an instance of a `Statement` class, which is used for `INSERT` type statements. The `executeQuery` method executes the query and returns a `ResultSet` instance that holds the contents of the table:

```
String selectSQL = "select * from URLTABLE";
Statement statement = connection.createStatement();
ResultSet resultSet = statement.executeQuery(selectSQL);
```

The following sequence iterates through the table, displaying one row at a time. The argument of the `getString` method specifies that we want to use the second column of the result set, which corresponds to the URL field:

```
out.println("List of URLs");
while (resultSet.next()) {
    out.println(resultSet.getString(2));
}
```

The output of this example, when executed, is as follows:

```
List of URLs
https://en.wikipedia.org/wiki/Data_science
https://en.wikipedia.org/wiki/Bishop_Rock,_Isles_of_Scilly
```

If you need to empty the contents of the table, use the following sequence:

```
Statement statement = connection.createStatement();
statement.execute("TRUNCATE URLTABLE;");
```

This was a brief introduction to database access using Java. There are many resources available that will provide more in-depth coverage of this topic. For example, Oracle provides a more in-depth introduction to this topic at `https://docs.oracle.com/javase/tutorial/jdbc/`.

Overview of PDF files

The **Portable Document Format** (**PDF**) is a format not tied to a specific platform or software application. A PDF document can hold formatted text and images. PDF is an open standard, making it useful in a variety of places.

There are a large number of documents stored as PDF, making it a valuable source of data. There are several Java APIs that allow access to PDF documents, including **Apache POI** and **PDFBox**. Techniques for extracting information from a PDF document are illustrated in `Chapter 3`, *Data Cleaning*.

Overview of JSON

JavaScript Object Notation (**JSON**) (`http://www.JSON.org/`) is a data format used to interchange data. It is easy for humans or machines to read and write. JSON is supported by many languages, including Java, which has several JSON libraries listed at `http://www.JSON.org/`.

A JSON entity is composed of a set of name-value pairs enclosed in curly braces. We will use this format in several of our examples. In handling YouTube, we will use a JSON object, some of which is shown next, representing the results of a request from a YouTube video:

```
{
  "kind": "youtube#searchResult",
  "etag": "etag",
  "id": {
    "kind": string,
    "videoId": string,
    "channelId": string,
    "playlistId": string
  },
  ...
}
```

Accessing the fields and values of such a document is not hard and is illustrated in `Chapter 3`, *Data Cleaning*.

Overview of XML

Extensible Markup Language (**XML**) is a markup language that specifies a standard document format. Widely used to communicate between applications and across the Internet, XML is popular due to its relative simplicity and flexibility. Documents encoded in XML are character-based and easily read by machines and humans.

XML documents contain markup and content characters. These characters allow parsers to classify the information contained within the document. The document consists of tags, and elements are stored within the tags. Elements may also contain other markup tags and form child elements. Additionally, elements may contain attributes or specific characteristics stored as a name-and-value pair.

An XML document must be well-formed. This means it must follow certain rules such as always using closing tags and only a single root tag. Other rules are discussed at `https://en.wikipedia.org/wiki/XML#Well-formedness_and_error-handling`.

The **Java API for XML Processing** (**JAXP**) consists of three interfaces for parsing XML data. The **Document Object Model** (**DOM**) interface parses an XML document and returns a tree structure delineating the structure of the document. The DOM interface parses an entire document as a whole. Alternatively, the **Simple API for XML** (**SAX**) parses a document one element at a time. SAX is preferable when memory usage is a concern as DOM requires more resources to construct the tree. DOM, however, offers flexibility over SAX in that any element can be accessed at any time and in any order.

The third Java API is known as **Streaming API for XML** (**StAX**). This streaming model was designed to accommodate the best parts of DOM and SAX models by granting flexibility without sacrificing resources. StAX exhibits higher performance, with the trade-off being that access is only available to one location in a document at a time. StAX is the preferred technique if you already know how you want to process the document, but it is also popular for applications with limited available memory.

The following is a simple XML file. Each `<text>` represents a tag, labelling the element contained within the tags. In this case, the largest node in our file is `<music>` and contained within it are sets of song data. Each tag within a `<song>` tag describes another element corresponding to that song. Every tag will eventually have a closing tag, such as `</song>`. Notice that the first tag contains information about which XML version should be used to parse the file:

```
<?xml version="1.0"?>
<music>
   <song id="1234">
      <artist>Patton, Courtney</artist>
      <name>So This Is Life</name>
      <genre>Country</genre>
      <price>2.99</price>
   </song>
   <song id="5678">
      <artist>Eady, Jason</artist>
      <name>AM Country Heaven</name>
      <genre>Country</genre>
      <price>2.99</price>
   </song>
</music>
```

There are numerous other XML-related technologies. For example, we can validate a specific XML document using either a DTD document or XML schema writing specifically for that XML document. XML documents can be transformed into a different format using XLST.

Overview of streaming data

Streaming data refers to data generated in a continuous stream and accessed in a sequential, piece-by-piece manner. Much of the data the average Internet user accesses is streamed, including video and audio channels, or text and image data on social media sites. Streaming data is the preferred method when the data is new and changing quickly, or when large data collections are sought.

Streamed data is often ideal for data science research because it generally exists in large quantities and raw format. Much public streaming data is available for free and supported by Java APIs. In this chapter, we are going to examine how to acquire data from streaming sources, including Twitter, Flickr, and YouTube. Despite the use of different techniques and APIs, you will notice similarities between the techniques used to pull data from these sites.

Overview of audio/video/images in Java

There are a large number of formats used to represent images, videos, and audio. This type of data is typically stored in binary format. Analog audio streams are sampled and digitized. Images are often simply collections of bits representing the color of a pixel. The following are links that provide a more in-depth discussion of some of these formats:

- Audio: `https://en.wikipedia.org/wiki/Audio_file_format`
- Images:`https://en.wikipedia.org/wiki/Image_file_formats`
- Video: `https://en.wikipedia.org/wiki/Video_file_format`

Frequently, this type of data can be quite large and must be compressed. When data is compressed two approaches are used. The first is a lossless compression, where less space is used and there is no loss of information. The second is lossy, where information is lost. Losing information is not always a bad thing as sometimes the loss is not noticeable to humans.

As we will demonstrate in `Chapter 3`, *Data Cleaning*, this type of data often is compromised in an inconvenient fashion and may need to be cleaned. For example, there may be background noise in an audio recording or an image may need to be smoothed before it can be processed. Image smoothing is demonstrated in `Chapter 3`, *Data Cleaning*, using the OpenCV library.

Data acquisition techniques

In this section, we will illustrate how to acquire data from web pages. Web pages contain a potential bounty of useful information. We will demonstrate how to access web pages using several technologies, starting with a low-level approach supported by the `HttpUrlConnection` class. To find pages, a web crawler application is often used. Once a useful page has been identified, then information needs to be extracted from the page. This is often performed using an HTML parser. Extracting this information is important because it is often buried amid a clutter of HTML tags and JavaScript code.

Using the HttpUrlConnection class

The contents of a web page can be accessed using the `HttpUrlConnection` class. This is a low-level approach that requires the developer to do a lot of footwork to extract relevant content. However, he or she is able to exercise greater control over how the content is handled. In some situations, this approach may be preferable to using other API libraries.

We will demonstrate how to download the content of Wikipedia's data science page using this class. We start with a `try/catch` block to handle exceptions. A URL object is created using the data science URL string. The `openConnection` method will create a connection to the Wikipedia server as shown here:

```
try {
    URL url = new URL(
        "https://en.wikipedia.org/wiki/Data_science");
    HttpURLConnection connection = (HttpURLConnection)
        url.openConnection();
   ...
} catch (MalformedURLException ex) {
    // Handle exceptions
} catch (IOException ex) {
    // Handle exceptions
}
```

The `connection` object is initialized with an HTTP `GET` command. The `connect` method is then executed to connect to the server:

```
connection.setRequestMethod("GET");
connection.connect();
```

Assuming no errors were encountered, we can determine whether the response was successful using the `getResponseCode` method. A normal return value is `200`. The content of a web page can vary. For example, the `getContentType` method returns a string describing the page's content. The `getContentLength` method returns its length:

```
out.println("Response Code: " + connection.getResponseCode());
out.println("Content Type: " + connection.getContentType());
out.println("Content Length: " + connection.getContentLength());
```

Assuming that we get an HTML formatted page, the next sequence illustrates how to get this content. A `BufferedReader` instance is created where one line at a time is read in from the web site and appended to a `BufferedReader` instance. The buffer is then displayed:

```
InputStreamReader isr = new InputStreamReader((InputStream)
    connection.getContent());
BufferedReader br = new BufferedReader(isr);
StringBuilder buffer = new StringBuilder();
String line;
do {
    line = br.readLine();
    buffer.append(line + "\n");
} while (line != null);
out.println(buffer.toString());
```

The abbreviated output is shown here:

```
Response Code: 200
Content Type: text/html; charset=UTF-8
Content Length: -1
<!DOCTYPE html>
<html lang="en" dir="ltr" class="client-nojs">
<head>
<meta charset="UTF-8"/>
<title>Data science - Wikipedia, the free encyclopedia</title>
<script>document.documentElement.className =
...
"wgHostname":"mw1251"});});</script>
</body>
</html>
```

While this is feasible, there are easier methods for getting the contents of a web page. One of these techniques is discussed in the next section.

Web crawlers in Java

Web crawling is the process of traversing a series of interconnected web pages and extracting relevant information from those pages. It does this by isolating and then following links on a page. While there are many precompiled datasets readily available, it may still be necessary to collect data directly off the Internet. Some sources such as news sites are continually being updated and need to be revisited from time to time.

A web crawler is an application that visits various sites and collects information. The web crawling process consists of a series of steps:

1. Select a URL to visit
2. Fetch the page
3. Parse the page
4. Extract relevant content
5. Extract relevant URLs to visit

This process is repeated for each URL visited.

There are several issues that need to be considered when fetching and parsing a page such as:

- **Page importance**: We do not want to process irrelevant pages.
- **Exclusively HTML**: We will not normally follow links to images, for example.
- **Spider traps**: We want to bypass sites that may result in an infinite number of requests. This can occur with dynamically generated pages where one request leads to another.
- **Repetition**: It is important to avoid crawling the same page more than once.
- **Politeness**: Do not make an excessive number of requests to a website. Observe the `robot.txt` files; they specify which parts of a site should not be crawled.

The process of creating a web crawler can be daunting. For all but the simplest needs, it is recommended that one of several open source web crawlers be used. A partial list follows:

- **Nutch**: `http://nutch.apache.org`
- **crawler4j**: `https://github.com/yasserg/crawler4j`
- **JSpider**: `http://j-spider.sourceforge.net/`
- **WebSPHINX**: `http://www.cs.cmu.edu/~rcm/websphinx/`
- **Heritrix**: `https://webarchive.jira.com/wiki/display/Heritrix`

We can either create our own web crawler or use an existing crawler and in this chapter we will examine both approaches. For specialized processing, it can be desirable to use a custom crawler. We will demonstrate how to create a simple web crawler in Java to provide more insight into how web crawlers work. This will be followed by a brief discussion of other web crawlers.

Creating your own web crawler

Now that we have a basic understanding of web crawlers, we are ready to create our own. In this simple web crawler, we will keep track of the pages visited using `ArrayList` instances. In addition, jsoup will be used to parse a web page and we will limit the number of pages we visit. Jsoup (`https://jsoup.org/`) is an open source HTML parser. This example demonstrates the basic structure of a web crawler and also highlights some of the issues involved in creating a web crawler.

We will use the `SimpleWebCrawler` class, as declared here:

```
public class SimpleWebCrawler {

    private String topic;
    private String startingURL;
    private String urlLimiter;
    private final int pageLimit = 20;
    private ArrayList<String> visitedList = new ArrayList<>();
    private ArrayList<String> pageList = new ArrayList<>();
    ...
    public static void main(String[] args) {
        new SimpleWebCrawler();
    }

}
```

The instance variables are detailed here:

Variable	Use
`topic`	The keyword that needs to be in a page for the page to be accepted
`startingURL`	The URL of the first page
`urlLimiter`	A string that must be contained in a link before it will be followed
`pageLimit`	The maximum number of pages to retrieve
`visitedList`	The `ArrayList` containing pages that have already been visited
`pageList`	An `ArrayList` containing the URLs of the pages of interest

In the `SimpleWebCrawler` constructor, we initialize the instance variables to begin the search from the Wikipedia page for Bishop Rock, an island off the coast of Italy. This was chosen to minimize the number of pages that might be retrieved. As we will see, there are many more Wikipedia pages dealing with Bishop Rock than one might think.

The `urlLimiter` variable is set to `Bishop_Rock`, which will restrict the embedded links to follow to just those containing that string. Each page of interest must contain the value stored in the `topic` variable. The `visitPage` method performs the actual crawl:

```
public SimpleWebCrawler() {
    startingURL = https://en.wikipedia.org/wiki/Bishop_Rock, "
        + "Isles_of_Scilly";
    urlLimiter = "Bishop_Rock";
    topic = "shipping route";
    visitPage(startingURL);
}
```

In the `visitPage` method, the `pageList` ArrayList is checked to see whether the maximum number of accepted pages has been exceeded. If the limit has been exceeded, then the search terminates:

```
public void visitPage(String url) {
    if (pageList.size() >= pageLimit) {
        return;
    }
   ...
}
```

If the page has already been visited, then we ignore it. Otherwise, it is added to the visited list:

```
if (visitedList.contains(url)) {
    // URL already visited
} else {
    visitedList.add(url);
        ...
}
```

`Jsoup` is used to parse the page and return a `Document` object. There are many different exceptions and problems that can occur such as a malformed URL, retrieval timeouts, or simply bad links. The `catch` block needs to handle these types of problems. We will provide a more in-depth explanation of jsoup in web scraping in Java:

```
try {
    Document doc = Jsoup.connect(url).get();
        ...
    }
} catch (Exception ex) {
    // Handle exceptions
}
```

If the document contains the topic text, then the link is displayed and added to the `pageList` ArrayList. Each embedded link is obtained, and if the link contains the limiting text, then the `visitPage` method is called recursively:

```
if (doc.text().contains(topic)) {
    out.println((pageList.size() + 1) + ": [" + url + "]");
    pageList.add(url);

    // Process page links
    Elements questions = doc.select("a[href]");
    for (Element link : questions) {
        if (link.attr("href").contains(urlLimiter)) {
            visitPage(link.attr("abs:href"));
        }
    }
}
```

This approach only examines links in those pages that contain the topic text. Moving the `for` loop outside of the if statement will test the links for all pages.

The output follows:

```
1: [https://en.wikipedia.org/wiki/Bishop_Rock,_Isles_of_Scilly]
2: [https://en.wikipedia.org/wiki/Bishop_Rock_Lighthouse]
3:
[https://en.wikipedia.org/w/index.php?title=Bishop_Rock,_Isles_of_Scilly&ol
did=717634231#Lighthouse]
4:
[https://en.wikipedia.org/w/index.php?title=Bishop_Rock,_Isles_of_Scilly&di
ff=prev&oldid=717634231]
5:
[https://en.wikipedia.org/w/index.php?title=Bishop_Rock,_Isles_of_Scilly&ol
did=716622943]
6:
[https://en.wikipedia.org/w/index.php?title=Bishop_Rock,_Isles_of_Scilly&di
ff=prev&oldid=716622943]
7:
[https://en.wikipedia.org/w/index.php?title=Bishop_Rock,_Isles_of_Scilly&ol
did=716608512]
8:
[https://en.wikipedia.org/w/index.php?title=Bishop_Rock,_Isles_of_Scilly&di
ff=prev&oldid=716608512]
...
20:
[https://en.wikipedia.org/w/index.php?title=Bishop_Rock,_Isles_of_Scilly&di
ff=prev&oldid=716603919]
```

In this example, we did not save the results of the crawl in an external source. Normally this is necessary and can be stored in a file or database.

Using the crawler4j web crawler

Here we will illustrate the use of the crawler4j (`https://github.com/yasserg/crawler4j`) web crawler. We will use an adapted version of the basic crawler found at `https://github.com/yasserg/crawler4j/tree/master/src/test/java/edu/uci/ics/crawler4j/examples/basic`. We will create two classes: `CrawlerController` and `SampleCrawler`. The former class set ups the crawler while the latter contains the logic that controls what pages will be processed.

As with our previous crawler, we will crawl the Wikipedia article dealing with Bishop Rock. The results using this crawler will be smaller as many extraneous pages are ignored.

Let's look at the `CrawlerController` class first. There are several parameters that are used with the crawler as detailed here:

- **Crawl storage folder**: The location where crawl data is stored
- **Number of crawlers**: This controls the number of threads used for the crawl
- **Politeness delay**: How many seconds to pause between requests
- **Crawl depth**: How deep the crawl will go
- **Maximum number of pages to fetch**: How many pages to fetch
- **Binary data**: Whether to crawl binary data such as PDF files

The basic class is shown here:

```
public class CrawlerController {

  public static void main(String[] args) throws Exception {
    int numberOfCrawlers = 2;
    CrawlConfig config = new CrawlConfig();
    String crawlStorageFolder = "data";

    config.setCrawlStorageFolder(crawlStorageFolder);
    config.setPolitenessDelay(500);
    config.setMaxDepthOfCrawling(2);
    config.setMaxPagesToFetch(20);
    config.setIncludeBinaryContentInCrawling(false);
    ...
  }
}
```

Next, the `CrawlController` class is created and configured. Notice the `RobotstxtConfig` and `RobotstxtServer` classes used to handle `robot.txt` files. These files contain instructions that are intended to be read by a web crawler. They provide direction to help a crawler to do a better job such as specifying which parts of a site should not be crawled. This is useful for auto generated pages:

```
    PageFetcher pageFetcher = new PageFetcher(config);
    RobotstxtConfig robotstxtConfig = new RobotstxtConfig();
    RobotstxtServer robotstxtServer =
        new RobotstxtServer(robotstxtConfig, pageFetcher);
    CrawlController controller =
        new CrawlController(config, pageFetcher, robotstxtServer);
```

The crawler needs to start at one or more pages. The `addSeed` method adds the starting pages. While we used the method only once here, it can be used as many times as needed:

```
    controller.addSeed(
```

```
        "https://en.wikipedia.org/wiki/Bishop_Rock,_Isles_of_Scilly");
```

The `start` method will begin the crawling process:

```
controller.start(SampleCrawler.class, numberOfCrawlers);
```

The `SampleCrawler` class contains two methods of interest. The first is the `shouldVisit` method that determines whether a page will be visited and the `visit` method that actually handles the page. We start with the class declaration and the declaration of a Java regular expression class `Pattern` object. It will be one way of determining whether a page will be visited. In this declaration, standard images are specified and will be ignored:

```
public class SampleCrawler extends WebCrawler {
    private static final Pattern IMAGE_EXTENSIONS =
        Pattern.compile(".*\\.(bmp|gif|jpg|png)$");

    ...
}
```

The `shouldVisit` method is passed a reference to the page where this URL was found along with the URL. If any of the images match, the method returns `false` and the page is ignored. In addition, the URL must start with `https://en.wikipedia.org/wiki/`. We added this to restrict our searches to the Wikipedia website:

```
public boolean shouldVisit(Page referringPage, WebURL url) {
    String href = url.getURL().toLowerCase();
    if (IMAGE_EXTENSIONS.matcher(href).matches()) {
        return false;
    }
    return href.startsWith("https://en.wikipedia.org/wiki/");
}
```

The `visit` method is passed a `Page` object representing the page being visited. In this implementation, only those pages containing the string `shipping route` will be processed. This further restricts the pages visited. When we find such a page, its `URL`, `Text`, and `Text length` are displayed:

```
public void visit(Page page) {
    String url = page.getWebURL().getURL();

    if (page.getParseData() instanceof HtmlParseData) {
        HtmlParseData htmlParseData =
            (HtmlParseData) page.getParseData();
        String text = htmlParseData.getText();
        if (text.contains("shipping route")) {
            out.println("\nURL: " + url);
```

```
                out.println("Text: " + text);
                out.println("Text length: " + text.length());
            }
        }
    }
```

The following is the truncated output of the program when executed:

```
URL: https://en.wikipedia.org/wiki/Bishop_Rock,_Isles_of_Scilly
Text: Bishop Rock, Isles of Scilly...From Wikipedia, the free encyclopedia
... Jump to: ... navigation, search For the Bishop Rock in the Pacific
Ocean, see Cortes Bank. Bishop Rock Bishop Rock Lighthouse (2005)
...
Text length: 14677
```

Notice that only one page was returned. This web crawler was able to identify and ignore previous versions of the main web page.

We could perform further processing, but this example provides some insight into how the API works. Significant amounts of information can be obtained when visiting a page. In the example, we only used the URL and the length of the text. The following is a sample of other data that you may be interested in obtaining:

- URL path
- Parent URL
- Anchor
- HTML text
- Outgoing links
- Document ID

Web scraping in Java

Web scraping is the process of extracting information from a web page. The page is typically formatted using a series of HTML tags. An HTML parser is used to navigate through a page or series of pages and to access the page's data or metadata.

Jsoup (`https://jsoup.org/`) is an open source Java library that facilitates extracting and manipulating HTML documents using an HTML parser. It is used for a number of purposes, including web scraping, extracting specific elements from an HTML page, and cleaning up HTML documents.

There are several ways of obtaining an HTML document that may be useful. The HTML document can be extracted from a:

- URL
- String
- File

The first approach is illustrated next where the Wikipedia page for data science is loaded into a `Document` object. This `Jsoup` object represents the HTML document. The `connect` method connects to the site and the `get` method retrieves the `document`:

```
try {
    Document document = Jsoup.connect(
        "https://en.wikipedia.org/wiki/Data_science").get();
    ...
 } catch (IOException ex) {
    // Handle exception
}
```

Loading from a file uses the `File` class as shown next. The overloaded `parse` method uses the file to create the `document` object:

```
try {
    File file = new File("Example.html");
    Document document = Jsoup.parse(file, "UTF-8", "");
    ...
} catch (IOException ex) {
    // Handle exception
}
```

The `Example.html` file follows:

```
<html>
<head><title>Example Document</title></head>
<body>
<p>The body of the document</p>
Interesting Links:
<br>
<a href="https://en.wikipedia.org/wiki/Data_science">Data Science</a>
<br>
<a href="https://en.wikipedia.org/wiki/Jsoup">Jsoup</a>
```

```
<br>
Images:
<br>
 <img src="eyechart.jpg" alt="Eye Chart">
</body>
</html>
```

To create a `Document` object from a string, we will use the following sequence where the `parse` method processes the string that duplicates the previous HTML file:

```
String html = "<html>\n"
    + "<head><title>Example Document</title></head>\n"
    + "<body>\n"
    + "<p>The body of the document</p>\n"
    + "Interesting Links:\n"
    + "<br>\n"
    + "<a href="https://en.wikipedia.org/wiki/Data_science">" +
      "DataScience</a>\n"
    + "<br>\n"
    + "<a href="https://en.wikipedia.org/wiki/Jsoup">" +
      "Jsoup</a>\n"
    + "<br>\n"
    + "Images:\n"
    + "<br>\n"
    + " <img src="eyechart.jpg" alt="Eye Chart"> \n"
    + "</body>\n"
    + "</html>";
Document document = Jsoup.parse(html);
```

The `Document` class possesses a number of useful methods. The `title` method returns the title. To get the text contents of the document, the `select` method is used. This method uses a string specifying the element of a document to retrieve:

```
String title = document.title();
out.println("Title: " + title);
Elements element = document.select("body");
out.println("  Text: " + element.text());
```

The output for the Wikipedia data science page is shown here. It has been shortened to conserve space:

```
Title: Data science - Wikipedia, the free encyclopedia
Text: Data science From Wikipedia, the free encyclopedia Jump to:
navigation, search Not to be confused with information science. Part of a
...
policy About Wikipedia Disclaimers Contact Wikipedia Developers Cookie
statement Mobile view
```

The parameter type of the `select` method is a string. By using a string, the type of information selected is easily changed. Details on how to formulate this string are found at the jsoup Javadocs for the `Selector` class at `https://jsoup.org/apidocs/`:

We can use the `select` method to retrieve the images in a document, as shown here:

```
Elements images = document.select("img[src$=.png]");
for (Element image : images) {
    out.println("\nImage: " + image);
}
```

The output for the Wikipedia data science page is shown here. It has been shortened to conserve space:

```
Image: <img alt="Data Visualization" src="//upload.wikimedia.org/...>
Image: <img alt=""
src="//upload.wikimedia.org/wikipedia/commons/thumb/b/ba/...>
```

Links can be easily retrieved as shown next:

```
Elements links = document.select("a[href]");
for (Element link : links) {
    out.println("Link: " + link.attr("href")
        + " Text: " + link.text());
}
```

The output for the `Example.html` page is shown here:

```
Link: https://en.wikipedia.org/wiki/Data_science Text: Data Science
Link: https://en.wikipedia.org/wiki/Jsoup Text: Jsoup
```

jsoup possesses many additional capabilities. However, this example demonstrates the web scraping process. There are also other Java HTML parsers available. A comparison of Java HTML parser, among others, can be found at `https://en.wikipedia.org/wiki/Comparison_of_HTML_parsers`.

Using API calls to access common social media sites

Social media contain a wealth of information that can be processed and is used by many data analysis applications. In this section, we will illustrate how to access a few of these sources using their Java APIs. Most of them require some sort of access key, which is normally easy to obtain. We start with a discussion on the `OAuth` class, which provides one approach to authenticating access to a data source.

When working with the type of data source, it is important to keep in mind that the data is not always public. While it may be accessible, the owner of the data may be an individual who does not necessarily want the information shared. Most APIs provide a means to determine how the data can be distributed, and these requests should be honored. When private information is used, permission from the author must be obtained.

In addition, these sites have limits on the number of requests that can be made. Keep this in mind when pulling data from a site. If these limits need to be exceeded, then most sites provide a way of doing this.

Using OAuth to authenticate users

OAuth is an open standard used to authenticate users to many different websites. A resource owner effectively delegates access to a server resource without having to share their credentials. It works over HTTPS. OAuth 2.0 succeeded OAuth and is not backwards compatible. It provides client developers a simple way of providing authentication. Several companies use OAuth 2.0 including PayPal, Comcast, and Blizzard Entertainment.

A list of OAuth 2.0 providers is found at `https://en.wikipedia.org/wiki/List_of_OAuth_providers`. We will use several of these in our discussions.

Handing Twitter

The sheer volume of data and the popularity of the site, among celebrities and the general public alike, make Twitter a valuable resource for mining social media data. Twitter is a popular social media platform allowing users to read and post short messages called **tweets**. Twitter provides API support for posting and pulling tweets, including streaming data from all public users. While there are services available for pulling the entire set of public tweet data, we are going to examine other options that, while limiting in the amount of data retrieved at one time, are available at no cost.

We are going to focus on the Twitter API for retrieving streaming data. There are other options for retrieving tweets from a specific user as well as posting data to a specific account but we will not be addressing those in this chapter. The public stream API, at the default access level, allows the user to pull a sample of public tweets currently streaming on Twitter. It is possible to refine the data by specifying parameters to track keywords, specific users, and location.

We are going to use HBC, a Java HTTP client, for this example. You can download a sample HBC application at `https://github.com/twitter/hbc`. If you prefer to use a different HTTP client, ensure it will return incremental response data. The Apache HTTP client is one option. Before you can create the HTTP connection, you must first create a Twitter account and an application within that account. To get started with the app, visit apps.twitter.com. Once your app is created, you will be assigned a consumer key, consumer secret, access token, and access secret token. We will also use OAuth, as discussed previously in this chapter.

First, we will write a method to perform the authentication and request data from Twitter. The parameters for our method are the authentication information given to us by Twitter when we created our app. We will create a `BlockingQueue` object to hold our streaming data. For this example, we will set a default capacity of 10,000. We will also specify our endpoint and turn off stall warnings:

```
public static void streamTwitter(
    String consumerKey, String consumerSecret,
    String accessToken, String accessSecret)
        throws InterruptedException {

    BlockingQueue<String> statusQueue =
        new LinkedBlockingQueue<String>(10000);
    StatusesSampleEndpoint ending =
        new StatusesSampleEndpoint();
    ending.stallWarnings(false);
    ...
}
```

Next, we create an `Authentication` object using `OAuth1`, a variation of the `OAuth` class. We can then build our connection client and complete the HTTP connection:

```
Authentication twitterAuth = new OAuth1(consumerKey,
    consumerSecret, accessToken, accessSecret);
BasicClient twitterClient = new ClientBuilder()
        .name("Twitter client")
        .hosts(Constants.STREAM_HOST)
        .endpoint(ending)
        .authentication(twitterAuth)
```

```
            .processor(new StringDelimitedProcessor(statusQueue))
            .build();
    twitterClient.connect();
```

For the purposes of this example, we will simply read the messages received from the stream and print them to the screen. The messages are returned in JSON format and the decision of how to process them in a real application will depend upon the purpose and limitations of that application:

```
    for (int msgRead = 0; msgRead < 1000; msgRead++) {
      if (twitterClient.isDone()) {
        out.println(twitterClient.getExitEvent().getMessage());
        break;
      }

      String msg = statusQueue.poll(10, TimeUnit.SECONDS);
      if (msg == null) {
        out.println("Waited 10 seconds - no message received");
      } else {
        out.println(msg);
      }
    }
    twitterClient.stop();
```

To execute our method, we simply pass our authentication information to the `streamTwitter` method. For security purposes, we have replaced our personal keys here. Authentication information should always be protected:

```
    public static void main(String[] args) {

      try {
        SampleStreamExample.streamTwitter(
            myKey, mySecret, myToken, myAccess);
      } catch (InterruptedException e) {
        out.println(e);
      }
    }
```

Here is truncated sample data retrieved using the methods listed above. Your data will vary based upon Twitter's live stream, but it should resemble this example:

```
{"created_at":"Fri May 20 15:47:21 +0000
2016","id":733685552789098496,"id_str":"733685552789098496","text":"bwisit
si em bahala sya","source":"\u003ca href="http:\/\/twitter.com"
rel="nofollow"\u003eTwitter Web
...
ntions":[],"symbols":[]},"favorited":false,"retweeted":false,"filter_level"
:"low","lang":"tl","timestamp_ms":"1463759241660"}
```

Twitter also provides support for pulling all data for one specific user account, as well as posting data directly to an account. A REST API is also available and provides support for specific queries via the search API. These also use the OAuth standard and return data in JSON files.

Handling Wikipedia

Wikipedia (`https://www.wikipedia.org/`) is a useful source of text and image type information. It is an Internet encyclopedia that hosts 38 million articles written in over 250 languages (`https://en.wikipedia.org/wiki/Wikipedia`). As such, it is useful to know how to programmatically access its contents.

MediaWiki is an open source wiki application that supports wiki type sites. It is used to support Wikipedia and many other sites. The MediaWiki API (`http://www.mediawiki.org/wiki/API`) provides access to a wiki's data and metadata over HTTP. An application, using this API, can log in, read data, and post changes to a site.

There are several Java APIs that support programmatic access to a wiki site as listed at `https://www.mediawiki.org/wiki/API:Client_code#Java`. To demonstrate Java access to a wiki we will use Bliki found at `https://bitbucket.org/axelclk/info.bliki.wiki/wiki/Home`. It provides good access and is easy to use for most basic operations.

The MediaWiki API is complex and has many features. The intent of this section is to illustrate the basic process of obtaining text from a Wikipedia article using this API. It is not possible to cover the API completely here.

We will use the following classes from the `info.bliki.api` and `info.bliki.wiki.model` packages:

- `Page`: Represents a retrieved page
- `User`: Represents a user
- `WikiModel`: Represents the wiki

Javadocs for Bliki are found at `http://www.javadoc.io/doc/info.bliki.wiki/bliki-core/3.1.0`.

The following example has been adapted from `http://www.integratingstuff.com/2012/04/06/hook-into-wikipedia-using-java-and-the-mediawiki-api/`. This example will access the English Wikipedia page for the subject, data science. We start by creating an instance of the `User` class. The first two arguments of the three-argument constructor are the `user ID` and `password`, respectively. In this case, they are empty strings. This combination allows us to read a page without having to set up an account. The third argument is the URL for the MediaWiki API page:

```
User user = new User("", "",
    "http://en.wikipedia.org/w/api.php");
user.login();
```

An account will enable us to modify the document. The `queryContent` method returns a list of `Page` objects for the subjects found in a string array. Each string should be the title of a page. In this example, we access a single page:

```
String[] titles = {"Data science"};
List<Page> pageList = user.queryContent(titles);
```

Each `Page` object contains the content of a page. There are several methods that will return the contents of the page. For each page, a `WikiModel` instance is created using the two-argument constructor. The first argument is the image base URL and the second argument is the link base URL. These URLs use Wiki variables called `image` and `title`, which will be replaced when creating links:

```
for (Page page : pageList) {
    WikiModel wikiModel = new WikiModel("${image}",
        "${title}");
    ...
}
```

The `render` method will take the wiki page and render it to HTML. There is also a method to render the page to a PDF document:

```
String htmlText = wikiModel.render(page.toString());
```

The HTML text is then displayed:

```
out.println(htmlText);
```

A partial listing of the output follows:

```
<p>PageID: 35458904; NS: 0; Title: Data science;
Image url:
Content:
{{distinguish}}
{{Use dmy dates}}
{{Data Visualization}}</p>
<p><b>Data science</b> is an interdisciplinary field about processes and
systems to extract <a href="Knowledge" >knowledge</a>
...
```

We can also obtain basic information about the article using one of several methods as shown here:

```
out.println("Title: " + page.getTitle() + "\n" +
    "Page ID: " + page.getPageid() + "\n" +
    "Timestamp: " + page.getCurrentRevision().getTimestamp());
```

It is also possible to obtain a list of references in the article and a list of the headers. Here, a list of the references is displayed:

```
List <Reference> referenceList = wikiModel.getReferences();
out.println(referenceList.size());
for(Reference reference : referenceList) {
    out.println(reference.getRefString());
}
```

The following illustrates the process of getting the section headers:

```
ITableOfContent toc = wikiModel.getTableOfContent();
List<SectionHeader> sections = toc.getSectionHeaders();
for(SectionHeader sh : sections) {
    out.println(sh.getFirst());
}
```

The entire content of Wikipedia can be downloaded. This process is discussed at `https://en.wikipedia.org/wiki/Wikipedia:Database_download`. It may be desirable to set up your own Wikipedia server to handle your request.

Handling Flickr

Flickr (`https://www.flickr.com/`) is an online photo management and sharing application. It is a possible source for images and videos. The Flickr Developer Guide (`https://www.flickr.com/services/developer/`) is a good starting point to learn more about Flickr's API.

One of the first steps to using the Flickr API is to request an API key. This key is used to sign your API requests. The process to obtain a key starts at `https://www.flickr.com/services/apps/create/`. Both commercial and noncommercial keys are available. When you obtain a key you will also get a "secret." Both of these are required to use the API.

We will illustrate the process of locating and downloading images from Flickr. The process involves:

- Creating a Flickr class instance
- Specifying the search parameters for a query
- Performing the search
- Downloading the image

A `FlickrException` or `IOException` may be thrown during this process. There are several APIs that support Flickr access. We will be using Flickr4Java, found at `https://github.com/callmeal/Flickr4Java`. The Flickr4Java Javadocs is found at `http://flickrj.sourceforge.net/api/`. We will start with a `try` block and the `apikey` and `secret` declarations:

```
try {
    String apikey = "Your API key";
    String secret = "Your secret";

} catch (FlickrException | IOException ex) {
    // Handle exceptions
}
```

The `Flickr` instance is created next, where the `apikey` and `secret` are supplied as the first two parameters. The last parameter specifies the transfer technique used to access Flickr servers. Currently, the REST transport is supported using the `REST` class:

```
Flickr flickr = new Flickr(apikey, secret, new REST());
```

To search for images, we will use the `SearchParameters` class. This class supports a number of criteria that will narrow down the number of images returned from a query and includes such criteria as latitude, longitude, media type, and user ID. In the following sequence, the `setBBox` method specifies the longitude and latitude for the search. The parameters are (in order): minimum longitude, minimum latitude, maximum longitude, and maximum latitude. The `setMedia` method specifies the type of media. There are three possible arguments---`"all"`, `"photos"`, and `"videos"`:

```
SearchParameters searchParameters = new SearchParameters();
searchParameters.setBBox("-180", "-90", "180", "90");
searchParameters.setMedia("photos");
```

The `PhotosInterface` class possesses a `search` method that uses the `SearchParameters` instance to retrieve a list of photos. The `getPhotosInterface` method returns an instance of the `PhotosInterface` class, as shown next. The `SearchParameters` instance is the first parameter. The second parameter determines how many photos are retrieved per page and the third parameter is the offset. A `PhotoList` class instance is returned:

```
PhotosInterface pi = new PhotosInterface(apikey, secret,
    new REST());
PhotoList<Photo> list = pi.search(searchParameters, 10, 0);
```

The next sequence illustrates the use of several methods to get information about the images retrieved. Each `Photo` instance is accessed using the `get` method. The title, image format, public flag, and photo URL are displayed:

```
out.println("Image List");
for (int i = 0; i < list.size(); i++) {
    Photo photo = list.get(i);
    out.println("Image: " + i +
        `"\nTitle: " + photo.getTitle() +
        "\nMedia: " + photo.getOriginalFormat() +
        "\nPublic: " + photo.isPublicFlag() +
        "\nUrl: " + photo.getUrl() +
        "\n");
}
out.println();
```

A partial listing is shown here where many of the specific values have been modified to protect the original data:

```
Image List
Image: 0
Title: XYZ Image
Media: jpg
Public: true
Url: https://flickr.com/photos/7723...@N02/269...
Image: 1
Title: IMG_5555.jpg
Media: jpg
Public: true
Url: https://flickr.com/photos/2665...@N07/264...
Image: 2
Title: DSC05555
Media: jpg
Public: true
Url: https://flickr.com/photos/1179...@N04/264...
```

The list of images returned by this example will vary since we used a fairly wide search range and images are being added all of the time.

There are two approaches that we can use to download an image. The first uses the image's URL and the second uses a `Photo` object. The image's URL can be obtained from a number of sources. We use the `Photo` class `getUrl` method for this example.

In the following sequence, we obtain an instance of `PhotosInterface` using its constructor to illustrate an alternate approach:

```
PhotosInterface pi = new PhotosInterface(apikey, secret,
    new REST());
```

We get the first `Photo` instance from the previous list and then its `getUrl` to get the image's URL. The `PhotosInterface` class's `getImage` method returns a `BufferedImage` object representing the image as shown here:

```
Photo currentPhoto = list.get(0);
BufferedImage bufferedImage =
    pi.getImage(currentPhoto.getUrl());
```

The image is then saved to a file using the `ImageIO` class:

```
File outputfile = new File("image.jpg");
ImageIO.write(bufferedImage, "jpg", outputfile);
```

The `getImage` method is overloaded. Here, the `Photo` instance and the size of the image desired are used as arguments to get the `BufferedImage` instance:

```
bufferedImage = pi.getImage(currentPhoto, Size.SMALL);
```

The image can be saved to a file using the previous technique.

The Flickr4Java API supports a number of other techniques for working with Flickr images.

Handling YouTube

YouTube is a popular video site where users can upload and share videos (`https://www.youtube.com/`). It has been used to share humorous videos, provide instructions on how to do any number of things, and share information among its viewers. It is a useful source of information as it captures the thoughts and ideas of a diverse group of people. This provides an interesting opportunity to analysis and gain insight into human behavior.

YouTube can serve as a useful source of videos and video metadata. A Java API is available to access its contents (`https://developers.google.com/youtube/v3/`). Detailed documentation of the API is found at `https://developers.google.com/youtube/v3/docs/`.

In this section, we will demonstrate how to search for videos by keyword and retrieve information of interest. We will also show how to download a video. To use the YouTube API, you will need a Google account, which can be obtained at `https://www.google.com/accounts/NewAccount`. Next, create an account in the Google Developer's Console (`https://console.developers.google.com/`). API access is supported using either API keys or OAuth 2.0 credentials. The project creation process and keys are discussed at `https://developers.google.com/youtube/registering_an_application#create_project`.

Searching by keyword

The process of searching for videos by keyword is adapted from `https://developers.google.com/youtube/v3/code_samples/java#search_by_keyword`. Other potentially useful code examples can be found at `https://developers.google.com/youtube/v3/code_samples/java`. The process has been simplified so that we can focus on the search process. We start with a try block and the creation of a `YouTube` instance. This class provides the basic access to the API. Javadocs for this API is found at `https://developers.google.com/resources/api-libraries/documentation/youtube/v3/java/latest/`.

The `YouTube.Builder` class is used to construct a `YouTube` instance. Its constructor takes three arguments:

- `Transport`: Object used for HTTP
- `JSONFactory`: Used to process JSON objects
- `HttpRequestInitializer`: None is needed for this example

Many of the APIs responses will be in the form of JSON objects. The YouTube class' `setApplicationName` method gives it a name and the `build` method creates a new YouTube instance:

```
try {
    YouTube youtube = new YouTube.Builder(
        Auth.HTTP_TRANSPORT,
        Auth.JSON_FACTORY,
        new HttpRequestInitializer() {
            public void initialize(HttpRequest request)
                    throws IOException {
            }
        })
            .setApplicationName("application_name")
    ...
```

```
        } catch (GoogleJSONResponseException ex) {
            // Handle exceptions
        } catch (IOException ex) {
            // Handle exceptions
        }
```

Next, we initialize a string to hold the search term of interest. In this case, we will look for videos containing the word `cats`:

```
        String queryTerm = "cats";
```

The class, `YouTube.Search.List`, maintains a collection of search results. The `YouTube` class's `search` method specifies the type of resource to be returned. In this case, the string specifies the `id` and `snippet` portions of the search result to be returned:

```
        YouTube.Search.List search = youtube
            .search()
            .list("id,snippet");
```

The search result is a JSON object that has the following structure. It is described in more detail at `https://developers.google.com/youtube/v3/docs/playlistItems#methods`. In the previous sequence, only the `id` and `snippet` parts of a search will be returned, resulting in a more efficient operation:

```
{
  "kind": "youtube#searchResult",
  "etag": etag,
  "id": {
    "kind": string,
    "videoId": string,
    "channelId": string,
    "playlistId": string
  },
  "snippet": {
    "publishedAt": datetime,
    "channelId": string,
    "title": string,
    "description": string,
    "thumbnails": {
      (key): {
        "url": string,
        "width": unsigned integer,
        "height": unsigned integer
      }
    },
    "channelTitle": string,
    "liveBroadcastContent": string
```

```
    }
}
```

Next, we need to specify the API key and various search parameters. The query term is specified, as well as the type of media to be returned. In this case, only videos will be returned. The other two options include `channel` and `playlist`:

```
String apiKey = "Your API key";
search.setKey(apiKey);
search.setQ(queryTerm);
search.setType("video");
```

In addition, we further specify the fields to be returned as shown here. These correspond to fields of the JSON object:

```
search.setFields("items(id/kind,id/videoId,snippet/title," +
    "snippet/description,snippet/thumbnails/default/url)");
```

We also specify the maximum number of results to retrieve using the `setMaxResults` method:

```
search.setMaxResults(10L);
```

The `execute` method will perform the actual query, returning a `SearchListResponse` object. Its `getItems` method returns a list of `SearchResult` objects, one for each video retrieved:

```
SearchListResponse searchResponse = search.execute();
List<SearchResult> searchResultList =
    searchResponse.getItems();
```

In this example, we do not iterate through each video returned. Instead, we retrieve the first video and display information about the video. The `SearchResult` video variable allows us to access different parts of the JSON object, as shown here:

```
SearchResult video = searchResultList.iterator().next();
Thumbnail thumbnail = video
    .getSnippet().getThumbnails().getDefault();

out.println("Kind: " + video.getKind());
out.println("Video Id: " + video.getId().getVideoId());
out.println("Title: " + video.getSnippet().getTitle());
out.println("Description: " +
    video.getSnippet().getDescription());
out.println("Thumbnail: " + thumbnail.getUrl());
```

One possible output follows where parts of the output have been modified:

```
Kind: null
Video Id: tntO...
Title: Funny Cats ...
Description: Check out the ...
Thumbnail: https://i.ytimg.com/vi/tntO.../default.jpg
```

We have skipped many error checking steps to simplify the example, but these should be considered when implementing this in a business application.

If we need to download the video, one of the simplest ways is to use axet/wget found at `https://github.com/axet/wget`. It provides an easy-to-use technique to download the video using its video ID.

In the following example, a URL is created using the video ID. You will need to provide a video ID for this to work properly. The file is saved to the current directory with the video's title as the filename:

```
String url = "http://www.youtube.com/watch?v=videoID";
String path = ".";
VGet vget = new VGet(new URL(url), new File(path));
vget.download();
```

There are other more sophisticated download techniques found at the GitHub site.

Summary

In this chapter, we discussed types of data that are useful for data science and readily accessible on the Internet. This discussion included details about file specifications and formats for the most common types of data sources.

We also examined Java APIs and other techniques for retrieving data, and illustrated this process with multiple sources. In particular, we focused on types of text-based document formats and multimedia files. We used web crawlers to access websites and then performed web scraping to retrieve data from the sites we encountered.

Finally, we extracted data from social media sites and examined the available Java support. We retrieved data from Twitter, Wikipedia, Flickr, and YouTube and examined the available API support.

3
Data Cleaning

Real-world data is frequently dirty and unstructured, and must be reworked before it is usable. Data may contain errors, have duplicate entries, exist in the wrong format, or be inconsistent. The process of addressing these types of issues is called **data cleaning**. Data cleaning is also referred to as **data wrangling**, **massaging**, **reshaping** , or **munging**. Data merging, where data from multiple sources is combined, is often considered to be a data cleaning activity.

We need to clean data because any analysis based on inaccurate data can produce misleading results. We want to ensure that the data we work with is quality data. Data quality involves:

- **Validity**: Ensuring that the data possesses the correct form or structure
- **Accuracy:** The values within the data are truly representative of the dataset
- **Completeness:** There are no missing elements
- **Consistency**: Changes to data are in sync
- **Uniformity**: The same units of measurement are used

There are several techniques and tools used to clean data. We will examine the following approaches:

- Handling different types of data
- Cleaning and manipulating text data
- Filling in missing data
- Validating data

In addition, we will briefly examine several image enhancement techniques.

There are often many ways to accomplish the same cleaning task. For example, there are a number of GUI tools that support data cleaning, such as OpenRefine (`http://openrefine.org/`). This tool allows a user to read in a dataset and clean it using a variety of techniques. However, it requires a user to interact with the application for each dataset that needs to be cleaned. It is not conducive to automation.

We will focus on how to clean data using Java code. Even then, there may be different techniques to clean the data. We will show multiple approaches to provide the reader with insights on how it can be done. Sometimes, this will use core Java string classes, and at other time, it may use specialized libraries.

These libraries often are more expressive and efficient. However, there are times when using a simple string function is more than adequate to address the problem. Showing complimentary techniques will improve the reader's skill set.

The basic text based tasks include:

- Data transformation
- Data imputation (handling missing data)
- Subsetting data
- Sorting data
- Validating data

In this chapter, we are interested in cleaning data. However, part of this process is extracting information from various data sources. The data may be stored in plaintext or in binary form. We need to understand the various formats used to store data before we can begin the cleaning process. Many of these formats were introduced in `Chapter 2`, *Data Acquisition*, but we will go into greater detail in the following sections.

Handling data formats

Data comes in all types of forms. We will examine the more commonly used formats and show how they can be extracted from various data sources. Before we can clean data it needs to be extracted from a data source such as a file. In this section, we will build upon the introduction to data formats found in `Chapter 2`, *Data Acquisition*, and show how to extract all or part of a dataset. For example, from an HTML page we may want to extract only the text without markup. Or perhaps we are only interested in its figures.

These data formats can be quite complex. The intent of this section is to illustrate the basic techniques commonly used with that data format. Full treatment of a specific data format is beyond the scope of this book. Specifically, we will introduce how the following data formats can be processed from Java:

- CSV data
- Spreadsheets
- Portable Document Format, or PDF files
- JavaScript Object Notation, or JSON files

There are many other file types not addressed here. For example, jsoup is useful for parsing HTML documents. Since we introduced how this is done in the Web scraping in Java section of `Chapter 2`, *Data Acquisition*, we will not duplicate the effort here.

Handling CSV data

A common technique for separating information is to use commas or similar separators. Knowing how to work with CSV data allows us to utilize this type of data in our analysis efforts. When we deal with CSV data there are several issues including escaped data and embedded commas.

We will examine a few basic techniques for processing comma-separated data. Due to the row-column structure of CSV data, these techniques will read data from a file and place the data in a two-dimensional array. First, we will use a combination of the `Scanner` class to read in tokens and the `String` class `split` method to separate the data and store it in the array. Next, we will explore using the third-party library, OpenCSV, which offers a more efficient technique.

However, the first approach may only be appropriate for quick and dirty processing of data. We will discuss each of these techniques since they are useful in different situations.

We will use a dataset downloaded from `https://www.data.gov/` containing U.S. demographic statistics sorted by ZIP code. This dataset can be downloaded at `https://catalog.data.gov/dataset/demographic-statistics-by-zip-code-acfc9`. For our purposes, this dataset has been stored in the file `Demographics.csv`. In this particular file, every row contains the same number of columns. However, not all data will be this clean and the solutions shown next take into account the possibility for jagged arrays.



A jagged array is an array where the number of columns may be different for different rows. For example, row 2 may have 5 elements while row 3 may have 6 elements. When using jagged arrays you have to be careful with your column indexes.

First, we use the `Scanner` class to read in data from our data file. We will temporarily store the data in an `ArrayList` since we will not always know how many rows our data contains.

```
try (Scanner csvData = new Scanner(new File("Demographics.csv"))) {
    ArrayList<String> list = new ArrayList<String>();
    while (csvData.hasNext()) {
        list.add(csvData.nextLine());
} catch (FileNotFoundException ex) {
    // Handle exceptions
}
```

The list is converted to an array using the `toArray` method. This version of the method uses a `String` array as an argument so that the method will know what type of array to create. A two-dimension array is then created to hold the CSV data.

```
String[] tempArray = list.toArray(new String[1]);
String[][] csvArray = new String[tempArray.length][];
```

The `split` method is used to create an array of `String`s for each row. This array is assigned to a row of the `csvArray`.

```
for(int i=0; i<tempArray.length; i++) {
    csvArray[i] = tempArray[i].split(",");
}
```

Our next technique will use a third-party library to read in and process CSV data. There are multiple options available, but we will focus on the popular OpenCSV (`http://opencsv.sourceforge.net`). This library offers several advantages over our previous technique. We can have an arbitrary number of items on each row without worrying about handling exceptions. We also do not need to worry about embedded commas or embedded carriage returns within the data tokens. The library also allows us to choose between reading the entire file at once or using an iterator to process data line-by-line.

First, we need to create an instance of the `CSVReader` class. Notice the second parameter allows us to specify the delimiter, a useful feature if we have similar file format delimited by tabs or dashes, for example. If we want to read the entire file at one time, we use the `readAll` method.

```
CSVReader dataReader = new CSVReader(new
FileReader("Demographics.csv"),',');
ArrayList<String> holdData = (ArrayList)dataReader.readAll();
```

We can then process the data as we did above, by splitting the data into a two-dimension array using `String` class methods. Alternatively, we can process the data one line at a time. In the example that follows, each token is printed out individually but the tokens can also be stored in a two-dimension array or other data structure as appropriate.

```
CSVReader dataReader = new CSVReader(new
FileReader("Demographics.csv"),',');
String[] nextLine;
while ((nextLine = dataReader.readNext()) != null){
for(String token : nextLine){
    out.println(token);
  }
}
dataReader.close();
```

We can now clean or otherwise process the array.

Handling spreadsheets

Spreadsheets have proven to be a very popular tool for processing numeric and textual data. Due to the wealth of information that has been stored in spreadsheets over the past decades, knowing how to extract information from spreadsheets enables us to take advantage of this widely available data source. In this section, we will demonstrate how this is accomplished using the Apache POI API.

Open Office also supports a spreadsheet application. Open Office documents are stored in XML format which makes it readily accessible using XML parsing technologies. However, the Apache ODF Toolkit (`http://incubator.apache.org/odftoolkit/`) provides a means of accessing data within a document without knowing the format of the OpenOffice document. This is currently an incubator project and is not fully mature. There are a number of other APIs that can assist in processing OpenOffice documents as detailed on the **Open Document Format** (**ODF**) for developers (`http://www.opendocumentformat.org/developers/`) page.

Handling Excel spreadsheets

Apache POI (`http://poi.apache.org/index.html`) is a set of APIs providing access to many Microsoft products including Excel and Word. It consists of a series of components designed to access a specific Microsoft product. An overview of these components is found at `http://poi.apache.org/overview.html`.

In this section we will demonstrate how to read a simple Excel spreadsheet using the XSSF component to access Excel 2007+ spreadsheets. The Javadocs for the Apache POI API is found at `https://poi.apache.org/apidocs/index.html`.

We will use a simple Excel spreadsheet consisting of a series of rows containing an ID along with minimum, maximum, and average values. These numbers are not intended to represent any specific type of data. The spreadsheet follows:

ID	Minimum	Maximum	Average
12345	45	89	65.55
23456	78	96	86.75
34567	56	89	67.44
45678	86	99	95.67

We start with a try-with-resources block to handle any `IOExceptions` that may occur:

```
try (FileInputStream file = new FileInputStream(
        new File("Sample.xlsx"))) {
    ...
    }
} catch (IOException e) {
    // Handle exceptions
}
```

An instance of a `XSSFWorkbook` class is created using the spreadsheet. Since a workbook may consists of multiple spreadsheets, we select the first one using the `getSheetAt` method.

```
XSSFWorkbook workbook = new XSSFWorkbook(file);
XSSFSheet sheet = workbook.getSheetAt(0);
```

The next step is to iterate through the rows, and then each column, of the spreadsheet:

```
for(Row row : sheet) {
    for (Cell cell : row) {
        ...
    }
out.println();
```

Each cell of the spreadsheet may use a different format. We use the `getCellType` method to determine its type and then use the appropriate method to extract the data in the cell. In this example we are only working with numeric and text data.

```
switch (cell.getCellType()) {
    case Cell.CELL_TYPE_NUMERIC:
        out.print(cell.getNumericCellValue() + "\t");
        break;
    case Cell.CELL_TYPE_STRING:
        out.print(cell.getStringCellValue() + "\t");
        break;
   }
```

When executed we get the following output:

```
ID Minimum Maximum Average
12345.0 45.0 89.0 65.55
23456.0 78.0 96.0 86.75
34567.0 56.0 89.0 67.44
45678.0 86.0 99.0 95.67
```

POI supports other more sophisticated classes and methods to extract data.

Handling PDF files

There are several APIs supporting the extraction of text from a PDF file. Here we will use PDFBox. The Apache PDFBox (`https://pdfbox.apache.org/`) is an open source API that allows Java programmers to work with PDF documents. In this section we will illustrate how to extract simple text from a PDF document. Javadocs for the PDFBox API is found at `https://pdfbox.apache.org/docs/2.0.1/javadocs/`.

This is a simple PDF file. It consists of several bullets:

- Line 1
- Line 2
- Line 3

This is the end of the document.

A `try` block is used to catch `IOExceptions`. The `PDDocument` class will represent the PDF document being processed. Its `load` method will load in the PDF file specified by the `File` object:

```
try {
    PDDocument document = PDDocument.load(new File("PDF File.pdf"));
    ...
} catch (Exception e) {
    // Handle exceptions
}
```

Once loaded, the `PDFTextStripper` class `getText` method will extract the text from the file. The text is then displayed as shown here:

```
PDFTextStripper Tstripper = new PDFTextStripper();
String documentText = Tstripper.getText(document);
System.out.println(documentText);
```

The output of this example follows. Notice that the bullets are returned as question marks.

```
This is a simple PDF file. It consists of several bullets:
? Line 1
? Line 2
? Line 3
This is the end of the document.
```

This is a brief introduction to the use of PDFBox. It is a very powerful tool when we need to extract and otherwise manipulate PDF documents.

Handling JSON

In `Chapter 2`, *Data Acquisition* we learned that certain YouTube searches return JSON formatted results. Specifically, the `SearchResult` class holds information relating to a specific search. In that section we illustrate how to use YouTube specific techniques to extract information. In this section we will illustrate how to extract JSON information using the Jackson JSON implementation.

JSON supports three models for processing data:

- **Streaming API** - JSON data is processed token by token
- **Tree model** - The JSON data is held entirely in memory and then processed
- **Data binding** - The JSON data is transformed to a Java object

Using JSON streaming API

We will illustrate the first two approaches. The first approach is more efficient and is used when a large amount of data is processed. The second technique is convenient but the data must not be too large. The third technique is useful when it is more convenient to use specific Java classes to process data. For example, if the JSON data represent an address then a specific Java address class cane be defined to hold and process the data.

There are several Java libraries that support JSON processing including:

- Flexjson (`http://flexjson.sourceforge.net/`)
- Genson (`http://owlike.github.io/genson/`)
- Google-Gson (`https://github.com/google/gson`)
- Jackson library (`https://github.com/FasterXML/jackson`)
- JSON-io (`https://github.com/jdereg/json-io`)
- JSON-lib (`http://json-lib.sourceforge.net/`)

We will use the Jackson Project (`https://github.com/FasterXML/jackson`). Documentation is found at `https://github.com/FasterXML/jackson-docs`. We will use two JSON files to demonstrate how it can be used. The first file, `Person.json`, is shown next where a single person data is stored. It consists of four fields where the last field is an array of location information.

```
{
   "firstname":"Smith",
   "lastname":"Peter",
   "phone":8475552222,
   "address":["100 Main Street","Corpus","Oklahoma"]
}
```

The code sequence that follows shows how to extract the values for each of the fields. Within the try-catch block a `JsonFactory` instance is created which then creates a `JsonParser` instance based on the `Person.json` file.

```
try {
    JsonFactory jsonfactory = new JsonFactory();
    JsonParser parser = jsonfactory.createParser(new File("Person.json"));
    ...
    parser.close();
} catch (IOException ex) {
    // Handle exceptions
}
```

The `nextToken` method returns a `token`. However, the `JsonParser` object keeps track of the current token. In the `while` loop the `nextToken` method returns and advances the parser to the next token. The `getCurrentName` method returns the field name for the `token`. The `while` loop terminates when the last token is reached.

```
while (parser.nextToken() != JsonToken.END_OBJECT) {
    String token = parser.getCurrentName();
    ...
}
```

The body of the loop consists of a series of `if` statements that processes the field by its name. Since the `address` field is an array, another loop will extract each of its elements until the ending array `token` is reached.

```
if ("firstname".equals(token)) {
    parser.nextToken();
    String fname = parser.getText();
    out.println("firstname : " + fname);
}
if ("lastname".equals(token)) {
```

```
        parser.nextToken();
        String lname = parser.getText();
        out.println("lastname : " + lname);
    }
    if ("phone".equals(token)) {
        parser.nextToken();
        long phone = parser.getLongValue();
        out.println("phone : " + phone);
    }
    if ("address".equals(token)) {
        out.println("address :");
        parser.nextToken();
        while (parser.nextToken() != JsonToken.END_ARRAY) {
            out.println(parser.getText());
        }
    }
```

The output of this example follows:

```
firstname : Smith
lastname : Peter
phone : 8475552222
address :
100 Main Street
Corpus
Oklahoma
```

However, JSON objects are frequently more complex than the previous example. Here a `Persons.json` file consists of an array of three `persons`:

```
{
   "persons": {
      "groupname": "school",
      "person":
         [
            {"firstname":"Smith",
              "lastname":"Peter",
              "phone":8475552222,
              "address":["100 Main Street","Corpus","Oklahoma"] },
           {"firstname":"King",
              "lastname":"Sarah",
              "phone":8475551111,
              "address":["200 Main Street","Corpus","Oklahoma"] },
           {"firstname":"Frost",
              "lastname":"Nathan",
              "phone":8475553333,
              "address":["300 Main Street","Corpus","Oklahoma"] }
         ]
```

```
    }
}
```

To process this file, we use a similar set of code as shown previously. We create the parser and then enter a loop as before:

```
try {
    JsonFactory jsonfactory = new JsonFactory();
    JsonParser parser = jsonfactory.createParser(new File("Person.json"));
    while (parser.nextToken() != JsonToken.END_OBJECT) {
        String token = parser.getCurrentName();
        ...
    }
    parser.close();
} catch (IOException ex) {
    // Handle exceptions
}
```

However, we need to find the `persons` field and then extract each of its elements. The `groupname` field is extracted and displayed as shown here:

```
if ("persons".equals(token)) {
    JsonToken jsonToken = parser.nextToken();
    jsonToken = parser.nextToken();
    token = parser.getCurrentName();
    if ("groupname".equals(token)) {
        parser.nextToken();
        String groupname = parser.getText();
        out.println("Group : " + groupname);
        ...
    }
}
```

Next, we find the `person` field and call a `parsePerson` method to better organize the code:

```
parser.nextToken();
token = parser.getCurrentName();
if ("person".equals(token)) {
    out.println("Found person");
    parsePerson(parser);
}
```

The `parsePerson` method follows which is very similar to the process used in the first example.

```
public void parsePerson(JsonParser parser) throws IOException {
    while (parser.nextToken() != JsonToken.END_ARRAY) {
        String token = parser.getCurrentName();
```

```
            if ("firstname".equals(token)) {
                parser.nextToken();
                String fname = parser.getText();
                out.println("firstname : " + fname);
            }
            if ("lastname".equals(token)) {
                parser.nextToken();
                String lname = parser.getText();
                out.println("lastname : " + lname);
            }
            if ("phone".equals(token)) {
                parser.nextToken();
                long phone = parser.getLongValue();
                out.println("phone : " + phone);
            }
            if ("address".equals(token)) {
                out.println("address :");
                parser.nextToken();
                while (parser.nextToken() != JsonToken.END_ARRAY) {
                    out.println(parser.getText());
                }
            }
        }
    }
```

The output follows:

```
Group : school
Found person
firstname : Smith
lastname : Peter
phone : 8475552222
address :
100 Main Street
Corpus
Oklahoma
firstname : King
lastname : Sarah
phone : 8475551111
address :
200 Main Street
Corpus
Oklahoma
firstname : Frost
lastname : Nathan
phone : 8475553333address :
300 Main Street
```

```
Corpus
Oklahoma
```

Using the JSON tree API

The second approach is to use the tree model. An `ObjectMapper` instance is used to create a `JsonNode` instance using the `Persons.json` file. The `fieldNames` method returns `Iterator` allowing us to process each element of the file.

```
try {
    ObjectMapper mapper = new ObjectMapper();
    JsonNode node = mapper.readTree(new File("Persons.json"));
    Iterator<String> fieldNames = node.fieldNames();
    while (fieldNames.hasNext()) {
        ...
        fieldNames.next();
    }
} catch (IOException ex) {
    // Handle exceptions
}
```

Since the JSON file contains a `persons` field, we will obtain a `JsonNode` instance representing the field and then iterate over each of its elements.

```
JsonNode personsNode = node.get("persons");
Iterator<JsonNode> elements = personsNode.iterator();
while (elements.hasNext()) {
    ...
}
```

Each element is processed one at a time. If the element type is a string, we assume that this is the `groupname` field.

```
JsonNode element = elements.next();
JsonNodeType nodeType = element.getNodeType();

if (nodeType == JsonNodeType.STRING) {
    out.println("Group: " + element.textValue());
}
```

If the element is an array, we assume it contains a series of persons where each person is processed by the `parsePerson` method:

```
if (nodeType == JsonNodeType.ARRAY) {
    Iterator<JsonNode> fields = element.iterator();
    while (fields.hasNext()) {
```

```
            parsePerson(fields.next());
        }
    }
```

The `parsePerson` method is shown next:

```
    public void parsePerson(JsonNode node) {
        Iterator<JsonNode> fields = node.iterator();
        while(fields.hasNext()) {
            JsonNode subNode = fields.next();
            out.println(subNode.asText());
        }
    }
```

The output follows:

```
Group: school
Smith
Peter
8475552222
King
Sarah
8475551111
Frost
Nathan
8475553333
```

There is much more to JSON than we are able to illustrate here. However, this should give you an idea of how this type of data can be handled.

The nitty gritty of cleaning text

Strings are used to support text processing so using a good string library is important. Unfortunately, the `java.lang.String` class has some limitations. To address these limitations, you can either implement your own special string functions as needed or you can use a third-party library.

Creating your own library can be useful, but you will basically be reinventing the wheel. It may be faster to write a simple code sequence to implement some functionality, but to do things right, you will need to test them. Third-party libraries have already been tested and have been used on hundreds of projects. They provide a more efficient way of processing text.

There are several text processing APIs in addition to those found in Java. We will demonstrate two of these:

- **Apache Commons**: `https://commons.apache.org/`
- **Guava**: `https://github.com/google/guava`

Java provides many supports for cleaning text data, including methods in the `String` class. These methods are ideal for simple text cleaning and small amounts of data but can also be efficient with larger, complex datasets. We will demonstrate several `String` class methods in a moment. Some of the most helpful `String` class methods are summarized in the following table:

Method Name	Return Type	Description
`trim`	`String`	Removes leading and trailing blank spaces
`toUpperCase/toLowerCase`	`String`	Changes the casing of the entire string
`replaceAll`	`String`	Replaces all occurrences of a character sequence within the string
`contains`	`boolean`	Determines whether a given character sequence exists within the string
`compareTo` `compareToIgnoreCase`	`int`	Compares two strings lexographically and returns an integer representing their relationship
`matches`	`boolean`	Determines whether the string matches a given regular expression
`join`	`String`	Combines two or more strings with a specified delimiter
`split`	`String[]`	Separates elements of a given string using a specified delimiter

Many text operations are simplified by the use of regular expressions. Regular expressions use standardized syntax to represent patterns in text, which can be used to locate and manipulate text matching the pattern.

A regular expression is simply a string itself. For example, the string `Hello, my name is Sally` can be used as a regular expression to find those exact words within a given text. This is very specific and not broadly applicable, but we can use a different regular expression to make our code more effective. `Hello, my name is \\w` will match any text that starts with `Hello, my name is` and ends with a word character.

We will use several examples of more complex regular expressions, and some of the more useful syntax options are summarized in the following table. Note each must be double-escaped when used in a Java application.

Option	**Description**
`\d`	Any digit: *0-9*
`\D`	Any non-digit
`\s`	Any whitespace character
`\S`	Any non-whitespace character
`\w`	Any word character (including digits): *A-Z*, *a-z*, and *0-9*
`\W`	Any non-word character

The size and source of text data varies wildly from application to application but the methods used to transform the data remain the same. You may actually need to read data from a file, but for simplicity's sake, we will be using a string containing the beginning sentences of Herman Melville's Moby Dick for several examples within this chapter. Unless otherwise specified, the text will assumed to be as shown next:

```
String dirtyText = "Call me Ishmael. Some years ago– never mind how";
dirtyText += " long precisely – having little or no money in my purse,";
dirtyText += " and nothing particular to interest me on shore, I thought";
dirtyText += " I would sail about a little and see the watery part of the
world.";
```

Using Java tokenizers to extract words

Often it is most efficient to analyze text data as tokens. There are multiple tokenizers available in the core Java libraries as well as third-party tokenizers. We will demonstrate various tokenizers throughout this chapter. The ideal tokenizer will depend upon the limitations and requirements of an individual application.

Java core tokenizers

`StringTokenizer` was the first and most basic tokenizer and has been available since Java 1. It is not recommended for use in new development as the `String` class's `split` method is considered more efficient. While it does provide a speed advantage for files with narrowly defined and set delimiters, it is less flexible than other tokenizer options. The following is a simple implementation of the `StringTokenizer` class that splits a string on spaces:

```
StringTokenizer tokenizer = new StringTokenizer(dirtyText," ");
while(tokenizer.hasMoreTokens()){
  out.print(tokenizer.nextToken() + " ");
}
```

When we set the `dirtyText` variable to hold our text from Moby Dick, shown previously, we get the following truncated output:

```
Call me Ishmael. Some years ago– never mind how long precisely...
```

`StreamTokenizer` is another core Java tokenizer. `StreamTokenizer` grants more information about the tokens retrieved, and allows the user to specify data types to parse, but is considered more difficult to use than `StreamTokenizer` or the `split` method. The `String` class `split` method is the simplest way to split strings up based on a delimiter, but it does not provide a way to parse the split strings and you can only specify one delimiter for the entire string. For these reasons, it is not a true tokenizer, but it can be useful for data cleaning.

The `Scanner` class is designed to allow you to parse strings into different data types. We used it previously in the *Handling CSV data* section and we will address it again in the *Removing stop words* section.

Third-party tokenizers and libraries

Apache Commons consists of sets of open source Java classes and methods. These provide reusable code that complements the standard Java APIs. One popular class included in the Commons is `StrTokenizer`. This class provides more advanced support than the standard `StringTokenizer` class, specifically more control and flexibility. The following is a simple implementation of the `StrTokenizer`:

```
StrTokenizer tokenizer = new StrTokenizer(text);
while (tokenizer.hasNext()) {
  out.print(tokenizer.next() + " ");
}
```

This operates in a similar fashion to `StringTokenizer` and by default parses tokens on spaces. The constructor can specify the delimiter as well as how to handle double quotes contained in data.

When we use the string from Moby Dick, shown previously, the first tokenizer implementation produces the following truncated output:

```
Call me Ishmael. Some years ago– never mind how long precisely – having
little or no money in my purse...
```

We can modify our constructor as follows:

```
StrTokenizer tokenizer = new StrTokenizer(text,",");
```

The output for this implementation is:

```
Call me Ishmael. Some years ago– never mind how long precisely – having
little or no money in my purse
and nothing particular to interest me on shore
I thought I would sail about a little and see the watery part of the world.
```

Notice how each line is split where commas existed in the original text. This delimiter can be a simple char, as we have shown, or a more complex `StrMatcher` object.

Google Guava is an open source set of utility Java classes and methods. The primary goal of Guava, as with many APIs, is to relieve the burden of writing basic Java utilities so developers can focus on business processes. We are going to talk about two main tools in Guava in this chapter: the `Joiner` class and the `Splitter` class. Tokenization is accomplished in Guava using its `Splitter` class's `split` method. The following is a simple example:

```
Splitter simpleSplit = Splitter.on(',').omitEmptyStrings().trimResults();
Iterable<String> words = simpleSplit.split(dirtyText);
for(String token: words){
  out.print(token);
}
```

This splits each token on commas and produces output like our last example. We can modify the parameter of the `on` method to split on the character of our choosing. Notice the method chaining which allows us to omit empty strings and trim leading and trailing spaces. For these reasons, and other advanced capabilities, Google Guava is considered by some to be the best tokenizer available for Java.

LingPipe is a linguistical toolkit available for language processing in Java. It provides more specialized support for text splitting with its `TokenizerFactory` interface. We implement a LingPipe `IndoEuropeanTokenizerFactory` tokenizer in the *Simple text cleaning* section.

Transforming data into a usable form

Data often needs to be cleaned once it has been acquired. Datasets are often inconsistent, are missing in information, and contain extraneous information. In this section, we will examine some simple ways to transform text data to make it more useful and easier to analyze.

Simple text cleaning

We will use the string shown before from Moby Dick to demonstrate some of the basic `String` class methods. Notice the use of the `toLowerCase` and `trim` methods. Datasets often have non-standard casing and extra leading or trailing spaces. These methods ensure uniformity of our dataset. We also use the `replaceAll` method twice. In the first instance, we use a regular expression to replace all numbers and anything that is not a word or whitespace character with a single space. The second instance replaces all back-to-back whitespace characters with a single space:

```
out.println(dirtyText);
dirtyText =    dirtyText.toLowerCase().replaceAll("[\\d[^\\w\\s]]+", " ");
dirtyText = dirtyText.trim();
while(dirtyText.contains("  ")){
  dirtyText = dirtyText.replaceAll("  ", " ");
}
out.println(dirtyText);
```

When executed, the code produces the following output, truncated:

```
Call me Ishmael. Some years ago– never mind how long precisely –
call me ishmael some years ago never mind how long precisely
```

Our next example produces the same result but approaches the problem with regular expressions. In this case, we replace all of the numbers and other special characters first. Then we use method chaining to standardize our casing, remove leading and trailing spaces, and split our words into a `String` array. The `split` method allows you to break apart text on a given delimiter. In this case, we chose to use the regular expression `\\W`, which represents anything that is not a word character:

```
out.println(dirtyText);
dirtyText = dirtyText.replaceAll("[\\d[^\\w\\s]]+", "");
String[] cleanText = dirtyText.toLowerCase().trim().split("[\\W]+");
for(String clean : cleanText){
  out.print(clean + " ");
}
```

This code produces the same output as shown previously.

Although arrays are useful for many applications, it is often important to recombine text after cleaning. In the next example, we employ the `join` method to combine our words once we have cleaned them. We use the same chained methods as shown previously to clean and split our text. The `join` method joins every word in the array `words` and inserts a space between each word:

```
out.println(dirtyText);
String[] words =    dirtyText.toLowerCase().trim().split("[\\W\\d]+");
String cleanText = String.join(" ", words);
out.println(cleanText);
```

Again, this code produces the same output as shown previously. An alternate version of the `join` method is available using Google Guava. Here is a simple implementation of the same process we used before, but using the Guava `Joiner` class:

```
out.println(dirtyText);
String[] words =    dirtyText.toLowerCase().trim().split("[\\W\\d]+");
String cleanText = Joiner.on(" ").skipNulls().join(words);
out.println(cleanText);
```

This version provides additional options, including skipping nulls, as shown before. The output remains the same.

Removing stop words

Text analysis sometimes requires the omission of common, non-specific words such as *the*, *and*, or *but*. These words are known as stop words are there are several tools available for removing them from text. There are various ways to store a list of stop words, but for the following examples, we will assume they are contained in a file. To begin, we create a new `Scanner` object to read in our stop words. Then we take the text we wish to transform and store it in an `ArrayList` using the `Arrays` class's `asList` method. We will assume here the text has already been cleaned and normalized. It is essential to consider casing when using `String` class methods---*and* is not the same as *AND* or *And*, although all three may be stop words you wish to eliminate:

```
Scanner readStop = new Scanner(new File("C://stopwords.txt"));
ArrayList<String> words = new
ArrayList<String>(Arrays.asList((dirtyText));
out.println("Original clean text: " + words.toString());
```

We also create a new `ArrayList` to hold a list of stop words actually found in our text. This will allow us to use the `ArrayList` class `removeAll` method shortly. Next, we use our `Scanner` to read through our file of stop words. Notice how we also call the `toLowerCase` and `trim` methods against each stop word. This is to ensure that our stop words match the formatting in our text. In this example, we employ the `contains` method to determine whether our text contains the given stop word. If so, we add it to our `foundWords` ArrayList. Once we have processed all the stop words, we call `removeAll` to remove them from our text:

```
ArrayList<String> foundWords = new ArrayList();
while(readStop.hasNextLine()){
  String stopWord = readStop.nextLine().toLowerCase();
  if(words.contains(stopWord)){
    foundWords.add(stopWord);
  }
}
words.removeAll(foundWords);
out.println("Text without stop words: " + words.toString());
```

The output will depend upon the words designated as stop words. If your stop words file contains different words than used in this example, your output will differ slightly. Our output follows:

```
Original clean text: [call, me, ishmael, some, years, ago, never, mind,
how, long, precisely, having, little, or, no, money, in, my, purse, and,
nothing, particular, to, interest, me, on, shore, i, thought, i, would,
sail, about, a, little, and, see, the, watery, part, of, the, world]
Text without stop words: [call, ishmael, years, ago, never, mind, how,
long, precisely
```

There is also support outside of the standard Java libraries for removing stop words. We are going to look at one example, using LingPipe. In this example, we start by ensuring that our text is normalized in lowercase and trimmed. Then we create a new instance of the `TokenizerFactory` class. We set our factory to use default English stop words and then tokenize the text. Notice that the `tokenizer` method uses a `char` array, so we call `toCharArray` against our text. The second parameter specifies where to begin searching within the text, and the last parameter specifies where to end:

```
text = text.toLowerCase().trim();
TokenizerFactory fact = IndoEuropeanTokenizerFactory.INSTANCE;
fact = new EnglishStopTokenizerFactory(fact);
Tokenizer tok = fact.tokenizer(text.toCharArray(), 0, text.length());
for(String word : tok){
  out.print(word + " ");
}
```

The output follows:

```
Call me Ishmael. Some years ago– never mind how long precisely – having
little or no money in my purse, and nothing particular to interest me on
shore, I thought I would sail about a little and see the watery part of the
world.
call me ishmael . years ago – never mind how long precisely – having little
money my purse , nothing particular interest me shore , i thought i sail
little see watery part world .
```

Notice the differences between our previous examples. First of all, we did not clean the text as thoroughly and allowed special characters, such as the hyphen, to remain in the text. Secondly, the LingPipe list of stop words differs from the file we used in the previous example. Some words are removed, but LingPipe was less restrictive and allowed more words to remain in the text. The type and number of stop words you use will depend upon your particular application.

Finding words in text

The standard Java libraries offer support for searching through text for specific tokens. In previous examples, we have demonstrated the `matches` method and regular expressions, which can be useful when searching text. In this example, however, we will demonstrate a simple technique using the `contains` method and the `equals` method to locate a particular string. First, we normalize our text and the word we are searching for to ensure we can find a match. We also create an integer variable to hold the number of times the word is found:

```
dirtyText = dirtyText.toLowerCase().trim();
toFind = toFind.toLowerCase().trim();
int count = 0;
```

Next, we call the `contains` method to determine whether the word exists in our text. If it does, we split the text into an array and then loop through, using the `equals` method to compare each word. If we encounter the word, we increment our counter by one. Finally, we display the output to show how many times our word was encountered:

```
if(dirtyText.contains(toFind)){
      String[] words = dirtyText.split(" ");
      for(String word : words){
            if(word.equals(toFind)){
                  count++;
            }
      }
out.println("Found " + toFind + " " + count + " times in the text.");
}
```

In this example, we set `toFind` to the letter `I`. This produced the following output:

```
Found i 2 times in the text.
```

We also have the option to use the `Scanner` class to search through an entire file. One helpful method is the `findWithinHorizon` method. This uses a `Scanner` to parse the text up to a given horizon specification. If zero is used for the second parameter, as shown next, the entire `Scanner` will be searched by default:

```
dirtyText = dirtyText.toLowerCase().trim();
toFind = toFind.toLowerCase().trim();
Scanner textLine = new Scanner(dirtyText);
out.println("Found " + textLine.findWithinHorizon(toFind, 10));
```

This technique can be more efficient for locating a particular string, but it does make it more difficult to determine where, and how many times, the string was found.

It can also be more efficient to search an entire file using a `BufferedReader`. We specify the file to search and use a try-catch block to catch any IO exceptions. We create a new `BufferedReader` object from our path and process our file as long as the next line is not empty:

```
String path = "C:// MobyDick.txt";
try {
    String textLine = "";
    toFind = toFind.toLowerCase().trim();
    BufferedReader textToClean = new BufferedReader(
        new FileReader(path));
    while((textLine = textToClean.readLine()) != null){
        line++;
        if(textLine.toLowerCase().trim().contains(toFind)){
            out.println("Found " + toFind + " in " + textLine);
           }
    }
    textToClean.close();
} catch (IOException ex) {
    // Handle exceptions
}
```

We again test our data by searching for the word `I` in the first sentences of Moby Dick. The truncated output follows:

```
Found i in Call me Ishmael...
```

Finding and replacing text

We often not only want to find text but also replace it with something else. We begin our next example much like we did the previous examples, by specifying our text, our text to locate, and invoking the `contains` method. If we find the text, we call the `replaceAll` method to modify our string:

```
text = text.toLowerCase().trim();
toFind = toFind.toLowerCase().trim();
out.println(text);

if(text.contains(toFind)){
      text = text.replaceAll(toFind, replaceWith);
      out.println(text);
}
```

To test this code, we set `toFind` to the word `I` and `replaceWith` to `Ishmael`. Our output follows:

```
call me ishmael. some years ago- never mind how long precisely - having
little or no money in my purse, and nothing particular to interest me on
shore, i thought i would sail about a little and see the watery part of the
world.
call me ishmael. some years ago- never mind how long precisely - having
little or no money in my purse, and nothing particular to interest me on
shore, Ishmael thought Ishmael would sail about a little and see the watery
part of the world.
```

Apache Commons also provides a `replace` method with several variations in the `StringUtils` class. This class provides much of the same functionality as the `String` class, but with more flexibility and options. In the following example, we use our string from Moby Dick and replace all instances of the word `me` with `X` to demonstrate the `replace` method:

```
out.println(text);
out.println(StringUtils.replace(text, "me", "X"));
```

The truncated output follows:

```
Call me Ishmael. Some years ago- never mind how long precisely -
Call X Ishmael. SoX years ago- never mind how long precisely -
```

Notice how every instance of `me` has been replaced, even those instances contained within other words, such as `some`. This can be avoided by adding spaces around `me`, although this will ignore any instances where me is at the end of the sentence, like me. We will examine a better alternative using Google Guava in a moment.

The `StringUtils` class also provides a `replacePattern` method that allows you to search for and replace text based upon a regular expression. In the following example, we replace all non-word characters, such as hyphens and commas, with a single space:

```
out.println(text);
text = StringUtils.replacePattern(text, "\\W\\s", " ");
out.println(text);
```

This will produce the following truncated output:

```
Call me Ishmael. Some years ago– never mind how long precisely –
Call me Ishmael Some years ago never mind how long precisely
```

Google Guava provides additional support for matching and modify text data using the `CharMatcher` class. `CharMatcher` not only allows you to find data matching a particular char pattern, but also provides options as to how to handle the data. This includes allowing you to retain the data, replace the data, and trim whitespaces from within a particular string.

In this example, we are going to use the `replace` method to simply replace all instances of the word `me` with a single space. This will produce series of empty spaces within our text. We will then collapse the extra whitespace using the `trimAndCollapseFrom` method and print our string again:

```
text = text.replace("me", " ");
out.println("With double spaces: " + text);
String spaced = CharMatcher.WHITESPACE.trimAndCollapseFrom(text, ' ');
out.println("With double spaces removed: " + spaced);
```

Our output is truncated as follows:

```
With double spaces: Call Ishmael. So years ago– ...
With double spaces removed: Call Ishmael. So years ago– ...
```

Data imputation

Data imputation refers to the process of identifying and replacing missing data in a given dataset. In almost any substantial case of data analysis, missing data will be an issue, and it needs to be addressed before data can be properly analysed. Trying to process data that is missing information is a lot like trying to understand a conversation where every once in while a word is dropped. Sometimes we can understand what is intended. In other situations, we may be completely lost as to what is trying to be conveyed.

Among statistical analysts, there exist differences of opinion as to how missing data should be handled but the most common approaches involve replacing missing data with a reasonable estimate or with an empty or null value.

To prevent skewing and misalignment of data, many statisticians advocate for replacing missing data with values representative of the average or expected value for that dataset. The methodology for determining a representative value and assigning it to a location within the data will vary depending upon the data and we cannot illustrate every example in this chapter. However, for example, if a dataset contained a list of temperatures across a range of dates, and one date was missing a temperature, that date can be assigned a temperature that was the average of the temperatures within the dataset.

We will examine a rather trivial example to demonstrate the issues surrounding data imputation. Let's assume the variable `tempList` contains average temperature data for each month of one year. Then we perform a simple calculation of the average and print out our results:

```
double[] tempList = {50,56,65,70,74,80,82,90,83,78,64,52};
double sum = 0;
for(double d : tempList){
      sum += d;
}
out.printf("The average temperature is %1$,.2f", sum/12);
```

Notice that for the numbers used in this execution, the output is as follows:

```
The average temperature is 70.33
```

Next we will mimic missing data by changing the first element of our array to zero before we calculate our `sum`:

```
double sum = 0;
tempList[0] = 0;
for(double d : tempList){
      sum += d;
}
out.printf("The average temperature is %1$,.2f", sum/12);
```

This will change the average temperature displayed in our output:

```
The average temperature is 66.17
```

Notice that while this change may seem rather minor, it is statistically significant. Depending upon the variation within a given dataset and how far the average is from zero or some other substituted value, the results of a statistical analysis may be significantly skewed. This does not mean zero should never be used as a substitute for null or otherwise invalid values, but other alternatives should be considered.

One alternative approach can be to calculate the average of the values in the array, excluding zeros or nulls, and then substitute the average in each position with missing data. It is important to consider the type of data and purpose of data analysis when making these decisions. For example, in the preceding example, will zero always be an invalid average temperature? Perhaps not if the temperatures were averages for Antarctica.

When it is essential to handle null data, Java's `Optional` class provides helpful solutions. Consider the following example, where we have a list of names stored as an array. We have set one value to `null` for the purposes of demonstrating these methods:

```
String useName = "";
String[] nameList =
      {"Amy","Bob","Sally","Sue","Don","Rick",null,"Betsy"};
Optional<String> tempName;
for(String name : nameList){
      tempName = Optional.ofNullable(name);
      useName = tempName.orElse("DEFAULT");
      out.println("Name to use = " + useName);
}
```

We first created a variable called `useName` to hold the name we will actually print out. We also created an instance of the `Optional` class called `tempName`. We will use this to test whether a value in the array is null or not. We then loop through our array and create and call the `Optional` class `ofNullable` method. This method tests whether a particular value is null or not. On the next line, we call the `orElse` method to either assign a value from the array to `useName` or, if the element is null, assign `DEFAULT`. Our output follows:

```
Name to use = Amy
Name to use = Bob
Name to use = Sally
Name to use = Sue
Name to use = Don
Name to use = Rick
Name to use = DEFAULT
Name to use = Betsy
```

The `Optional` class contains several other methods useful for handling potential null data. Although there are other ways to handle such instances, this Java 8 addition provides simpler and more elegant solutions to a common data analysis problem.

Subsetting data

It is not always practical or desirable to work with an entire set of data. In these cases, we may want to retrieve a subset of data to either work with or remove entirely from the dataset. There are a few ways of doing this supported by the standard Java libraries. First, we will use the `subSet` method of the `SortedSet` interface. We will begin by storing a list of numbers in a `TreeSet`. We then create a new `TreeSet` object to hold the subset retrieved from the list. Next, we print out our original list:

```
Integer[] nums = {12, 46, 52, 34, 87, 123, 14, 44};
TreeSet<Integer> fullNumsList = new TreeSet<Integer>(new
ArrayList<>(Arrays.asList(nums)));
SortedSet<Integer> partNumsList;
out.println("Original List: " + fullNumsList.toString()
    + " " + fullNumsList.last());
```

The `subSet` method takes two parameters, which specify the range of integers within the data we want to retrieve. The first parameter is included in the results while the second is exclusive. In our example that follows, we want to retrieve a subset of all numbers between the first number in our array `12` and `46`:

```
partNumsList = fullNumsList.subSet(fullNumsList.first(), 46);
out.println("SubSet of List: " + partNumsList.toString()
    + " " + partNumsList.size());
```

Our output follows:

```
Original List: [12, 14, 34, 44, 46, 52, 87, 123]
SubSet of List: [12, 14, 34, 44]
```

Another option is to use the `stream` method in conjunction with the `skip` method. The `stream` method returns a Java 8 Stream instance which iterates over the set. We will use the same `numsList` as in the previous example, but this time we will specify how many elements to skip with the `skip` method. We will also use the `collect` method to create a new `Set` to hold the new elements:

```
out.println("Original List: " + numsList.toString());
Set<Integer> fullNumsList = new TreeSet<Integer>(numsList);
Set<Integer> partNumsList = fullNumsList
         .stream()
         .skip(5)
         .collect(toCollection(TreeSet::new));
out.println("SubSet of List: " + partNumsList.toString());
```

When we print out the new subset, we get the following output where the first five elements of the sorted set are skipped. Because it is a `SortedSet`, we will actually be omitting the five lowest numbers:

```
Original List: [12, 46, 52, 34, 87, 123, 14, 44]
SubSet of List: [52, 87, 123]
```

At times, data will begin with blank lines or header lines that we wish to remove from our dataset to be analyzed. In our final example, we will read data from a file and remove all blank lines. We use a `BufferedReader` to read our data and employ a lambda expression to test for a blank line. If the line is not blank, we print the line to the screen:

```
try (BufferedReader br = new BufferedReader(new
FileReader("C:\\text.txt"))) {
   br
         .lines()
         .filter(s -> !s.equals(""))
         .forEach(s -> out.println(s));
} catch (IOException ex) {
   // Handle exceptions
}
```

Sorting text

Sometimes it is necessary to sort data during the cleaning process. The standard Java library provides several resources for accomplishing different types of sorts, with improvements added with the release of Java 8. In our first example, we will use the `Comparator` interface in conjunction with a lambda expression.

We start by declaring our `Comparator` variable `compareInts`. The first set of parenthesis after the equals sign contains the parameters to be passed to our method. Within the lambda expression, we call the `compare` method, which determines which integer is larger:

```
Comparator<Integer> compareInts = (Integer first, Integer second) ->
  Integer.compare(first, second);
```

We can now call the `sort` method as we did previously:

```
Collections.sort(numsList,compareInts);
out.println("Sorted integers using Lambda: " + numsList.toString());
```

Our output follows:

```
Sorted integers using Lambda: [12, 14, 34, 44, 46, 52, 87, 123]
```

We then mimic the process with our `wordsList`. Notice the use of the `compareTo` method rather than `compare`:

```
 Comparator<String> compareWords = (String first, String second) ->
first.compareTo(second);
Collections.sort(wordsList,compareWords);
out.println("Sorted words using Lambda: " + wordsList.toString());
```

When this code is executed, we should see the following output:

```
Sorted words using Lambda: [boat, cat, dog, house, road, zoo]
```

In our next example, we are going to use the `Collections` class to perform basic sorting on `String` and integer data. For this example, `wordList` and `numsList` are both `ArrayList` and are initialized as follows:

```
List<String> wordsList
        = Stream.of("cat", "dog", "house", "boat", "road", "zoo")
        .collect(Collectors.toList());
List<Integer> numsList = Stream.of(12, 46, 52, 34, 87, 123, 14, 44)
        .collect(Collectors.toList());
```

First, we will print our original version of each list followed by a call to the `sort` method. We then display our data, sorted in ascending fashion:

```
out.println("Original Word List: " + wordsList.toString());
Collections.sort(wordsList);
out.println("Ascending Word List: " + wordsList.toString());
out.println("Original Integer List: " + numsList.toString());
Collections.sort(numsList);
out.println("Ascending Integer List: " + numsList.toString());
```

The output follows:

```
Original Word List: [cat, dog, house, boat, road, zoo]
Ascending Word List: [boat, cat, dog, house, road, zoo]
Original Integer List: [12, 46, 52, 34, 87, 123, 14, 44]
Ascending Integer List: [12, 14, 34, 44, 46, 52, 87, 123]
```

Next, we will replace the `sort` method with the `reverse` method of the `Collections` class in our integer data example. This method simply takes the elements and stores them in reverse order:

```
out.println("Original Integer List: " + numsList.toString());
Collections.reverse(numsList);
out.println("Reversed Integer List: " + numsList.toString());
```

The output displays our new `numsList`:

```
Original Integer List: [12, 46, 52, 34, 87, 123, 14, 44]
Reversed Integer List: [44, 14, 123, 87, 34, 52, 46, 12]
```

In our next example, we handle the sort using the `Comparator` interface. We will continue to use our `numsList` and assume that no sorting has occurred yet. First we create two objects that implement the `Comparator` interface. The `sort` method will use these objects to determine the desired order when comparing two elements. The expression `Integer::compare` is a Java 8 method reference. This is can be used where a lambda expression is used:

```
out.println("Original Integer List: " + numsList.toString());
Comparator<Integer> basicOrder = Integer::compare;
Comparator<Integer> descendOrder = basicOrder.reversed();
Collections.sort(numsList,descendOrder);
out.println("Descending Integer List: " + numsList.toString());
```

After we execute this code, we will see the following output:

```
Original Integer List: [12, 46, 52, 34, 87, 123, 14, 44]
Descending Integer List: [123, 87, 52, 46, 44, 34, 14, 12]
```

In our last example, we will attempt a more complex sort involving two comparisons. Let's assume there is a `Dog` class that contains two properties, `name` and `age`, along with the necessary accessor methods. We will begin by adding elements to a new `ArrayList` and then printing the names and ages of each `Dog`:

```
ArrayList<Dogs> dogs = new ArrayList<Dogs>();
dogs.add(new Dogs("Zoey", 8));
dogs.add(new Dogs("Roxie", 10));
dogs.add(new Dogs("Kylie", 7));
dogs.add(new Dogs("Shorty", 14));
dogs.add(new Dogs("Ginger", 7));
dogs.add(new Dogs("Penny", 7));
out.println("Name " + " Age");
for(Dogs d : dogs){
      out.println(d.getName() + " " + d.getAge());
}
```

Our output should resemble:

```
Name Age
Zoey 8
Roxie 10
Kylie 7
Shorty 14
Ginger 7
Penny 7
```

Next, we are going to use method chaining and the double colon operator to reference methods from the `Dog` class. We first call `comparing` followed by `thenComparing` to specify the order in which comparisons should occur. When we execute the code, we expect to see the `Dog` objects sorted first by `Name` and then by `Age`:

```
dogs.sort(Comparator.comparing(Dogs::getName).thenComparing(Dogs::getAge));
out.println("Name " + " Age");
for(Dogs d : dogs){
      out.println(d.getName() + " " + d.getAge());
}
```

Our output follows:

```
Name Age
Ginger 7
Kylie 7
Penny 7
Roxie 10
Shorty 14
Zoey 8
```

Now we will switch the order of comparison. Notice how the age of the dog takes priority over the name in this version:

```
dogs.sort(Comparator.comparing(Dogs::getAge).thenComparing(Dogs::getName));
out.println("Name " + " Age");
for(Dogs d : dogs){
      out.println(d.getName() + " " + d.getAge());
}
```

And our output is:

```
Name Age
Ginger 7
Kylie 7
Penny 7
Zoey 8
Roxie 10
Shorty 14
```

Data validation

Data validation is an important part of data science. Before we can analyze and manipulate data, we need to verify that the data is of the type expected. We have organized our code into simple methods designed to accomplish very basic validation tasks. The code within these methods can be adapted into existing applications.

Validating data types

Sometimes we simply need to validate whether a piece of data is of a specific type, such as integer or floating point data. We will demonstrate in the next example how to validate integer data using the `validateInt` method. This technique is easily modified for the other major data types supported in the standard Java library, including `Float` and `Double`.

We need to use a try-catch block here to catch a `NumberFormatException`. If an exception is thrown, we know our data is not a valid integer. We first pass our text to be tested to the `parseInt` method of the `Integer` class. If the text can be parsed as an integer, we simply print out the integer. If an exception is thrown, we display information to that effect:

```
public static void validateInt(String toValidate){
try{
      int validInt = Integer.parseInt(toValidate);
      out.println(validInt + " is a valid integer");
}catch(NumberFormatException e){
      out.println(toValidate + " is not a valid integer");

}
```

We will use the following method calls to test our method:

```
validateInt("1234");
validateInt("Ishmael");
```

The output follows:

```
1234 is a valid integer
Ishmael is not a valid integer
```

The Apache Commons contain an `IntegerValidator` class with additional useful functionalities. In this first example, we simply duplicate the process from before, but use `IntegerValidator` methods to accomplish our goal:

```
public static String validateInt(String text){
      IntegerValidator intValidator = IntegerValidator.getInstance();
      if(intValidator.isValid(text)){
            return text + " is a valid integer";
      }else{
            return text + " is not a valid integer";
      }
}
```

We again use the following method calls to test our method:

```
validateInt("1234");
validateInt("Ishmael");
```

The output follows:

```
1234 is a valid integer
Ishmael is not a valid integer
```

The `IntegerValidator` class also provides methods to determine whether an integer is greater than or less than a specific value, compare the number to a ranger of numbers, and convert `Number` objects to `Integer` objects. Apache Commons has a number of other validator classes. We will examine a few more in the rest of this section.

Validating dates

Many times our data validation is more complex than simply determining whether a piece of data is the correct type. When we want to verify that the data is a date for example, it is insufficient to simply verify that it is made up of integers. We may need to include hyphens and slashes, or ensure that the year is in two-digit or four-digit format.

To do this, we have created another simple method called `validateDate`. The method takes two `String` parameters, one to hold the date to validate and the other to hold the acceptable date format. We create an instance of the `SimpleDateFormat` class using the format specified in the parameter. Then we call the `parse` method to convert our `String` date to a `Date` object. Just as in our previous integer example, if the data cannot be parsed as a date, an exception is thrown and the method returns. If, however, the `String` can be parsed to a date, we simply compare the format of the test date with our acceptable format to determine whether the date is valid:

```
 public static String validateDate(String theDate, String dateFormat){
      try {
            SimpleDateFormat format = new SimpleDateFormat(dateFormat);
            Date test = format.parse(theDate);
            if(format.format(test).equals(theDate)){
                  return theDate.toString() + " is a valid date";
            }else{
                  return theDate.toString() + " is not a valid date";
            }
      } catch (ParseException e) {
            return theDate.toString() + " is not a valid date";
      }
}
```

We make the following method calls to test our method:

```
String dateFormat = "MM/dd/yyyy";
out.println(validateDate("12/12/1982",dateFormat));
out.println(validateDate("12/12/82",dateFormat));
out.println(validateDate("Ishmael",dateFormat));
```

The output follows:

```
12/12/1982 is a valid date
12/12/82 is not a valid date
Ishmael is not a valid date
```

This example highlights why it is important to consider the restrictions you place on data. Our second method call did contain a legitimate date, but it was not in the format we specified. This is good if we are looking for very specifically formatted data. But we also run the risk of leaving out useful data if we are too restrictive in our validation.

Validating e-mail addresses

It is also common to need to validate e-mail addresses. While most e-mail addresses have the `@` symbol and require at least one period after the symbol, there are many variations. Consider that each of the following examples can be valid e-mail addresses:

- `myemail@mail.com`
- `MyEmail@some.mail.com`
- `My.Email.123!@mail.net`

One option is to use regular expressions to attempt to capture all allowable e-mail addresses. Notice that the regular expression used in the method that follows is very long and complex. This can make it easy to make mistakes, miss valid e-mail addresses, or accept invalid addresses as valid. But a carefully crafted regular expression can be a very powerful tool.

We use the `Pattern` and `Matcher` classes to compile and execute our regular expression. If the pattern of the e-mail we pass in matches the regular expression we defined, we will consider that text to be a valid e-mail address:

```
public static String validateEmail(String email) {
      String emailRegex = "^[a-zA-Z0-9.!$'*+/=?^_`{|}~-" +
          "]+@((\\[[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\." +
          "[0-9]{1,3}\\])|(([a-zAZ\\-0-9]+\\.)+[a-zA-Z]{2,}))$";
      Pattern.compile(emailRegex);
      Matcher matcher = pattern.matcher(email);
      if(matcher.matches()){
            return email + " is a valid email address";
      }else{
            return email + " is not a valid email address";
      }
}
```

We make the following method calls to test our data:

```
out.println(validateEmail("myemail@mail.com"));
out.println(validateEmail("My.Email.123!@mail.net"));
out.println(validateEmail("myEmail"));
```

The output follows:

```
myemail@mail.com is a valid email address
My.Email.123!@mail.net is a valid email address
myEmail is not a valid email address
```

There is a standard Java library for validating e-mail addresses as well. In this example, we use the `InternetAddress` class to validate whether a given string is a valid e-mail address or not:

```
public static String validateEmailStandard(String email){
    try{
        InternetAddress testEmail = new InternetAddress(email);
        testEmail.validate();
        return email + " is a valid email address";
    }catch(AddressException e){
        return email + " is not a valid email address";
    }
}
```

When tested against the same data as in the previous example, our output is identical. However, consider the following method call:

```
out.println(validateEmailStandard("myEmail@mail"));
```

Despite not being in standard e-mail format, the output is as follows:

```
myEmail@mail is a valid email address
```

Additionally, the `validate` method by default accepts local e-mail addresses as valid. This is not always desirable, depending upon the purpose of the data.

One last option we will look at is the Apache Commons `EmailValidator` class. This class's `isValid` method examines an e-mail address and determines whether it is valid or not. Our `validateEmail` method shown previously is modified as follows to use `EmailValidator`:

```
public static String validateEmailApache(String email){
      email = email.trim();
      EmailValidator eValidator = EmailValidator.getInstance();
      if(eValidator.isValid(email)){
            return email + " is a valid email address.";
      }else{
            return email + " is not a valid email address.";
      }
}
```

Validating ZIP codes

Postal codes are generally formatted specific to their country or local requirements. For this reason, regular expressions are useful because they can accommodate any postal code required. The example that follows demonstrates how to validate a standard United States postal code, with or without the hyphen and last four digits:

```
public static void validateZip(String zip){
      String zipRegex = "^[0-9]{5}(?:-[0-9]{4})?$";
      Pattern pattern = Pattern.compile(zipRegex);
      Matcher matcher = pattern.matcher(zip);
      if(matcher.matches()){
            out.println(zip + " is a valid zip code");
      }else{
            out.println(zip + " is not a valid zip code");
      }
}
```

We make the following method calls to test our data:

```
out.println(validateZip("12345"));
out.println(validateZip("12345-6789"));
out.println(validateZip("123"));
```

The output follows:

```
12345 is a valid zip code
12345-6789 is a valid zip code
123 is not a valid zip code
```

Validating names

Names can be especially tricky to validate because there are so many variations. There are no industry standards or technical limitations, other than what characters are available on the keyboard. For this example, we have chosen to use Unicode in our regular expression because it allows us to match any character from any language. The Unicode property `\\p{L}` provides this flexibility. We also use `\\s-'`, to allow spaces, apostrophes, commas, and hyphens in our name fields. It is possible to perform string cleaning, as discussed earlier in this chapter, before attempting to match names. This will simplify the regular expression required:

```
public static void validateName(String name){
      String nameRegex = "^[\\p{L}\\s-',]+$";
      Pattern pattern = Pattern.compile(nameRegex);
      Matcher matcher = pattern.matcher(name);
```

```
        if(matcher.matches()){
              out.println(name + " is a valid name");
        }else{
              out.println(name + " is not a valid name");
        }
  }
```

We make the following method calls to test our data:

```
validateName("Bobby Smith, Jr.");
validateName("Bobby Smith the 4th");
validateName("Albrecht Muller");
validateName("Francois Moreau");
```

The output follows:

```
Bobby Smith, Jr. is a valid name
Bobby Smith the 4th is not a valid name
Albrecht Muller is a valid name
Francois Moreau is a valid name
```

Notice that the comma and period in `Bobby Smith, Jr.` are acceptable, but the `4` in `4th` is not. Additionally, the special characters in `Francois` and `Muller` are considered valid.

Cleaning images

While image processing is a complex task, we will introduce a few techniques to clean and extract information from an image. This will provide the reader with some insight into image processing. We will also demonstrate how to extract text data from an image using **Optical Character Recognition (OCR).**

There are several techniques used to improve the quality of an image. Many of these require tweaking of parameters to get the improvement desired. We will demonstrate how to:

- Enhance an image's contrast
- Smooth an image
- Brighten an image
- Resize an image
- Convert images to different formats

We will use OpenCV (`http://opencv.org/`), an open source project for image processing. There are several classes that we will use:

- `Mat`: This represents an n-dimensional array holding image data such as channel, grayscale, or color values
- `Imgproc`: Possesses many methods that process an image
- `Imgcodecs`: Possesses methods to read and write image files

The OpenCV Javadocs is found at `http://docs.opencv.org/java/2.4.9/`. In the examples that follow, we will use Wikipedia images since they can be freely downloaded. Specifically we will use the following images:

- **Parrot image**: `https://en.wikipedia.org/wiki/Grayscale#/media/File:Grayscale_8bits_palette_sample_image.png`
- **Cat image**: `https://en.wikipedia.org/wiki/Cat#/media/File:Kittyply_edit1.jpg`

Changing the contrast of an image

Here we will demonstrate how to enhance a black-and-white image of a parrot. The `Imgcodecs` class's `imread` method reads in the image. Its second parameter specifies the type of color used by the image, which is grayscale in this case. A new `Mat` object is created for the enhanced image using the same size and color type as the original.

The actual work is performed by the `equalizeHist` method. This equalizes the histogram of the image which has the effect of normalizing the brightness and increases the contrast of the image. An image histogram is a histogram representing the tonal distribution of an image. **Tonal** is also referred to as lightness. It represents the variation in the brightness found in an image.

The last step is to write out the image.

```
Mat source = Imgcodecs.imread("GrayScaleParrot.png",
        Imgcodecs.CV_LOAD_IMAGE_GRAYSCALE);
Mat destination = new Mat(source.rows(), source.cols(), source.type());
Imgproc.equalizeHist(source, destination);
Imgcodecs.imwrite("enhancedParrot.jpg", destination);
```

The following is the original image:



The enhanced image follows:



Smoothing an image

Smoothing an image, also called **blurring**, will make the edges of an image smoother. Blurring is the process of making an image less distinct. We recognize blurred objects when we take a picture with the camera out of focus. Blurring can be used for special effects. Here, we will use it to create an image that we will then sharpen.

The following example loads an image of a cat and repeatedly applies the `blur` method to the image. In this example, the process is repeated `25` times. Increasing the number of iterations will result in more blur or smoothing.

The third argument of the blur method is the blurring kernel size. The kernel is a matrix of pixels, 3 by 3 in this example, that is used for convolution. This is the process of multiplying each element of an image by weighted values of its neighbors. This allows neighboring values to effect an element's value:

```
Mat source = Imgcodecs.imread("cat.jpg");
Mat destination = source.clone();
for (int i = 0; i < 25; i++) {
    Mat sourceImage = destination.clone();
    Imgproc.blur(sourceImage, destination, new Size(3.0, 3.0));
}
Imgcodecs.imwrite("smoothCat.jpg", destination);
```

The following is the original image:



The enhanced image follows:



Brightening an image

The `convertTo` method provides a means of brightening an image. The original image is copied to a new image where the contrast and brightness is adjusted. The first parameter is the destination image. The second specifies that the type of image should not be changed. The third and fourth parameters control the contrast and brightness respectively. The first value is multiplied by this value while the second is added to the multiplied value:

```
Mat source = Imgcodecs.imread("cat.jpg");
Mat destination = new Mat(source.rows(), source.cols(),
        source.type());
source.convertTo(destination, -1, 1, 50);
Imgcodecs.imwrite("brighterCat.jpg", destination);
```

The enhanced image follows:



Resizing an image

Sometimes it is desirable to resize an image. The `resize` method shown next illustrates how this is done. The image is read in and a new `Mat` object is created. The `resize` method is then applied where the width and height are specified in the `Size` object parameter. The resized image is then saved:

```
Mat source = Imgcodecs.imread("cat.jpg");
Mat resizeimage = new Mat();
Imgproc.resize(source, resizeimage, new Size(250, 250));
Imgcodecs.imwrite("resizedCat.jpg", resizeimage);
```

The enhanced image follows:



Converting images to different formats

Another common operation is to convert an image that uses one format into an image that uses a different format. In OpenCV, this is easy to accomplish as shown next. The image is read in and then immediately written out. The extension of the file is used by the `imwrite` method to convert the image to the new format:

```
Mat source = Imgcodecs.imread("cat.jpg");
Imgcodecs.imwrite("convertedCat.jpg", source);
Imgcodecs.imwrite("convertedCat.jpeg", source);
Imgcodecs.imwrite("convertedCat.webp", source);
Imgcodecs.imwrite("convertedCat.png", source);
Imgcodecs.imwrite("convertedCat.tiff", source);
```

The images can now be used for specialized processing if necessary.

Summary

Many times, half the battle in data science is manipulating data so that it is clean enough to work with. In this chapter, we examined many techniques for taking real-world, messy data and transforming it into workable datasets. This process is generally known as data cleaning, wrangling, reshaping, or munging. Our focus was on core Java techniques, but we also examined third-party libraries.

Before we can clean data, we need to have a solid understanding of the format of our data. We discussed CSV data, spreadsheets, PDF, and JSON file types, as well as provided several examples of manipulating text file data. As we examined text data, we looked at multiple approaches for processing the data, including tokenizers, `Scanners`, and `BufferedReaders`. We showed ways to perform simple cleaning operations, remove stop words, and perform find and replace functions.

This chapter also included a discussion on data imputation and the importance of identifying and rectifying missing data situations. Missing data can cause problems during data analysis and we proposed different methods for dealing with this problem. We demonstrated how to retrieve subsets of data and sort data as well.

Finally, we discussed image cleaning and demonstrated several methods of modifying image data. This included changing contrast, smoothing, brightening, and resizing information. We concluded with a discussion on extracting text imposed on an image.

With this background, we will introduce basic statistical methods and their Java support in the next chapter.

4
Data Visualization

The human mind is often good at seeing patterns, trends, and outliers in visual representations. The large amount of data present in many data science problems can be analyzed using visualization techniques. Visualization is appropriate for a wide range of audiences, ranging from analysts, to upper-level management, to clientele. In this chapter, we present various visualization techniques and demonstrate how they are supported in Java.

In this chapter, we will illustrate how to create different types of graph, plot, and chart. The majority of the examples use JavaFX, with a few using a free library called **GRAphing Library** (**GRAL**). There are several open source Java plotting libraries available. A brief comparison of several of these libraries can be found at `https://github.com/eseifert/gral/wiki/comparison`. We chose JavaFX because it is packaged as part of Java SE.

GRAL is used to illustrate plots that are not as easily created using JavaFX. GRAL is a free Java library useful for creating a variety of charts and graphs. This graphing library provides flexibility in types of plots, axis formatting, and export options. GRAL resources (`http://trac.erichseifert.de/gral/`) include example code and helpful how to sections.

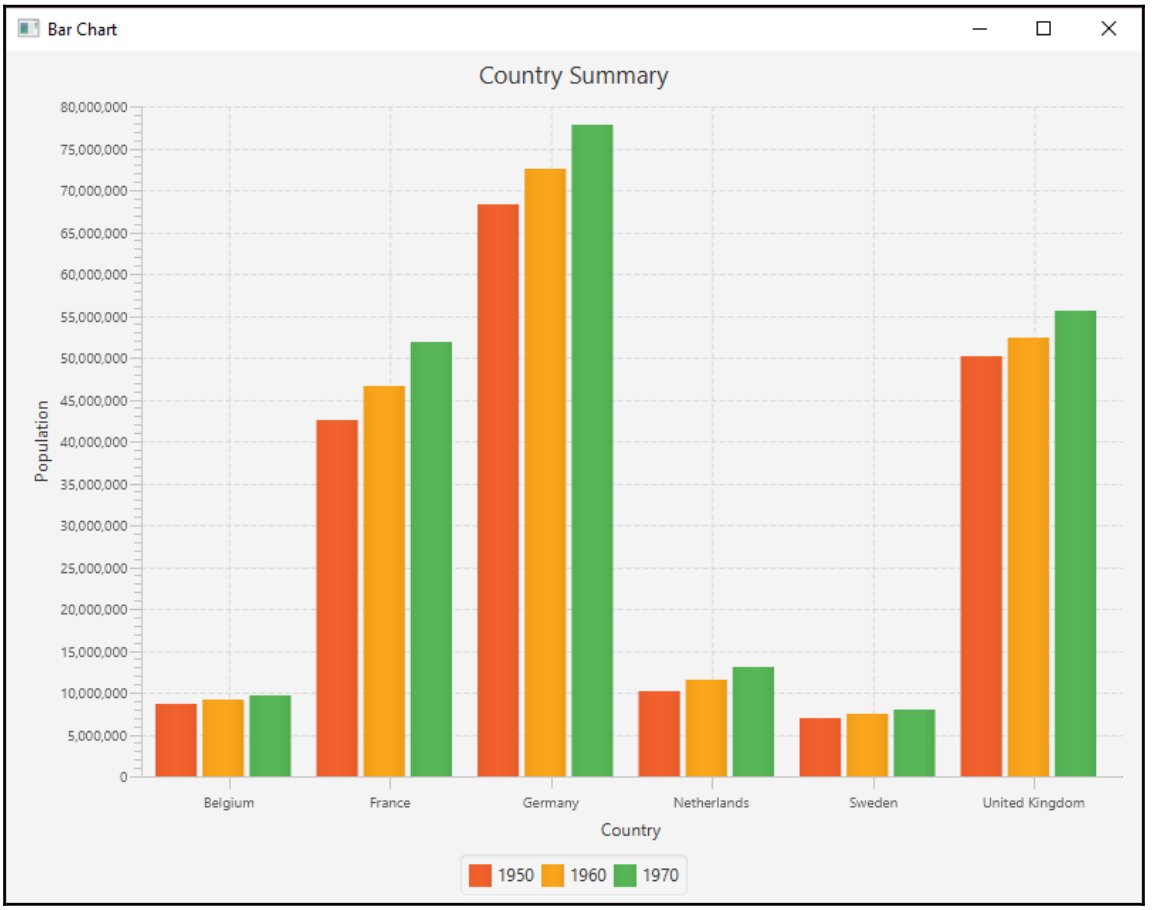
Visualization is an important step in data analysis because it allows us to conceive of large datasets in practical and meaningful ways. We can look at small datasets of values and perhaps draw conclusions from the patterns we see, but this is an overwhelming and unreliable process. Using visualization tools helps us identify potential problems or unexpected data results, as well as construct meaningful interpretations of good data.

One example of the usefulness of data visualization comes with the presence of **outliers**. Visualizing data allows us to quickly see data results significantly outside of our expectations, and we can choose how to modify the data to build a clean and usable dataset. This process allows us to see errors quickly and deal with them before they become a problem later on. Additionally, visualization allows us to easily classify information and help analysts organize their inquiries in a manner best suited to their particular dataset.

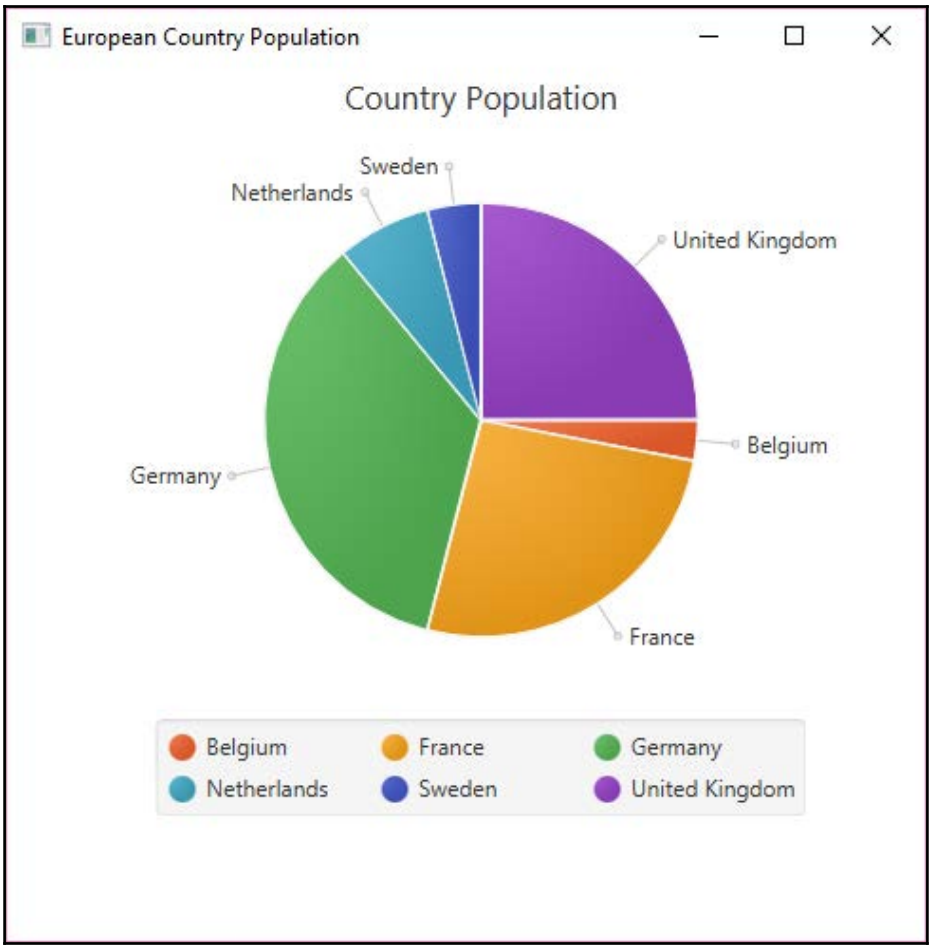
Understanding plots and graphs

There are many types of visual expression available to aid in visualization. We are going to briefly discuss the most common and useful ones, and then demonstrate several Java techniques for achieving these types of expression. The choice of graph, or other visualization tool will depend upon the dataset and application needs and constraints.

A **bar chart** is a very common technique for displaying relationships in data. In this type of graph, data is represented in either vertical or horizontal bars placed along an *X* and Y axis. The data is scaled so the values represented by each bar can be compared to one another. The following is a simple example of a bar chart we will create in the *Using country as the category* section:



A **pie chart** is most useful when you want to demonstrate a value in relation to a larger set. Think of this as a way to visualize how large the piece of pie is in relation to the entire pie. The following is a simple example of a pie chart showing the distribution of population for selected European countries:

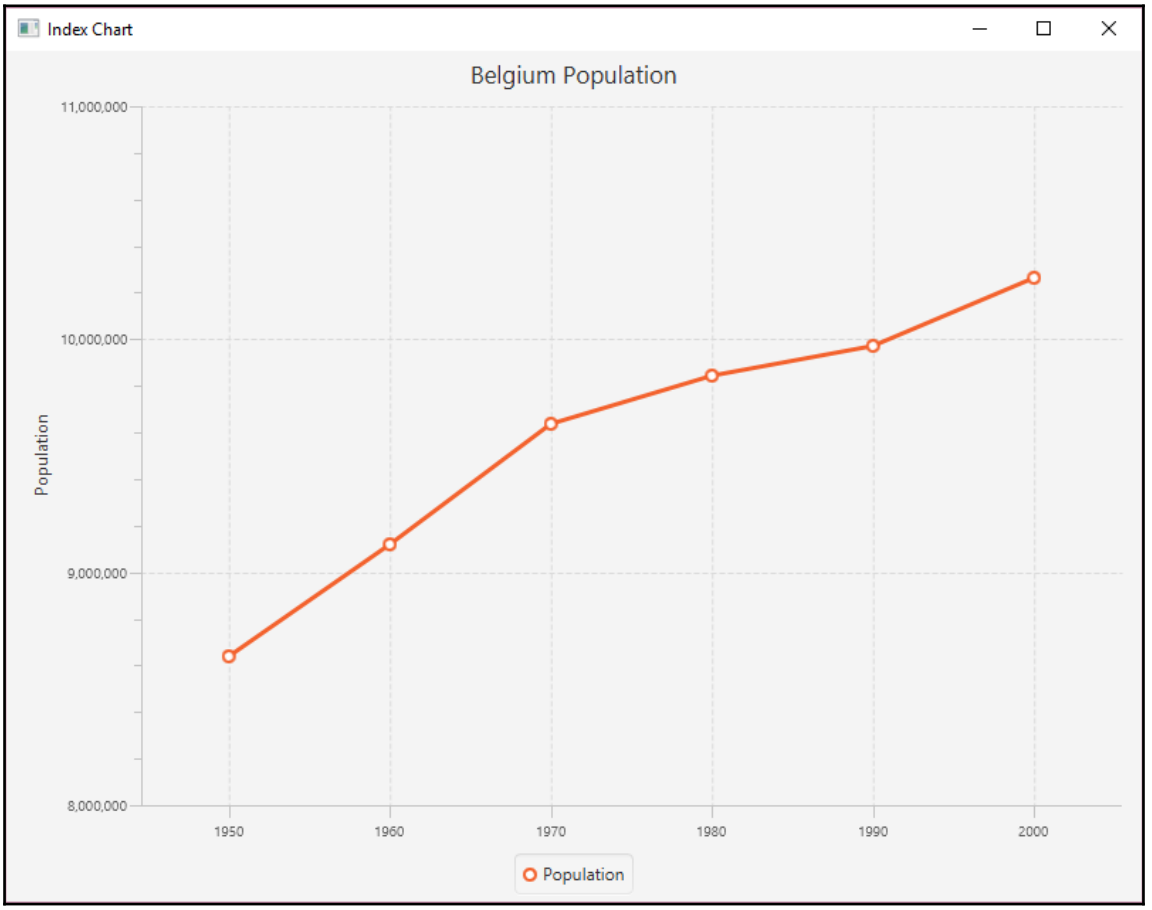


Time series graphs are a special type of graph used for displaying time-related values. These are most appropriate when the data analysis requires an understanding of how data changes over a period of time. In these graphs, the vertical axis corresponds to the values and the horizontal axis corresponds to particular points in time. In particular, this type of graph can be useful for identifying trends across time,or suggesting correlations between data values and particular events within a given time period.

For example, stock prices and home prices will change, but their rate of change varies. Pollution levels and crime rates also change over time. There are several techniques that visualize this type of data. Often, specific values are not as important as their trend over time.

An **index chart** is also called a line chart. **Line charts** use the *X* and *Y* axis to plot points on a grid. They can be used to represent time series data. These points are connected by lines, and these lines are used to compare values of multiple data at one time. This comparison is usually achieved by plotting independent variables, such as time, along the *X* axis, and independent variables, such as frequency or percentages, along the *Y* axis.

The following is a simple example of an index chart showing the distribution of population for selected European countries:



When we wish to arrange larger amounts of data in a compact and useful manner, we may opt for a stem and leaf plot. This type of visual expression allows you to demonstrate the correlation of one value to many values in a readable manner. The stem refers to a data value, and the leaves are the corresponding data points. One common example of this is a train timetable. In the following table, the departure times for a train are listed:

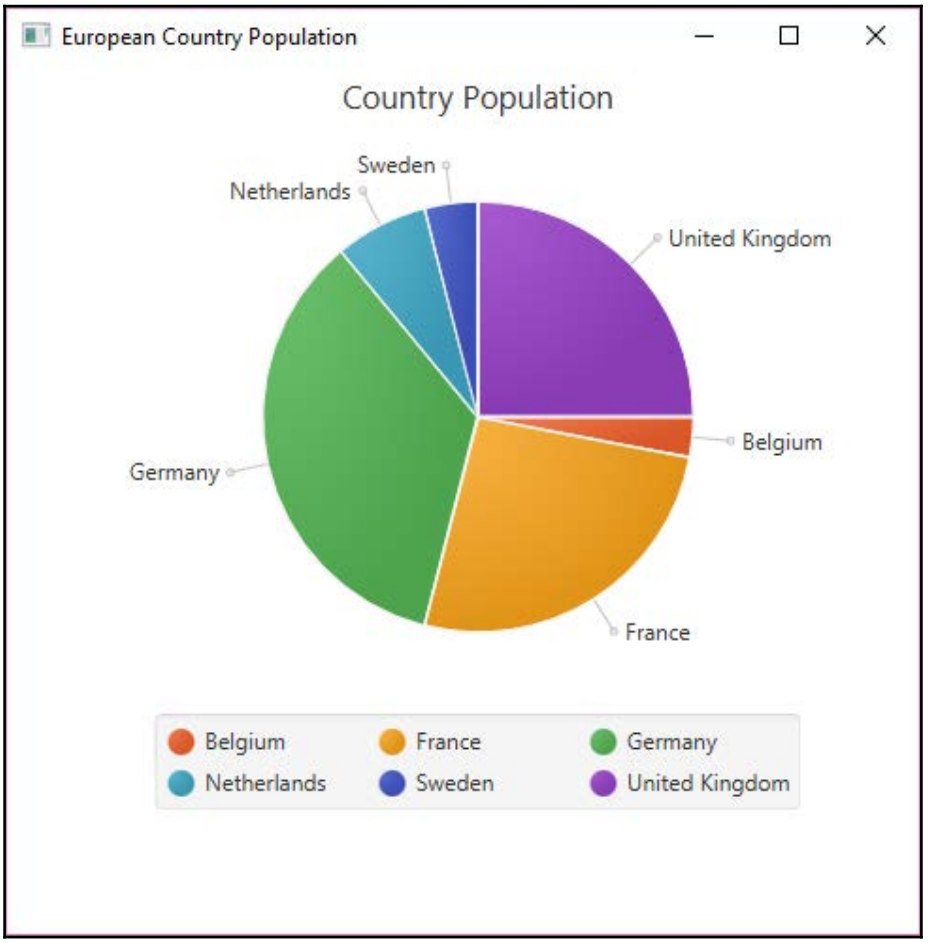
06:15	06 :20	06:25	06:30
06:40	06:45	06:55	07:15
07:20	07:25	07:30	07:40
07:45	07:55	08:00	08:12
08:24	08:36	08:48	09:00
09:12	09:24	09:36	09:48
10:00	10:12	10:24	10:36
10:48			

However, this table can be hard to read. Instead, in the following partial stem and leaf plot, the stem represents the hours at which a train may depart, while the leaves represent the minutes within each hour:

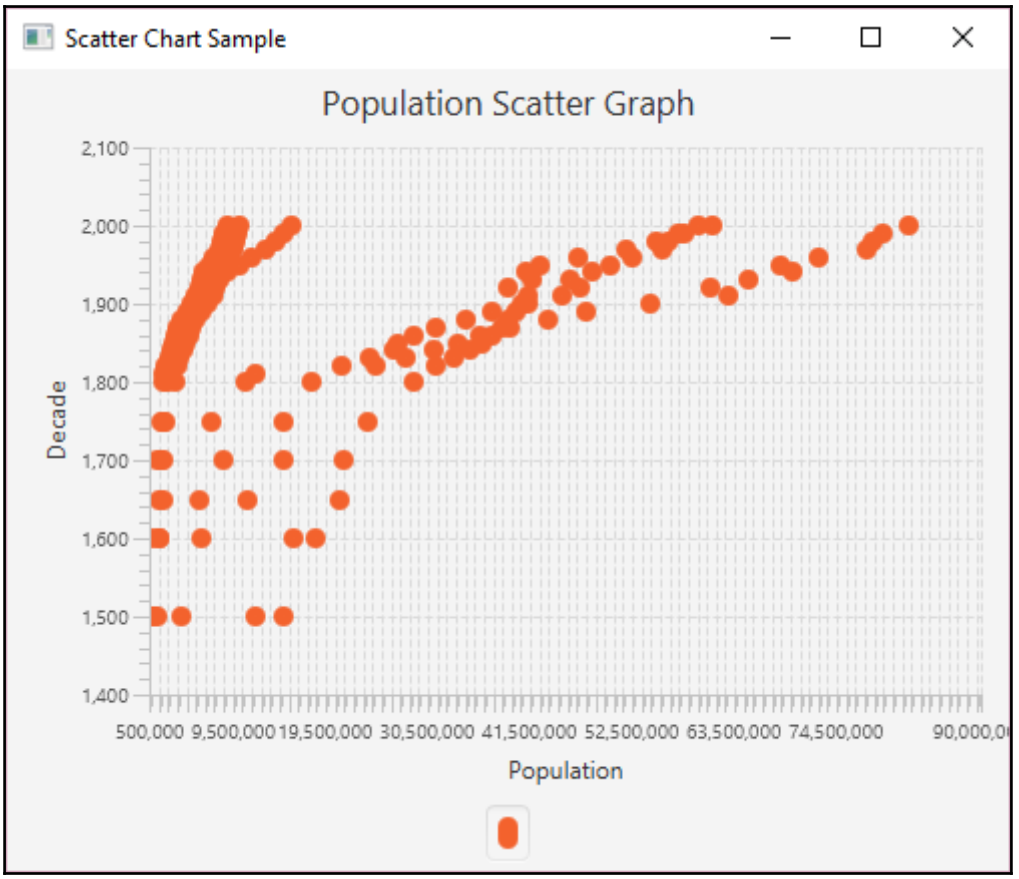
Hour	**Minute**
06	*:15 :20 :25 :30 :40 :45 :55*
07	*:15 :20 :25 :30 :40 :45 :55*
08	*:00 :12 :24 :36 :48*
09	*:00 :12 :24 :36 :48*
10	*:00 :12 :24 :36 :48*

This is much easier to read and process.

A very popular form of visualization in statistical analysis is the **histogram**. Histograms allow you to display frequencies within data using bars, similar to a bar chart. The main difference is that histograms are used to identify frequencies and trends within a dataset while bar charts are used to compare specific data values within a dataset. The following is an example of a histogram we will create in the *Creating histograms* section:



A **scatter plot** is simply collections of points, and analysis techniques, such as correlation or regression, can be used to identify trends within these types of graph. In the following scatter chart, as developed in *Creating scatter charts*, the population along the *X* axis is plotted against the decade along the *Y* axis:



Visual analysis goals

Each type of visual expression lends itself to different types of data and data analysis purposes. One common purpose of data analysis is **data classification**. This involves determining which subset within a dataset a particular data value belongs to. This process may occur early in the data analysis process because breaking data apart into manageable and related pieces simplifies the analysis process. Often, classification is not the end goal but rather an important intermediary step before further analysis can be undertaken.

Regression analysis is a complex and important form of data analysis. It involves studying relationships between independent and dependent variables, as well as multiple independent variables. This type of statistical analysis allows the analyst to identify ranges of acceptable or expected values and determine how individual values may fit into a larger dataset. Regression analysis is a significant part of machine learning, and we will discuss it in more detail in `Chapter 5`, *Statisitcal Data Analysis Techniques*.

Clustering allows us to identify groups of data points within a particular set or class. While classification sorts data into similar types of datasets, clustering is concerned with the data within the set. For example, we may have a large dataset containing all feline species in the world, in the family Felidae. We could then classify these cats into two groups, Pantherinae (containing most larger cats) and Felinae (all other cats). Clustering would involve grouping subsets of similar cats within one of these classifications. For example, all tigers could be a cluster within the Pantherinae group.

Sometimes, our data analysis requires that we extract specific types of information from our dataset. The process of selecting the data to extract is known as **attribute selection** or **feature selection**. This process helps analysts simplify the data models and allows us to overcome issues with redundant or irrelevant information within our dataset.

With this introduction to basic plot and chart types, we will discuss Java support for creating these plots and charts.

Creating index charts

An index chart is a line chart that shows the percentage change of something over time. Frequently, such a chart is based on a single data attribute. In the following example, we will be using the Belgian population for six decades. The data is a subset of population data found at `https://ourworldindata.org/grapher/population-by-country?tab=data`:

Decade	Population
1950	8639369
1960	9118700
1970	9637800
1980	9846800
1990	9969310
2000	10263618

We start by creating the `MainApp` class, which extends `Application`. We create a series of instance variables. The `XYChart.Series` class represents a series of data points for some plot. In our case, this will be for the decades and population, which we will initialize shortly. The next declaration is for the `CategoryAxis` and `NumberAxis` instances. These represent the *X* and *Y* axes respectively. The declaration for the *Y* axis includes range and increment values for the population. This makes the chart a bit more readable. The last declaration is a string variable for the country:

```
public class MainApp extends Application {
    final XYChart.Series<String, Number> series =
        new XYChart.Series<>();
    final CategoryAxis xAxis = new CategoryAxis();
    final NumberAxis yAxis =
        new NumberAxis(8000000, 11000000, 1000000);
    final static String belgium = "Belgium";
    ...
}
```

In JavaFX, the `main` method usually launches the application using the base class `launch` method. Eventually, the `start` method is called, which we override. In this example, we call the `simpleLineChart` method where the user interface is created:

```
public static void main(String[] args) {
    launch(args);
}

public void start(Stage stage) {
    simpleIndexChart (stage);
}
```

The `simpleLineChart` follows and is passed an instance of the `Stage` class. This represents the client area of the application's window. We start by setting a title for the application and the line chart proper. The label of the *Y* axis is set. An instance of the `LineChart` class is initialized using the *X* and *Y* axis instances. This class represents the line chart:

```
public void simpleIndexChart (Stage stage) {
    stage.setTitle("Index Chart");
    lineChart.setTitle("Belgium Population");
    yAxis.setLabel("Population");
    final LineChart<String, Number> lineChart
            = new LineChart<>(xAxis, yAxis);

    ...
}
```

The series is given a name, and then the population for each decade is added to the series using the `addDataItem` helper method:

```
series.setName("Population");
addDataItem(series, "1950", 8639369);
addDataItem(series, "1960", 9118700);
addDataItem(series, "1970", 9637800);
addDataItem(series, "1980", 9846800);
addDataItem(series, "1990", 9969310);
addDataItem(series, "2000", 10263618);
```

The `addDataItem` method follows, which creates an `XYChart.Data` class instance using the `String` and `Number` values passed to it. It then adds the instance to the series:

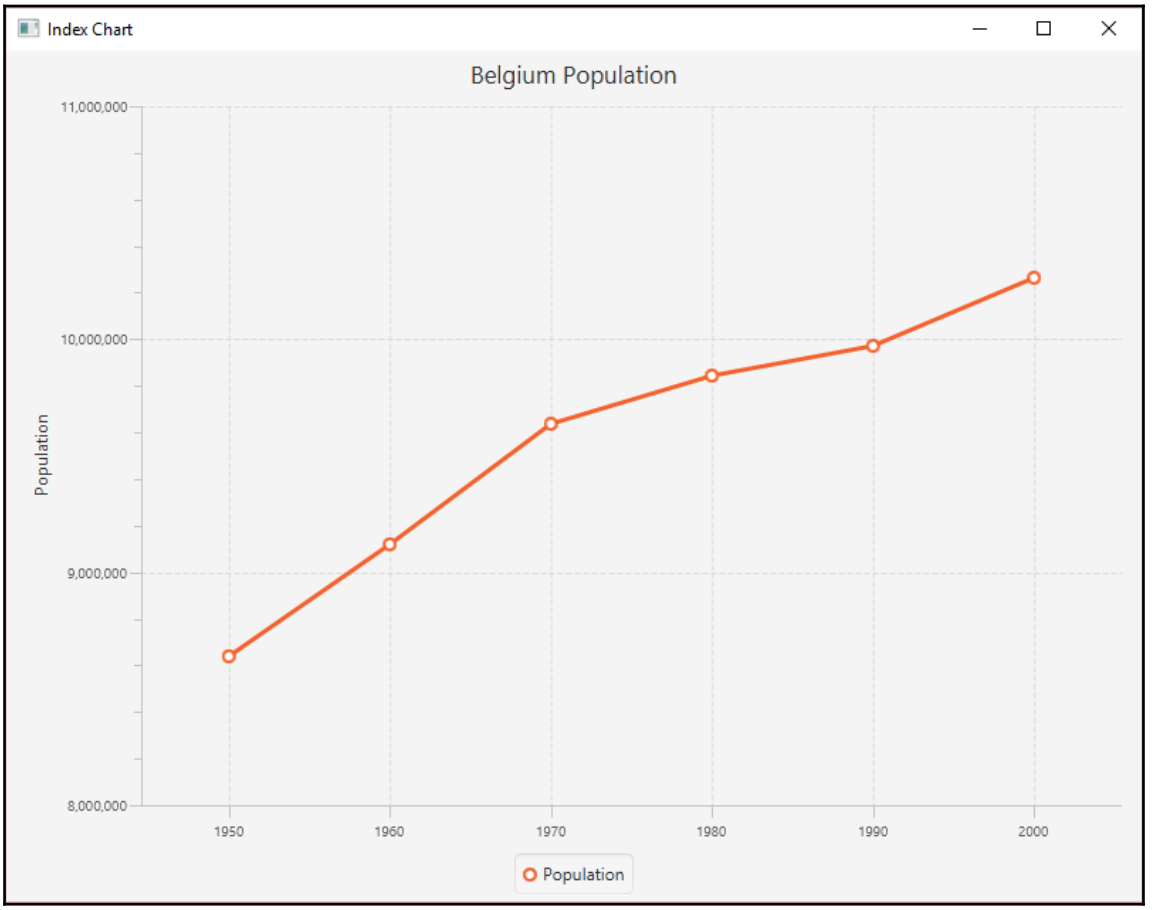
```
public void addDataItem(XYChart.Series<String, Number> series,
        String x, Number y) {
    series.getData().add(new XYChart.Data<>(x, y));
}
```

The last part of the `simpleLineChart` method creates a `Scene` class instance that represents the content of the `stage`. JavaFX uses the concept of a stage and scene to deal with the internals of the application's GUI.

The `scene` is created using a line chart, and the application's size is set to `800` by `600` pixels. The series is then added to the line chart and `scene` is added to `stage`. The `show` method displays the application:

```
Scene scene = new Scene(lineChart, 800, 600);
lineChart.getData().add(series);
stage.setScene(scene);
stage.show();
```

When the application executes the following window will be displayed:



Creating bar charts

A bar chart uses two axes with rectangular bars that can be either positioned either vertically or horizontally. The length of a bar is proportional to the value it represents. A bar chart can be used to show time series data.

In the following series of examples, we will be using a set of European country populations for three decades, as listed in the following table. The data is a subset of population data found at `https://ourworldindata.org/grapher/population-by-country?tab=data`:

Country	**1950**	**1960**	**1970**
Belgium	8,639,369	9,118,700	9,637,800
France	42,518,000	46,584,000	51,918,000
Germany	68,374,572	72,480,869	77,783,164
Netherlands	10,113,527	11,486,000	13,032,335
Sweden	7,014,005	7,480,395	8,042,803
United Kingdom	50,127,000	52,372,000	55,632,000

The first of three bar charts will be constructed using JavaFX. We start with a series of declarations for the countries as part of a class that extends the `Application` class:

```
public class MainApp extends Application {
    final static String belgium = "Belgium";
    final static String france = "France";
    final static String germany = "Germany";
    final static String netherlands = "Netherlands";
    final static String sweden = "Sweden";
    final static String unitedKingdom = "United Kingdom";

...
}
```

Next, we declared a series of instance variables that represent the parts of a graph. The first are `CategoryAxis` and `NumberAxis` instances:

```
final CategoryAxis xAxis = new CategoryAxis();
final NumberAxis yAxis = new NumberAxis();
```

The population and country data is stored in a series of `XYChart.Series` instances. Here, we have declared six different series, which use a string and number pair. The first example does not use all six series, but later examples will. We will initially assign a country string and its corresponding population to three series. These series will represent the populations for the decades `1950`, `1960`, and `1970`:

```
final XYChart.Series<String, Number> series1 =
    new XYChart.Series<>();
final XYChart.Series<String, Number> series2
    new XYChart.Series<>();
```

```
final XYChart.Series<String, Number> series3 =
    new XYChart.Series<>();
final XYChart.Series<String, Number> series4 =
    new XYChart.Series<>();
final XYChart.Series<String, Number> series5 =
    new XYChart.Series<>();
final XYChart.Series<String, Number> series6 =
    new XYChart.Series<>();
```

We will start with two simple bar charts. The first one will show the countries as categories where the year changes occur within the category on the *X* axis and the population along the *Y* axis. The second shows the decades as categories containing the counties. The last example is a stacked bar chart.

Using country as the category

The elements of the bar chart are set up in the `simpleBarChartByCountry` method. The title of the chart is set and a `BarChart` class instance is created using the two axes. The chart, its *X* axis, and its *Y* axis also have labels that are initialized here:

```
public void simpleBarChartByCountry(Stage stage) {
    stage.setTitle("Bar Chart");
    final BarChart<String, Number> barChart
                = new BarChart<>(xAxis, yAxis);
    barChart.setTitle("Country Summary");
    xAxis.setLabel("Country");
    yAxis.setLabel("Population");
    ...
}
```

Next, the first three series are initialized with a name, and then the country and population data for that series. A helper method, `addDataItem`, as introduced in the previous section, is used to add the data to each series:

```
series1.setName("1950");
addDataItem(series1,belgium, 8639369);
addDataItem(series1,france, 42518000);
addDataItem(series1,germany, 68374572);
addDataItem(series1,netherlands, 10113527);
addDataItem(series1,sweden, 7014005);
addDataItem(series1,unitedKingdom, 50127000);

series2.setName("1960");
addDataItem(series2,belgium, 9118700);
addDataItem(series2,france, 46584000);
```

```
addDataItem(series2,germany, 72480869);
addDataItem(series2,netherlands, 11486000);
addDataItem(series2,sweden, 7480395);
addDataItem(series2,unitedKingdom, 52372000);

series3.setName("1970");
addDataItem(series3,belgium, 9637800);
addDataItem(series3,france, 51918000);
addDataItem(series3,germany, 77783164);
addDataItem(series3,netherlands, 13032335);
addDataItem(series3,sweden, 8042803);
addDataItem(series3,unitedKingdom, 55632000);
```

The last part of the method creates a `scene` instance. The three series are added to the `scene` and the `scene` is attached to the `stage` using the `setScene` method. A `stage` is a class that essentially represents the client area of a window:

```
Scene scene = new Scene(barChart, 800, 600);
barChart.getData().addAll(series1, series2, series3);
stage.setScene(scene);
stage.show();
```

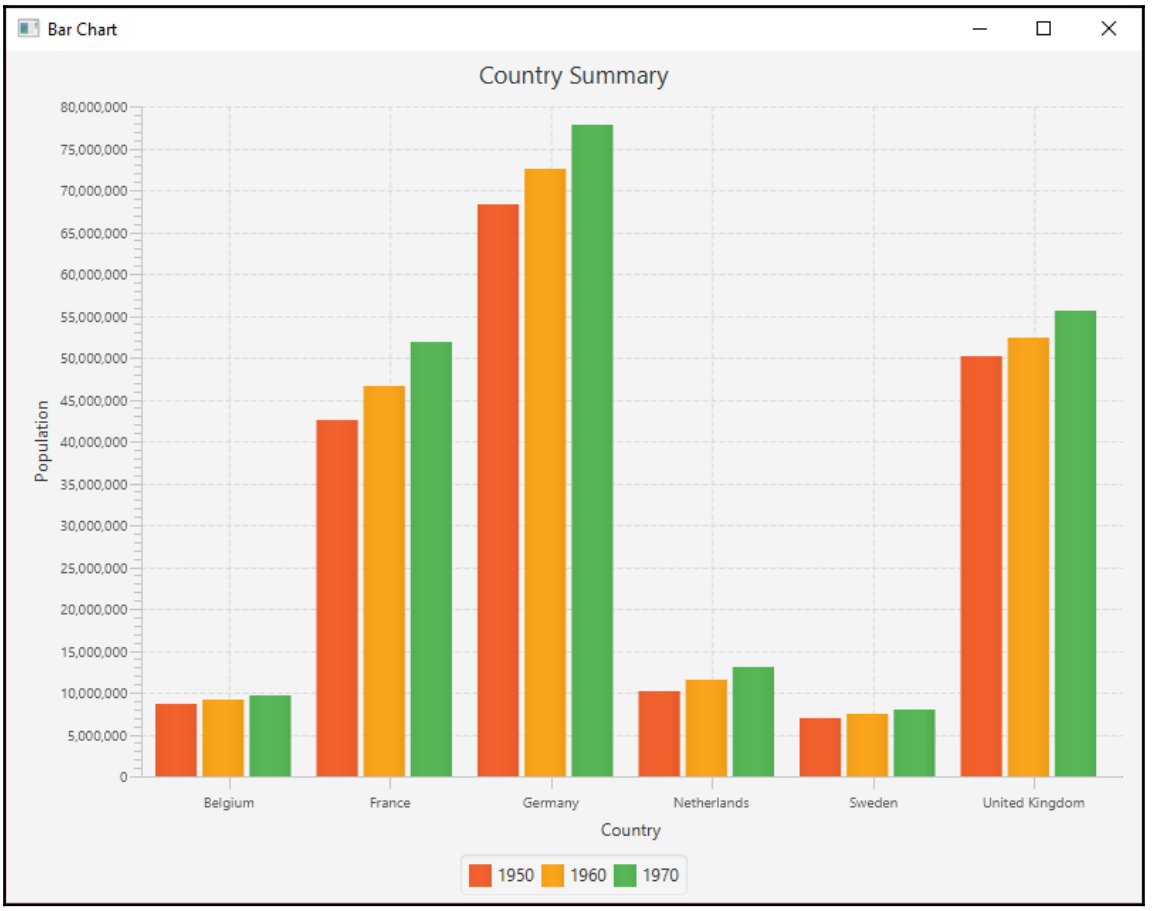
The last of the two methods is the `start` method, which is called automatically when the window is displayed. It is passed the `Stage` instance. Here, we call the `simpleBarChartByCountry` method:

```
public void start(Stage stage) {
    simpleBarChartByCountry(stage);
}
```

The `main` method consists of a call to the `Application` class's `launch` method:

```
public static void main(String[] args) {
    launch(args);
}
```

When the application is executed, the following graph is displayed:



Using decade as the category

In the following example, we will demonstrate how to display the same information, but we will organize the *X* axis categories by year. We will use the `simpleBarChartByYear` method, as shown next. The axis and titles are set up in the same way as before, but with different values for the title and labels:

```
public void simpleBarChartByYear(Stage stage) {
    stage.setTitle("Bar Chart");
    final BarChart<String, Number> barChart
            = new BarChart<>(xAxis, yAxis);
```

```
        barChart.setTitle("Year Summary");
        xAxis.setLabel("Year");
        yAxis.setLabel("Population");
        ...
    }
```

The following string variables are declared for the three decades:

```
String year1950 = "1950";
String year1960 = "1960";
String year1970 = "1970";
```

The data series are created in the same way as before, except the country name is used for the series name and the year is used for the category. In addition, six series are used, one for each country:

```
series1.setName(belgium);
addDataItem(series1, year1950, 8639369);
addDataItem(series1, year1960, 9118700);
addDataItem(series1, year1970, 9637800);

series2.setName(france);
addDataItem(series2, year1950, 42518000);
addDataItem(series2, year1960, 46584000);
addDataItem(series2, year1970, 51918000);

series3.setName(germany);
addDataItem(series3, year1950, 68374572);
addDataItem(series3, year1960, 72480869);
addDataItem(series3, year1970, 77783164);

series4.setName(netherlands);
addDataItem(series4, year1950, 10113527);
addDataItem(series4, year1960, 11486000);
addDataItem(series4, year1970, 13032335);

series5.setName(sweden);
addDataItem(series5, year1950, 7014005);
addDataItem(series5, year1960, 7480395);
addDataItem(series5, year1970, 8042803);

series6.setName(unitedKingdom);
addDataItem(series6, year1950, 50127000);
addDataItem(series6, year1960, 52372000);
addDataItem(series6, year1970, 55632000);
```

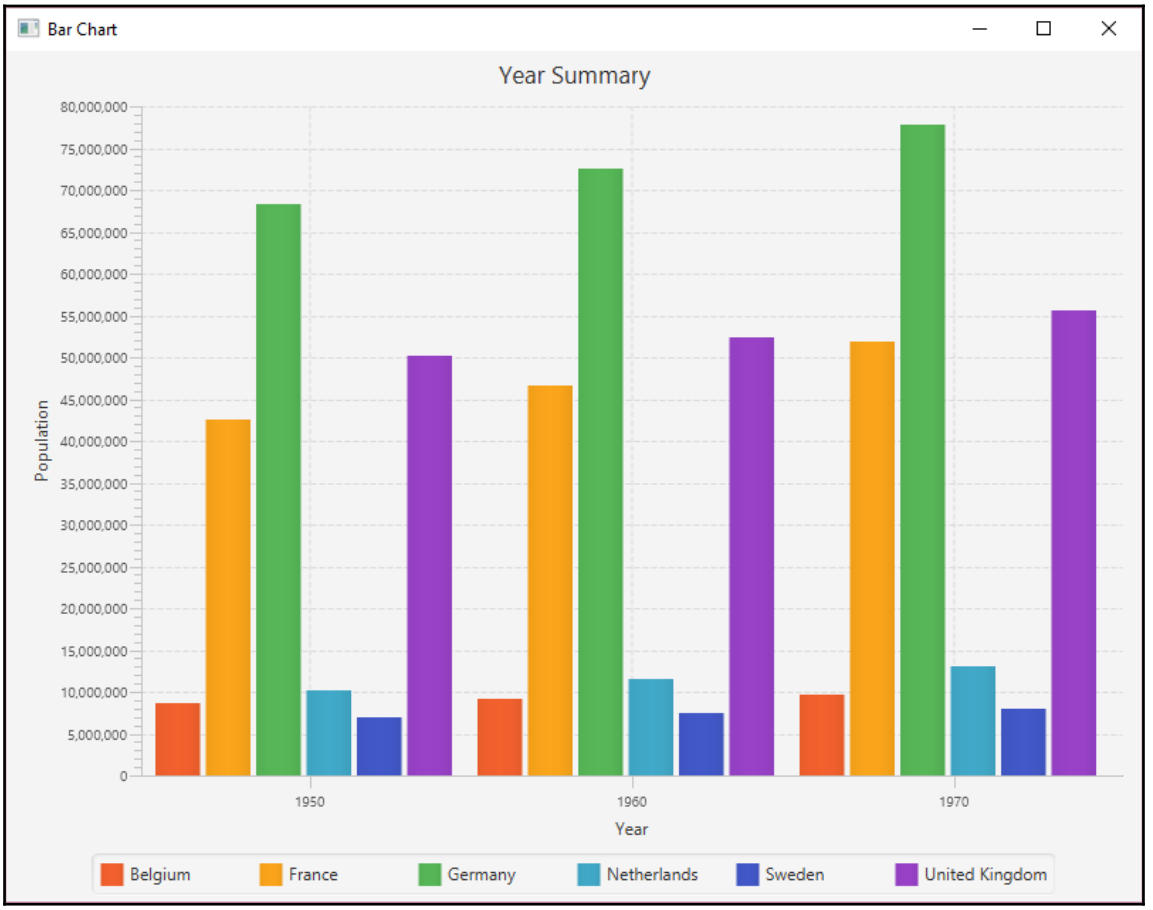
The `scene` is created and attached to the `stage`:

```
Scene scene = new Scene(barChart, 800, 600);
barChart.getData().addAll(series1, series2,
    series3, series4, series5, series6);
stage.setScene(scene);
stage.show();
```

The `main` method is unchanged, but the `start` method calls the `simpleBarChartByYear` method instead:

```
public void start(Stage stage) {
    simpleBarChartByYear(stage);
}
```

When the application is executed, the following graph is displayed:



Creating stacked graphs

An area chart depicts information by allocating more space for larger values. By stacking area charts on top of each other we create a stacked graph, sometimes called a stream graph. However, stacked graphs do not work well with negative values and cannot be used for data where summation does not make sense, such as with temperatures. If too many graphs are stacked, then it can become difficult to interpret.

Next, we will show how to create a stacked bar chart. The `stackedGraphExample` method contains the code to create the bar chart. We start with familiar code to set the title and labels. However, for the *X* axis, the `setCategories` method `FXCollections.<String>observableArrayList` instance is used to set the categories. The argument of this constructor is an array of strings created by the `Arrays` class's `asList` method and the names of the countries:

```
public void stackedGraphExample(Stage stage) {
    stage.setTitle("Stacked Bar Chart");
    final StackedBarChart<String, Number> stackedBarChart
            = new StackedBarChart<>(xAxis, yAxis);
    stackedBarChart.setTitle("Country Population");
    xAxis.setLabel("Country");
    xAxis.setCategories(
        FXCollections.<String>observableArrayList(
                Arrays.asList(belgium, germany, france,
                    netherlands, sweden, unitedKingdom)));
    yAxis.setLabel("Population");
    ...
}
```

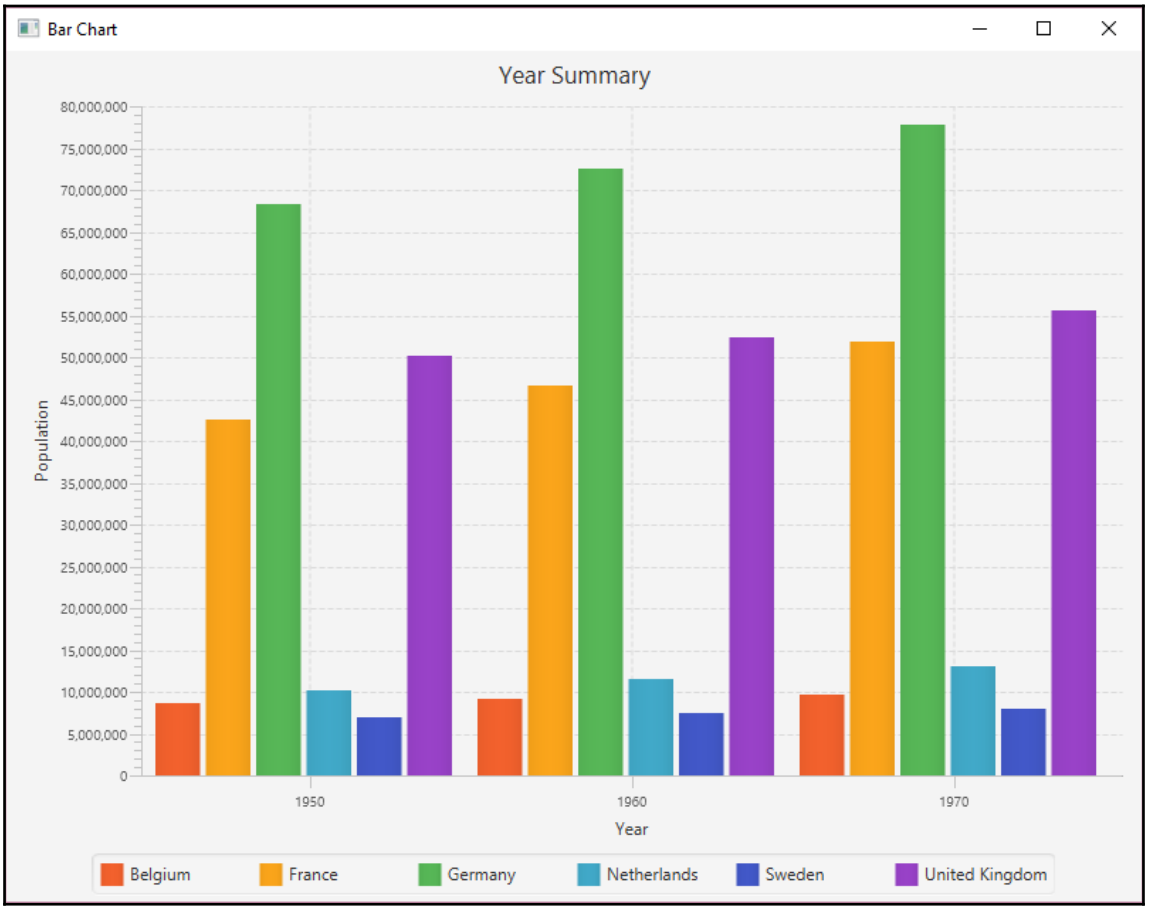
The series are initialized with the year being used for the series name and the country, and their population being added using the helper method `addDataItem`. The `scene` is then created:

```
series1.setName("1950");
addDataItem(series1, belgium, 8639369);
addDataItem(series1, france, 42518000);
addDataItem(series1, germany, 68374572);
addDataItem(series1, netherlands, 10113527);
addDataItem(series1, sweden, 7014005);
addDataItem(series1, unitedKingdom, 50127000);

series2.setName("1960");
addDataItem(series2, belgium, 9118700);
addDataItem(series2, france, 46584000);
addDataItem(series2, germany, 72480869);
addDataItem(series2, netherlands, 11486000);
addDataItem(series2, sweden, 7480395);
addDataItem(series2, unitedKingdom, 52372000);

series3.setName("1970");
addDataItem(series3, belgium, 9637800);
addDataItem(series3, france, 51918000);
addDataItem(series3, germany, 77783164);
addDataItem(series3, netherlands, 13032335);
addDataItem(series3, sweden, 8042803);
addDataItem(series3, unitedKingdom, 55632000);

Scene scene = new Scene(stackedBarChart, 800, 600);
stackedBarChart.getData().addAll(series1, series2, series3);
stage.setScene(scene);
stage.show();
```

The `main` method is unchanged, but the `start` method calls the `stackedGraphExample` method instead:

```
public void start(Stage stage) {
    stackedGraphExample(stage);
}
```

When the application is executed, the following graph is displayed:



Creating pie charts

The following pie chart example is based on the 2000 population of selected European countries as summarized here:

Country	Population	Percentage
Belgium	10,263,618	3
France	61,137,000	26
Germany	82,187,909	35

Netherlands	15,907,853	7
Sweden	8,872,000	4
United Kingdom	59,522,468	25

The JavaFX implementation uses the same `Application` base class and `main` method as used in the previous examples. We will not use a separate method for creating the GUI, but instead place this code in the `start` method, as shown here:

```
public class PieChartSample extends Application {

    public void start(Stage stage) {
        Scene scene = new Scene(new Group());
        stage.setTitle("Europian Country Population");
        stage.setWidth(500);
        stage.setHeight(500);
        ...
    }

    public static void main(String[] args) {
        launch(args);
    }

}
```

A pie chart is represented by the `PieChart` class. We can create and initialize the pie chart in the constructor by using an `ObservableList` of pie chart data. This data consists of a series of `PieChart.Data` instances, each containing a text label and a percentage value.

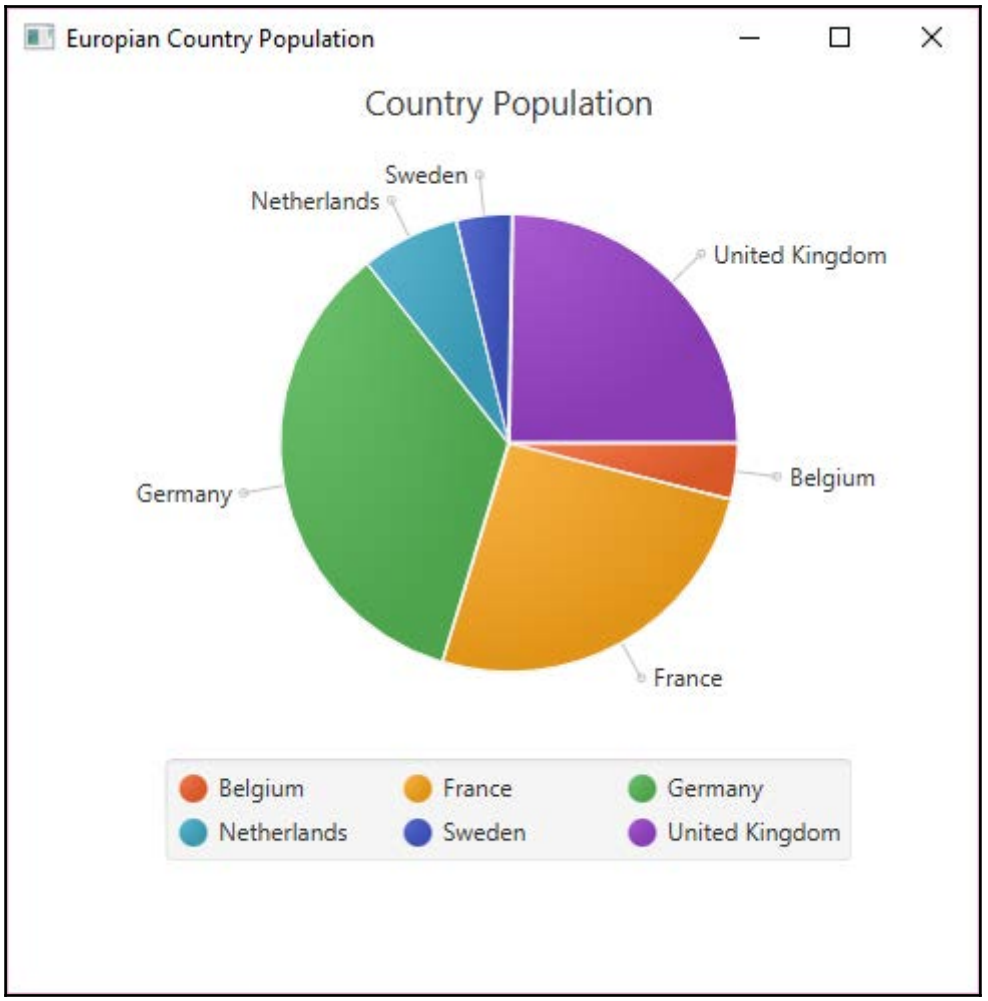
The next sequence creates an `ObservableList` instance based on the European population data presented earlier. The `FXCollections` class's `observableArrayList` method returns an `ObservableList` instance with a list of pie chart data:

```
ObservableList<PieChart.Data> pieChartData =
        FXCollections.observableArrayList(
        new PieChart.Data("Belgium", 3),
        new PieChart.Data("France", 26),
        new PieChart.Data("Germany", 35),
        new PieChart.Data("Netherlands", 7),
        new PieChart.Data("Sweden", 4),
        new PieChart.Data("United Kingdom", 25));
```

We then create the pie chart and set its title. The pie chart is then added to the `scene`, the `scene` is associated with the `stage`, and then the window is displayed:

```
final PieChart pieChart = new PieChart(pieChartData);
pieChart.setTitle("Country Population");
((Group) scene.getRoot()).getChildren().add(pieChart);
stage.setScene(scene);
stage.show();
```

When the application is executed, the following graph is displayed:



Creating scatter charts

Scatter charts also use the `XYChart.Series` class in JavaFX. For this example, we will use a set of European data that includes the previous Europeans countries and their population data for the decades 1500 through 2000. This information is stored in a file called `EuropeanScatterData.csv`. The first part of this file is shown here:

```
1500 1400000
1600 1600000
1650 1500000
1700 2000000
1750 2250000
1800 3250000
1820 3434000
1830 3750000
1840 4080000
...
```

We start with the declaration of the JavaFX `MainApp` class, as shown next. The `main` method launches the application and the `start` method creates the user interface:

```
public class MainApp extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        ...
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Within the `start` method we set the title, create the axes, and create an instance of the `ScatterChart` that represents the scatter plot. The `NumberAxis` class's constructors used values that better match the data range than the default values used by its default constructor:

```
stage.setTitle("Scatter Chart Sample");
final NumberAxis yAxis = new NumberAxis(1400, 2100, 100);
final NumberAxis xAxis = new NumberAxis(500000, 90000000,
    1000000);
final ScatterChart<Number, Number> scatterChart = new
    ScatterChart<>(xAxis, yAxis);
```

Next, the axes' labels are set along with the scatter chart's title:

```
xAxis.setLabel("Population");
yAxis.setLabel("Decade");
scatterChart.setTitle("Population Scatter Graph");
```

An instance of the `XYChart.Series` class is created and named:

```
XYChart.Series series = new XYChart.Series();
```

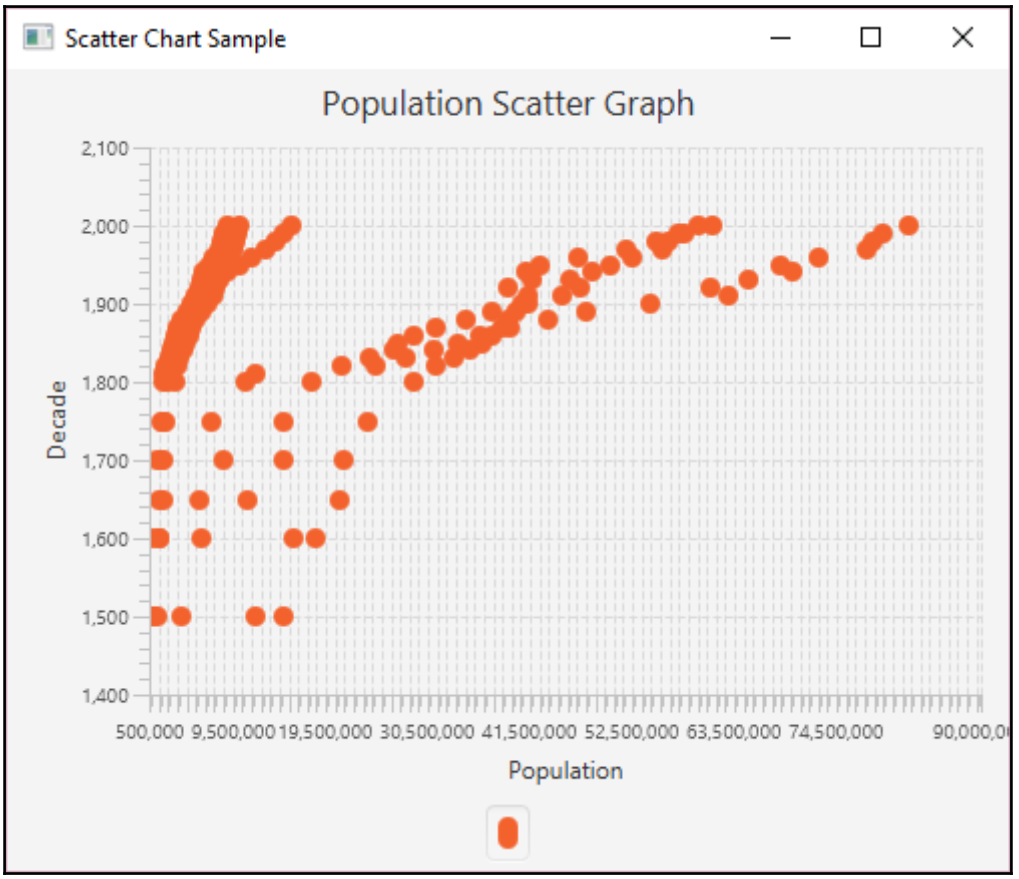
The series is populated using a `CSVReader` class instance and the file `EuropeanScatterData.csv`. This process was discussed in `Chapter 3`, *Data Cleaning*:

```
try (CSVReader dataReader = new CSVReader(new
FileReader("EuropeanScatterData.csv"), ',')) {
    String[] nextLine;
    while ((nextLine = dataReader.readNext()) != null) {
            int decade = Integer.parseInt(nextLine[0]);
            int population = Integer.parseInt(nextLine[1]);
            series.getData().add(new XYChart.Data(
                population, decade));
            out.println("Decade: " + decade +
                "  Population: " + population);
    }
}
scatterChart.getData().addAll(series);
```

The JavaFX `scene` and `stage` are created, and then the plot is displayed:

```
Scene scene = new Scene(scatterChart, 500, 400);
stage.setScene(scene);
stage.show();
```

When the application is executed, the following graph is displayed:



Creating histograms

Histograms, though similar in appearance to bar charts, are used to display the frequency of data items in relation to other items within the dataset. Each of the following examples using GRAL will use the `DataTable` class to initially hold the data to be displayed. In this example, we will read data from a sample file called `AgeofMarriage.csv`. This comma-separated file holds a list of ages at which people were first married.

We will create a new class, called `HistogramExample`, which extends the `JFrame` class and contains the following code within its constructor. We first create a `DataReader` object to specify that the data is in CSV format. We then use a try-catch block to handle IO exceptions and call the `DataReader` class's `read` method to place the data directly into a `DataTable` object. The first parameter of the `read` method is a `FileInputStream` object, and the second specifies the type of data expected from within the file:

```
DataReader readType=
  DataReaderFactory.getInstance().get("text/csv");
String fileName = "C://AgeofMarriage.csv";
try {
    DataTable histData = (DataTable) readType.read(
        New FileInputStream(fileName), Integer.class);
    ...
}
```

Next, we create a `Number` array to specify the ages for which we expect to have data. In this case, we expect the ages of marriage will range from `19` to `30`. We use this array to create our `Histogram` object. We include our `DataTable` from earlier and specify the orientation as well. Then we create our `DataSource`, specify our starting age, and specify the spacing along our *X* axis:

```
Number ageRange[] = {19,20,21,22,23,24,25,26,27,28,29,30};
Histogram sampleHisto = new Histogram1D(
    histData, Orientation.VERTICAL, ageRange);
DataSource sampleHistData = new EnumeratedData(sampleHisto, 19,
  1.0);
```

We use the `BarPlot` class to create our histogram from the data we read in earlier:

```
BarPlot testPlot = new BarPlot(sampleHistData);
```

The next few steps serve to format various aspects of our histogram. We use the `setInsets` method to specify how much space to place around each side of the graph within the window. We can provide a title for our graph and specify the bar width:

```
testPlot.setInsets(new Insets2D.Double(20.0, 50.0, 50.0, 20.0));
testPlot.getTitle().setText("Average Age of Marriage");
testPlot.setBarWidth(0.7);
```

We also need to format our *X* and *Y* axes. We have chosen to set our range for the *X* axis to closely match our expected age range but to provide some space on the side of the graph. Because we know the amount of sample data, we set our *Y* axis to range from `0` to `10`. In a business application, these ranges would be calculated by examining the actual dataset. We can also specify whether we want tick marks to show and where we would like the axes to intersect:

```
testPlot.getAxis(BarPlot.AXIS_X).setRange(18, 30.0);
testPlot.getAxisRenderer(BarPlot.AXIS_X).setTickAlignment(0.0);
testPlot.getAxisRenderer(BarPlot.AXIS_X).setTickSpacing(1);
testPlot.getAxisRenderer(BarPlot.AXIS_X).setMinorTicksVisible(false  );

testPlot.getAxis(BarPlot.AXIS_Y).setRange(0.0, 10.0);
testPlot.getAxisRenderer(BarPlot.AXIS_Y).setTickAlignment(0.0);
testPlot.getAxisRenderer(BarPlot.AXIS_Y).setMinorTicksVisible(false  );
testPlot.getAxisRenderer(BarPlot.AXIS_Y).setIntersection(0);
```

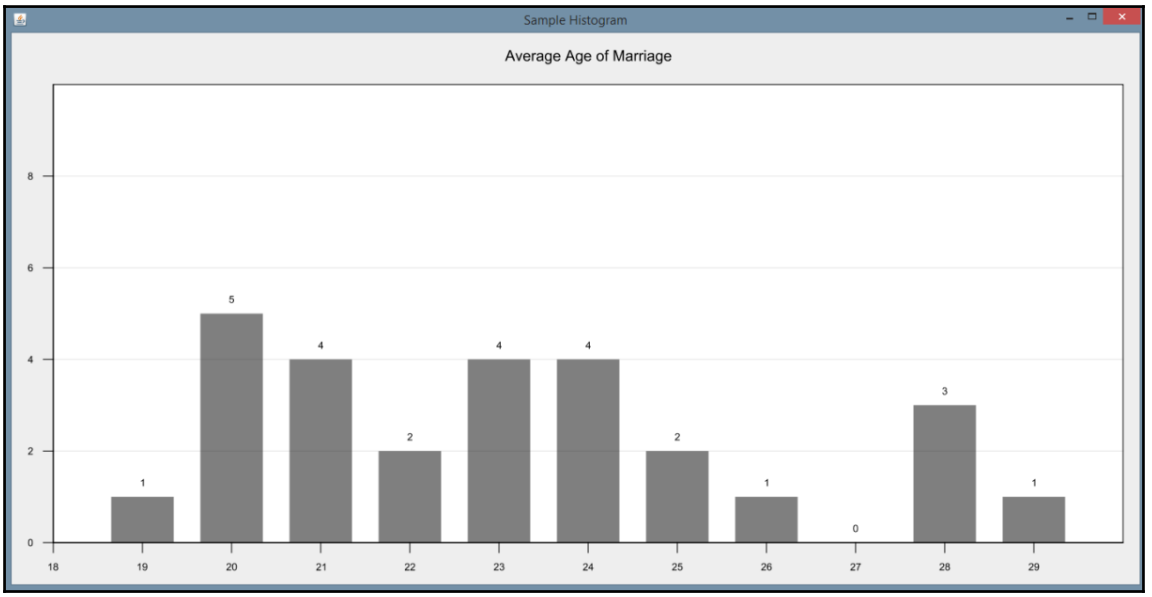
We also have a lot of flexibility with the color and values displayed on the graph. In this example, we have chosen to display the frequency value for each age and set our graph color to `black`:

```
PointRenderer renderHist =
    testPlot.getPointRenderers(sampleHistData).get(0);
renderHist.setColor(GraphicsUtils.deriveWithAlpha(Color.black,
  128));
renderHist.setValueVisible(true);
```

Finally, we set several properties for how we want our window to display:

```
InteractivePanel pan = new InteractivePanel(testPlot);
pan.setPannable(false);
pan.setZoomable(false);
add(pan);
setSize(1500, 700);
this.setVisible(true);
```

When the application is executed, the following graph is displayed:



Creating donut charts

Donut charts are similar to pie charts, but they are missing the middle section (hence the name donut). Some analysts prefer donut charts to pie charts because they do not emphasize the size of each piece within the chart and are easier to compare to other donut charts. They also provide the added advantage of taking up less space, allowing for more formatting options in the display.

In this example, we will assume our data is already populated in a two-dimensional array called `ageCount`. The first row of the array contains the possible age values, ranging again from `19` to `30` (inclusive). The second row contains the number of data values equal to each age. For example, in our dataset, there are six data values equal to `19`, so `ageCount[0][1]` contains the number six.

We create a `DataTable` and use the `add` method to add our values from the array. Notice we are testing to see if the value of a particular age is zero. In our test case, there will be zero data values equal to `23`. We are opting to add a blank space in our donut chart if there are no data values for that point. This is accomplished by using a negative number as the first parameter in the `add` method. This will set an empty space of size `3`:

```
DataTable donutData = new DataTable(Integer.class, Integer.class);
for(int Y = 0; Y < ageCount[0].length; y++){
    if(ageCount[1][y] == 0){
        donutData.add(-3, ageCount[0][y]);
    }else{
        donutData.add(ageCount[1][y], ageCount[0][y]);
    }
}
```

Next, we create our donut plot using the `PiePlot` class. We set basic properties of the plot, including specifying the values for the legend. In this case, we want our legend to reflect our age possibilities, so we use the `setLabelColumn` method to change the default labels. We also set our insets as we did in the previous example:

```
PiePlot testPlot = new PiePlot(donutData);
((ValueLegend) testPlot.getLegend()).setLabelColumn(1);
testPlot.getTitle().setText("Donut Plot Example");
testPlot.setRadius(0.9);
testPlot.setLegendVisible(true);
testPlot.setInsets(new Insets2D.Double(20.0, 20.0, 20.0, 20.0));
```

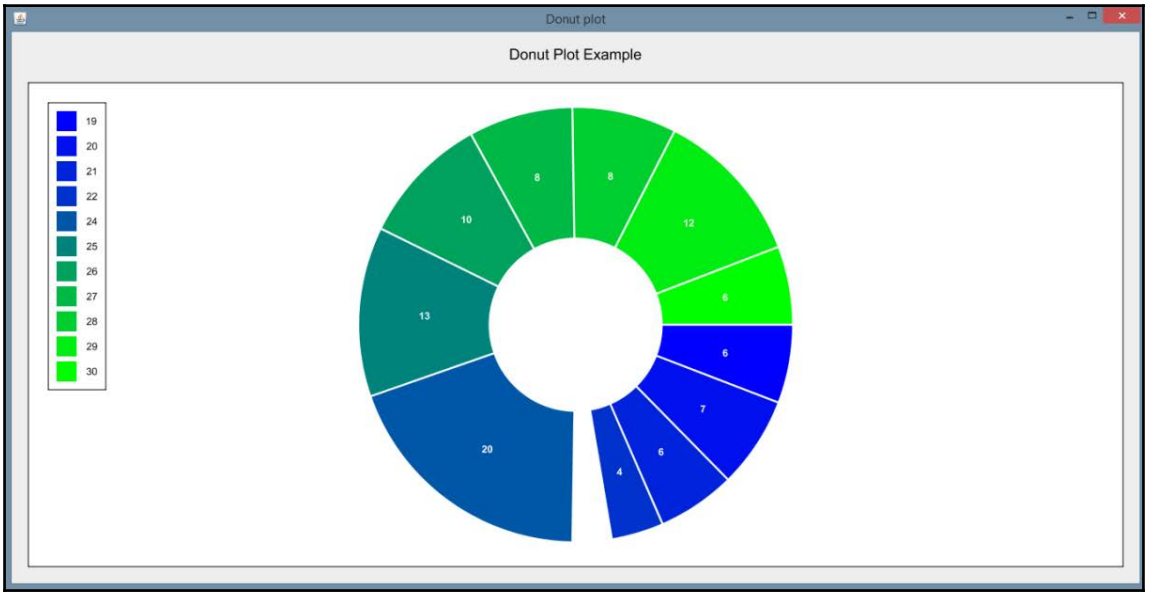
Next, we create a `PieSliceRenderer` object to set more advanced properties. Because a donut plot is basically a pie plot in essence, we will render a donut plot by calling the `setInnerRadius` method. We also specify the gap between the pie slices, the colors used, and the style of the labels:

```
PieSliceRenderer renderPie = (PieSliceRenderer)
testPlot.getPointRenderer(donutData);
renderPie.setInnerRadius(0.4);
renderPie.setGap(0.2);
LinearGradient colors = new LinearGradient(
    Color.blue, Color.green);
renderPie.setColor(colors);
renderPie.setValueVisible(true);
renderPie.setValueColor(Color.WHITE);
renderPie.setValueFont(Font.decode(null).deriveFont(Font.BOLD));
```

Finally, we create our panel and set its size:

```
add(new InteractivePanel(testPlot), BorderLayout.CENTER);
setSize(1500, 700);
setVisible(true);
```

When the application is executed, the following graph is displayed:



Creating bubble charts

Bubble charts are similar to scatter plots except they represent data with three dimensions. The first two dimensions are expressed on the *X* and *Y* axes and the third is represented by the size of the point plotted. This can be helpful in determining relationships between data values.

We will again use the `DataTable` class to initially hold the data to be displayed. In this example, we will read data from a sample file called `MarriageByYears.csv`. This is also a CSV file, and contains one column representing the year a marriage occurred, a second column holding the age at which a person was married, and a third column holding integers representing marital satisfaction on a scale from `1` (least satisfied) to `10` (most satisfied). We create a `DataSeries` to represent our type of desired data plot and then create a `XYPlot` object:

```
DataReader readType =
    DataReaderFactory.getInstance().get("text/csv");
String fileName = "C://MarriageByYears.csv";
try {
    DataTable bubbleData = (DataTable) readType.read(
        new FileInputStream(fileName), Integer.class,
            Integer.class, Integer.class);
DataSeries bubbleSeries = new DataSeries("Bubble", bubbleData);
XYPlot testPlot = new XYPlot(bubbleSeries);
```

Next, we set basic property information about our chart. We will set the color and turn off the vertical and horizontal grids in this example. We will also make our *X* and *Y* axes invisible in this example. Notice that we still set a range for the axes, even though they are not displayed:

```
testPlot.setInsets(new Insets2D.Double(30.0));  testPlot.setBackground(new
Color(0.75f, 0.75f, 0.75f));
XYPlotArea2D areaProp = (XYPlotArea2D) testPlot.getPlotArea();
areaProp.setBorderColor(null);
areaProp.setMajorGridX(false);
areaProp.setMajorGridY(false);
areaProp.setClippingArea(null);

testPlot.getAxisRenderer(XYPlot.AXIS_X).setShapeVisible(false);
testPlot.getAxisRenderer(XYPlot.AXIS_X).setTicksVisible(false);
testPlot.getAxisRenderer(XYPlot.AXIS_Y).setShapeVisible(false);
testPlot.getAxisRenderer(XYPlot.AXIS_Y).setTicksVisible(false);
testPlot.getAxis(XYPlot.AXIS_X).setRange(1940, 2020);
testPlot.getAxis(XYPlot.AXIS_Y).setRange(17, 30);
```

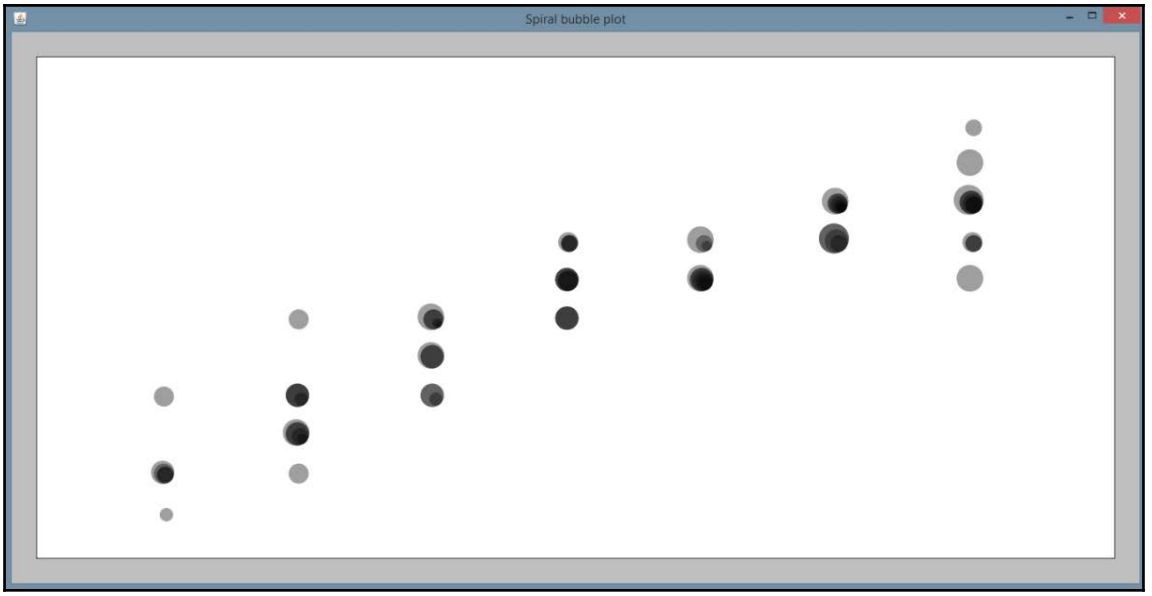
We can also set properties related to the bubbles drawn on the chart. Here, we set the color and shape, and specify which column of the data will be used to scale the shapes. In this case, the third column, with the marital satisfaction rating, will be used. We set it using the `setColumn` method:

```
Color color = GraphicsUtils.deriveWithAlpha(Color.black, 96);
SizeablePointRenderer renderBubble = new SizeablePointRenderer();
renderBubble.setShape(new Ellipse2D.Double(-3.5, -3.5, 4.0, 4.0));
renderBubble.setColor(color);
renderBubble.setColumn(2);
testPlot.setPointRenderers(bubbleSeries, renderBubble);
```

Finally, we create our panel and set its size:

```
add(new InteractivePanel(testPlot), BorderLayout.CENTER);
setSize(new Dimension(1500, 700));
setVisible(true);
```

When the application is executed, the following graph is displayed. Notice both the size and color of the points changes depending upon the frequency of that particular data point:



Summary

In this chapter, we introduce basic graphs, plots, and charts used to visualize data. The process of visualization enables an analyst to graphically examine the data under review. This is more intuitive, and often facilitates the rapid identification of anomalies in the data that can be hard to extract from the raw data.

Several visual representations were examined, including line charts, a variety of bar charts, pie charts, scatterplots, histograms, donut charts, and bubble charts. Each of these graphical depictions of data provides a different perspective of the data being analyzed. The most appropriate technique depends on the nature of the data being used. While we have not covered all of the possible graphical techniques, this sample provides a good overview of what is available.

We were also concerned with how Java is used to draw these graphics. Many of the examples used JavaFX. This is a readily available tool that is bundled with Java SE. However, there are several other libraries available. We used GRAL to illustrate how to generate some graphs.

With the overview of visualization techniques, we are ready to move on to other topics, where visualization will be used to better convey the essence of data science techniques. In the next chapter, we will introduce basic statistical processes, including linear regression, and we will use the techniques introduced in this chapter.

5
Statistical Data Analysis Techniques

The intent of this chapter is not to make the reader an expert in statistical techniques. Rather, it is to familiarize the reader with the basic statistical techniques in use and demonstrate how Java can support statistical analysis. While there are quite a variety of data analysis techniques, in this chapter, we will focus on the more common tasks.

These techniques range from the relatively simple mean calculation to sophisticated regression analysis models. Statistical analysis can be a very complicated process and requires significant study to be conducted properly. We will start with an introduction to basic statistical analysis techniques, including calculating the mean, median, mode, and standard deviation of a dataset. There are numerous approaches used to calculate these values, which we will demonstrate using standard Java and third-party APIs. We will also briefly discuss sample size and hypothesis testing.

Regression analysis is an important technique for analyzing data. The technique creates a line that tries to match the dataset. The equation representing the line can be used to predict future behavior. There are several types of regression analysis. In this chapter, we will focus on simple linear regression and multiple regression. With simple linear regression, a single factor such as age is used to predict some behavior such as the likelihood of eating out. With multiple regression, multiple factors such as age, income level, and marital status may be used to predict how often a person eats out.

Predictive analytics, or analysis, is concerned with predicting future events. Many of the techniques used in this book are concerned with making predictions. Specifically, the regression analysis part of this chapter predicts future behavior.

Before we see how Java supports regression analysis, we need to discuss basic statistical techniques. We begin with mean, mode, and median.

In this chapter, we will cover the following topics:

- Working with mean, mode, and median
- Standard deviation and sample size determination
- Hypothesis testing
- Regression analysis

Working with mean, mode, and median

The mean, median, and mode are basic ways to describe characteristics or summarize information from a dataset. When a new, large dataset is first encountered, it can be helpful to know basic information about it to direct further analysis. These values are often used in later analysis to generate more complex measurements and conclusions. This can occur when we use the mean of a dataset to calculate the standard deviation, which we will demonstrate in the *Standard deviation* section of this chapter.

Calculating the mean

The term **mean,** also called the average, is computed by adding values in a list and then dividing the sum by the number of values. This technique is useful for determining the general trend for a set of numbers. It can also be used to fill in missing data elements. We are going to examine several ways to calculate the mean for a given set of data using standard Java libraries as well as third-party APIs.

Using simple Java techniques to find mean

In our first example, we will demonstrate a basic way to calculate mean using standard Java capabilities. We will use an array of `double` values called `testData`:

```
double[] testData = {12.5, 18.7, 11.2, 19.0, 22.1, 14.3, 16.9, 12.5,
   17.8, 16.9};
```

We create a `double` variable to hold the sum of all of the values and a `double` variable to hold the `mean`. A loop is used to iterate through the data and add values together. Next, the sum is divided by the `length` of our array (the total number of elements) to calculate the `mean`:

```
double total = 0;
for (double element : testData) {
```

```
    total += element;
}
double mean = total / testData.length;
out.println("The mean is " + mean);
```

Our output is as follows:

The mean is 16.19

Using Java 8 techniques to find mean

Java 8 provided additional capabilities with the introduction of optional classes. We are going to use the `OptionalDouble` class in conjunction with the `Arrays` class's `stream` method in this example. We will use the same array of doubles we used in the previous example to create an `OptionalDouble` object. If any of the numbers in the array, or the sum of the numbers in the array, is not a real number, the value of the `OptionalDouble` object will also not be a real number:

```
OptionalDouble mean = Arrays.stream(testData).average();
```

We use the `isPresent` method to determine whether we calculated a valid number for our mean. If we do not get a good result, the `isPresent` method will return `false` and we can handle any exceptions:

```
if (mean.isPresent()) {
    out.println("The mean is " + mean.getAsDouble());
} else {
    out.println("The stream was empty");
}
```

Our output is the following:

```
The mean is 16.19
```

Another, more succinct, technique using the `OptionalDouble` class involves lambda expressions and the `ifPresent` method. This method executes its argument if `mean` is a valid `OptionalDouble` object:

```
OptionalDouble mean = Arrays.stream(testData).average();
mean.ifPresent(x-> out.println("The mean is " + x));
```

Our output is as follows:

```
The mean is 16.19
```

Finally, we can use the `orElse` method to either print the mean or an alternate value if `mean` is not a valid `OptionalDouble` object:

```
OptionalDouble mean = Arrays.stream(testData).average();
out.println("The mean is " + mean.orElse(0));
```

Our output is the same:

```
The mean is 16.19
```

For our next two mean examples, we will use third-party libraries and continue using the array of doubles, `testData`.

Using Google Guava to find mean

In this example, we will use Google Guava libraries, introduced in `Chapter 3`, *Data Cleaning*. The `Stats` class provides functionalities for handling numeric data, including finding mean and standard deviation, which we will demonstrate later. To calculate the `mean`, we first create a `Stats` object using our `testData` array and then execute the `mean` method:

```
Stats testStat = Stats.of(testData);
double mean = testStat.mean();
out.println("The mean is " + mean);
```

Notice the difference between the default format of the output in this example.

Using Apache Commons to find mean

In our final mean examples, we use Apache Commons libraries, also introduced in `Chapter 3`, *Data Cleaning*. We first create a `Mean` object and then execute the `evaluate` method using our `testData`. This method returns a `double`, representing the mean of the values in the array:

```
Mean mean = new Mean();
double average = mean.evaluate(testData);
out.println("The mean is " + average);
```

Our output is the following:

```
The mean is 16.19
```

Apache Commons also provides a helpful `DescriptiveStatistics` class. We will use this later to demonstrate median and standard deviation, but first we will begin by calculating the mean. Using the `SynchronizedDescriptiveStatistics` class is advantageous as it is synchronized and therefore thread safe.

We start by creating our `DescriptiveStatistics` object, `statTest`. We then loop through our double array and add each item to `statTest`. We can then invoke the `getMean` method to calculate the `mean`:

```
DescriptiveStatistics statTest =
    new SynchronizedDescriptiveStatistics();
for(double num : testData){
   statTest.addValue(num);
}
out.println("The mean is " + statTest.getMean());
```

Our output is as follows:

```
The mean is 16.19
```

Next, we will cover the related topic: median.

Calculating the median

The mean can be misleading if the dataset contains a large number of outlying values or is otherwise skewed. When this happens, the mode and median can be useful. The term **median** is the value in the middle of a range of values. For an odd number of values, this is easy to compute. For an even number of values, the median is calculated as the average of the middle two values.

Using simple Java techniques to find median

In our first example, we will use a basic Java approach to calculate the median. For these examples, we have modified our `testData` array slightly:

```
double[] testData = {12.5, 18.3, 11.2, 19.0, 22.1, 14.3, 16.2, 12.5,
   17.8, 16.5};
```

First, we use the `Arrays` class to sort our data because finding the median is simplified when the data is in numeric order:

```
Arrays.sort(testData);
```

We then handle three possibilities:

- Our list is empty
- Our list has an even number of values
- Our list has an odd number of values

The following code could be shortened, but we have been explicit to help clarify the process. If our list has an even number of values, we divide the length of the list by `2`. The first variable, `mid1`, will hold the first of two middle values. The second variable, `mid2`, will hold the second middle value. The average of these two numbers is our median value. The process for finding the median index of a list with an odd number of values is simpler and requires only that we divide the length by `2` and add `1`:

```
if(testData.length==0){    // Empty list
   out.println("No median. Length is 0");
}else if(testData.length%2==0){    // Even number of elements
   double mid1 = testData[(testData.length/2)-1];
   double mid2 = testData[testData.length/2];
   double med = (mid1 + mid2)/2;
   out.println("The median is " + med);
}else{   // Odd number of elements
   double mid = testData[(testData.length/2)+1];
   out.println("The median is " + mid);
}
```

Using the preceding array, which contains an even number of values, our output is:

```
The median is 16.35
```

To test our code for an odd number of elements, we will add the double `12.5` to the end of the array. Our new output is as follows:

```
The median is 16.5
```

Using Apache Commons to find the median

We can also calculate the median using the Apache Commons `DescriptiveStatistics` class demonstrated in the *Calculating the mean* section. We will continue using the `testData` array with the following values:

```
double[] testData = {12.5, 18.3, 11.2, 19.0, 22.1, 14.3, 16.2, 12.5,
   17.8, 16.5, 12.5};
```

Our code is very similar to what we used to calculate the mean. We simply create our `DescriptiveStatistics` object and call the `getPercentile` method, which returns an estimate of the value stored at the percentile specified in its argument. To find the median, we use the value of `50`:

```
DescriptiveStatistics statTest =
    new SynchronizedDescriptiveStatistics();
for(double num : testData){
   statTest.addValue(num);
}
out.println("The median is " + statTest.getPercentile(50));
```

Our output is as follows:

```
The median is 16.2
```

Calculating the mode

The term **mode** is used for the most frequently occurring value in a dataset. This can be thought of as the most popular result, or the highest bar in a histogram. It can be a useful piece of information when conducting statistical analysis but it can be more complicated to calculate than it first appears. To begin, we will demonstrate a simple Java technique using the following `testData` array:

```
double[] testData = {12.5, 18.3, 11.2, 19.0, 22.1, 14.3, 16.2, 12.5,
   17.8, 16.5, 12.5};
```

We start off by initializing variables to hold the mode, the number of times the mode appears in the list, and a `tempCnt` variable. The mode and `modeCount` variables are used to hold the mode value and the number of times this value occurs in the list respectively. The variable `tempCnt` is used to count the number of times an element occurs in the list:

```
int modeCount = 0;
double mode = 0;
int tempCnt = 0;
```

We then use nested for loops to compare each value of the array to the other values within the array. When we find matching values, we increment our `tempCnt`. After comparing each value, we test to see whether `tempCnt` is greater than `modeCount`, and if so, we change our `modeCount` and mode to reflect the new values:

```
for (double testValue : testData){
   tempCnt = 0;
   for (double value : testData){
         if (testValue == value){
               tempCnt++;
         }
   }

   if (tempCnt > modeCount){
         modeCount = tempCnt;
         mode = testValue;
   }
}
out.println("Mode" + mode + " appears " + modeCount + " times.");
```

Using this example, our output is as follows:

```
The mode is 12.5 and appears 3 times.
```

While our preceding example seems straightforward, it poses potential problems. Modify the `testData` array as shown here, where the last entry is changed to `11.2`:

```
double[] testData = {12.5, 18.3, 11.2, 19.0, 22.1, 14.3, 16.2, 12.5,
   17.8, 16.5, 11.2};
```

When we execute our code this time, our output is as follows:

```
The mode is 12.5 and appears 2 times.
```

The problem is that our `testData` array now contains two values that appear two times each, `12.5` and `11.2`. This is known as a multimodal set of data. We can address this through basic Java code and through third-party libraries, as we will show in a moment.

However, first we will show two approaches using simple Java. The first approach will use two `ArrayList` instances and the second will use an `ArrayList` and a `HashMap` instance.

Using ArrayLists to find multiple modes

In the first approach, we modify the code used in the last example to use an `ArrayList` class. We will create two `ArrayLists`, one to hold the unique numbers within the dataset and one to hold the count of each number. We also need a `tempMode` variable, which we use next:

```
ArrayList<Integer> modeCount = new ArrayList<Integer>();
ArrayList<Double> mode = new ArrayList<Double>();
int tempMode = 0;
```

Next, we will loop through the array and test for each value in our mode list. If the value is not found in the list, we add it to `mode` and set the same position in `modeCount` to `1`. If the value is found, we increment the same position in `modeCount` by `1`:

```
for (double testValue : testData){
   int loc = mode.indexOf(testValue);
   if(loc == -1){
         mode.add(testValue);
         modeCount.add(1);
   }else{
         modeCount.set(loc, modeCount.get(loc)+1);
   }
}
```

Next, we loop through our `modeCount` list to find the largest value. This represents the mode, or the frequency of the most common value in the dataset. This allows us to select multiple modes:

```
for(int cnt = 0; cnt < modeCount.size(); cnt++){
   if (tempMode < modeCount.get(cnt)){
         tempMode = modeCount.get(cnt);
   }
}
```

Finally, we loop through our `modeCount` array again and print out any elements in mode that correspond to elements in `modeCount` containing the largest value, or mode:

```
for(int cnt = 0; cnt < modeCount.size(); cnt++){
   if (tempMode == modeCount.get(cnt)){
         out.println(mode.get(cnt) + " is a mode and appears " +
             modeCount.get(cnt) + " times.");
   }
}
```

When our code is executed, our output reflects our multimodal dataset:

```
12.5 is a mode and appears 2 times.
11.2 is a mode and appears 2 times.
```

Using a HashMap to find multiple modes

The second approach uses `HashMap`. First, we create `ArrayList` to hold possible modes, as in the previous example. We also create our `HashMap` and a variable to hold the mode:

```
ArrayList<Double> modes = new ArrayList<Double>();
HashMap<Double, Integer> modeMap = new HashMap<Double, Integer>();
int maxMode = 0;
```

Next, we loop through our `testData` array and count the number of occurrences of each value in the array. We then add the count of each value and the value itself to the `HashMap`. If the count for the value is larger than our `maxMode` variable, we set `maxMode` to our new largest number:

```
for (double value : testData) {
   int modeCnt = 0;
   if (modeMap.containsKey(value)) {
         modeCnt = modeMap.get(value) + 1;
   } else {
         modeCnt = 1;
   }
   modeMap.put(value, modeCnt);
   if (modeCnt > maxMode) {
         maxMode = modeCnt;
   }
}
```

Finally, we loop through our `HashMap` and retrieve our modes, or all values with a count equal to our `maxMode`:

```
for (Map.Entry<Double, Integer> multiModes : modeMap.entrySet()) {
   if (multiModes.getValue() == maxMode) {
         modes.add(multiModes.getKey());
   }
}
for(double mode : modes){
   out.println(mode + " is a mode and appears " + maxMode + " times.");
}
```

When we execute our code, we get the same output as in the previous example:

```
12.5 is a mode and appears 2 times.
11.2 is a mode and appears 2 times.
```

Using a Apache Commons to find multiple modes

Another option uses the Apache Commons `StatUtils` class. This class contains several methods for statistical analysis, including multiple methods for the mean, but we will only examine the mode here. The method is named `mode` and takes an array of doubles as its parameter. It returns an array of doubles containing all modes of the dataset:

```
double[] modes = StatUtils.mode(testData);
for(double mode : modes){
   out.println(mode + " is a mode.");
}
```

One disadvantage is that we are not able to count the number of times our mode appears within this method. We simply know what the mode is, not how many times it appears. When we execute our code, we get a similar output to our previous example:

```
12.5 is a mode.
11.2 is a mode.
```

Standard deviation

Standard deviation is a measurement of how values are spread around the mean. A high deviation means that there is a wide spread, whereas a low deviation means that the values are more tightly grouped around the mean. This measurement can be misleading if there is not a single focus point or there are numerous outliers.

We begin by showing a simple example using basic Java techniques. We are using our testData array from previous examples, duplicated here:

```
double[] testData = {12.5, 18.3, 11.2, 19.0, 22.1, 14.3, 16.2, 12.5,
   17.8, 16.5, 11.2};
```

Before we can calculate the standard deviation, we need to find the average. We could use any of our techniques listed in the *Calculating the mean* section, but we will add up our values and divide by the length of `testData` for simplicity's sake:

```
int sum = 0;
for(double value : testData){
   sum += value;
}
double mean = sum/testData.length;
```

Next, we create a variable, `sdSum`, to help us calculate the standard deviation. As we loop through our array, we subtract the mean from each data value, square that value, and add it to `sdSum`. Finally, we divide `sdSum` by the length of the array and square that result:

```
int sdSum = 0;
for (double value : testData){
    sdSum += Math.pow((value - mean), 2);
}
out.println("The standard deviation is " +
Math.sqrt( sdSum / ( testData.length ) ));
```

Our output is our standard deviation:

```
The standard deviation is 3.3166247903554
```

Our next technique uses Google Guava's `Stats` class to calculate the standard deviation. We start by creating a `Stats` object with our `testData`. We then call the `populationStandardDeviation` method:

```
Stats testStats = Stats.of(testData);
double sd = testStats.populationStandardDeviation();
out.println("The standard deviation is " + sd);
```

The output is as follows:

```
The standard deviation is 3.3943803826056653
```

This example calculates the standard deviation of an entire population. Sometimes it is preferable to calculate the standard deviation of a sample subset of a population, to correct possible bias. To accomplish this, we use essentially the same code as before but replace the `populationStandardDeviation` method with `sampleStandardDeviation`:

```
Stats testStats = Stats.of(testData);
double sd = testStats.sampleStandardDeviation();
out.println("The standard deviation is " + sd);
```

In this case, our output is:

```
The sample standard deviation is 3.560056179332006
```

Our next example uses the Apache Commons `DescriptiveStatistics` class, which we used to calculate the mean and median in previous examples. Remember, this technique has the advantage of being thread safe and synchronized. After we create a `SynchronizedDescriptiveStatistics` object, we add each value from the array. We then call the `getStandardDeviation` method.

```
DescriptiveStatistics statTest =
    new SynchronizedDescriptiveStatistics();
for(double num : testData){
   statTest.addValue(num);
}
out.println("The standard deviation is " +
statTest.getStandardDeviation());
```

Notice the output matches our output from our previous example. The `getStandardDeviation` method by default returns the standard deviation adjusted for a sample:

```
The standard deviation is 3.5600561793320065
```

We can, however, continue using Apache Commons to calculate the standard deviation in either form. The `StandardDeviation` class allows you to calculate the population standard deviation or subset standard deviation. To demonstrate the differences, replace the previous code example with the following:

```
StandardDeviation sdSubset = new StandardDeviation(false);
out.println("The population standard deviation is " +
sdSubset.evaluate(testData));

StandardDeviation sdPopulation = new StandardDeviation(true);
out.println("The sample standard deviation is " +
sdPopulation.evaluate(testData));
```

On the first line, we created a new `StandardDeviation` object and set our constructor's parameter to `false`, which will produce the standard deviation of a population. The second section uses a value of `true`, which produces the standard deviation of a sample. In our example, we used the same test dataset. This means we were first treating it as though it were a subset of a population of data. In our second example we assumed that our dataset was the entire population of data. In reality, you would might not use the same set of data with each of those methods. The output is as follows:

```
The population standard deviation is 3.3943803826056653
The sample standard deviation is 3.560056179332006
```

The preferred option will depend upon your sample and particular analyzation needs.

Sample size determination

Sample size determination involves identifying the quantity of data required to conduct accurate statistical analysis. When working with large datasets it is not always necessary to use the entire set. We use sample size determination to ensure we choose a sample small enough to manipulate and analyze easily, but large enough to represent our population of data accurately.

It is not uncommon to use a subset of data to train a model and another subset is used to test the model. This can be helpful for verifying accuracy and reliability of data. Some common consequences for a poorly determined sample size include false-positive results, false-negative results, identifying statistical significance where none exists, or suggesting a lack of significance where it is actually present. Many tools exist online for determining appropriate sample sizes, each with varying levels of complexity. One simple example is available at `https://www.surveymonkey.com/mp/sample-size-calculator/`.

Hypothesis testing

Hypothesis testing is used to test whether certain assumptions, or premises, about a dataset could not happen by chance. If this is the case, then the results of the test are considered to be statistically significant.

Performing hypothesis testing is not a simple task. There are many different pitfalls to avoid such as the placebo effect or the observer effect. In the former, a participant will attain a result that they think is expected. In the observer effect, also called the **Hawthorne effect**, the results are skewed because the participants know they are being watched. Due to the complex nature of human behavior analysis, some types of statistical analysis are particularly subject to skewing or corruption.

The specific methods for performing hypothesis testing are outside the scope of this book and require a solid background in statistical processes and best practices. Apache Commons provides a package, `org.apache.commons.math3.stat.inference`, with tools for performing hypothesis testing. This includes tools to perform a student's T-test, chi square, and calculating p values.

Regression analysis

Regression analysis is useful for determining trends in data. It indicates the relationship between dependent and independent variables. The independent variables determine the value of a dependent variable. Each independent variable can have either a strong or a weak effect on the value of the dependent variable. Linear regression uses a line in a scatterplot to show the trend. Non-linear regression uses some sort of curve to depict the relationships.

For example, there is a relationship between blood pressure and various factors such as age, salt intake, and **Body Mass Index** (**BMI**). The blood pressure can be treated as the dependent variable and the other factors as independent variables. Given a dataset containing these factors for a group of individuals we can perform regression analysis to see trends.

There are several types of regression analysis supported by Java. We will be examining simple linear regression and multiple linear regression. Both approaches take a dataset and derive a linear equation that best fits the data. Simple linear regression uses a single dependent and a single independent variable. Multiple linear regression uses multiple dependent variables.

There are several APIs that support simple linear regression including:

- **Apache Commons** - `http://commons.apache.org/proper/commons-math/javadocs/api-3.6.1/index.html`
- **Weka** - `http://weka.sourceforge.net/doc.dev/weka/core/matrix/LinearRegression.html`

- **JFree** - `http://www.jfree.org/jfreechart/api/javadoc/org/jfree/data/statistics/Regression.html`
- **Michael Thomas Flanagan's Java Scientific Library** - `http://www.ee.ucl.ac.uk/~mflanaga/java/Regression.html`

Nonlinear Java support can be found at:

- **odinsbane/least-squares-in-java** - `https://github.com/odinsbane/least-squares-in-java`
- **NonLinearLeastSquares** (**Parallel Java Library Documentation**) - `https://www.cs.rit.edu/~ark/pj/doc/edu/rit/numeric/NonLinearLeastSquares.html`

There are several statistics that evaluate the effectiveness of an analysis. We will focus on basic statistics.

Residuals are the difference between the actual data values and the predicted values. The **Residual Sum of Squares** (**RSS**) is the sum of the squares of residuals. Essentially it measures the discrepancy between the data and a regression model. A small RSS indicates the model closely matches the data. RSS is also known as the **Sum of Squared Residuals** (**SSR**) or the **Sum of Squared Errors** (**SSE**) of prediction.

The **Mean Square Error** (**MSE**) is the sum of squared residuals divided by the degrees of freedom. The number of degrees of freedom is the number of independent observations (*N*) minus the number of estimates of population parameters. For simple linear regression this *N - 2* because there are two parameters. For multiple linear regression it depends on the number of independent variables used.

A small MSE also indicates that the model fits the dataset well. You will see both of these statistics used when discussing linear regression models.

The correlation coefficient measures the association between two variables of a regression model. The correlation coefficient ranges from *-1* to *+1*. A value of *+1* means that two variables are perfectly related. When one increases, so does the other. A correlation coefficient of *-1* means that two variables are negatively related. When one increases, the other decreases. A value of 0 means there is no correlation between the variables. The coefficient is frequently designated as R. It will often be squared, thus ignoring the sign of the relation. The Pearson's product moment correlation coefficient is normally used.

Using simple linear regression

Simple linear regression uses a least squares approach where a line is computed that minimizes the sum of squared of the distances between the points and the line. Sometimes the line is calculated without using the *Y* intercept term. The regression line is an estimate. We can use the line's equation to predict other data points. This is useful when we want to predict future events based on past performance.

In the following example we use the Apache Commons SimpleRegression class with the Belgium population dataset used in `Chapter 4`, *Data Visualization*. The data is duplicated here for your convenience:

Decade	Population
1950	8639369
1960	9118700
1970	9637800
1980	9846800
1990	9969310
2000	10263618

While the application that we will demonstrate is a JavaFX application, we will focus on the linear regression aspects of the application. We used a JavaFX program to generate a chart to show the regression results.

The body of the `start` method follows. The input data is stored in a two-dimension array as shown here:

```
double[][] input = {{1950, 8639369}, {1960, 9118700},
    {1970, 9637800}, {1980, 9846800}, {1990, 9969310},
    {2000, 10263618}};
```

An instance of the `SimpleRegression` class is created and the data is added using the `addData` method:

```
SimpleRegression regression = new SimpleRegression();
regression.addData(input);
```

We will use the model to predict behavior for several years as declared in the array that follows:

```
double[] predictionYears = {1950, 1960, 1970, 1980, 1990, 2000,
    2010, 2020, 2030, 2040};
```

We will also format our output using the following `NumberFormat` instances. One is used for the year where the `setGroupingUsed` method with a parameter of false suppresses commas.

```
NumberFormat yearFormat = NumberFormat.getNumberInstance();
yearFormat.setMaximumFractionDigits(0);
yearFormat.setGroupingUsed(false);
NumberFormat populationFormat = NumberFormat.getNumberInstance();
populationFormat.setMaximumFractionDigits(0);
```

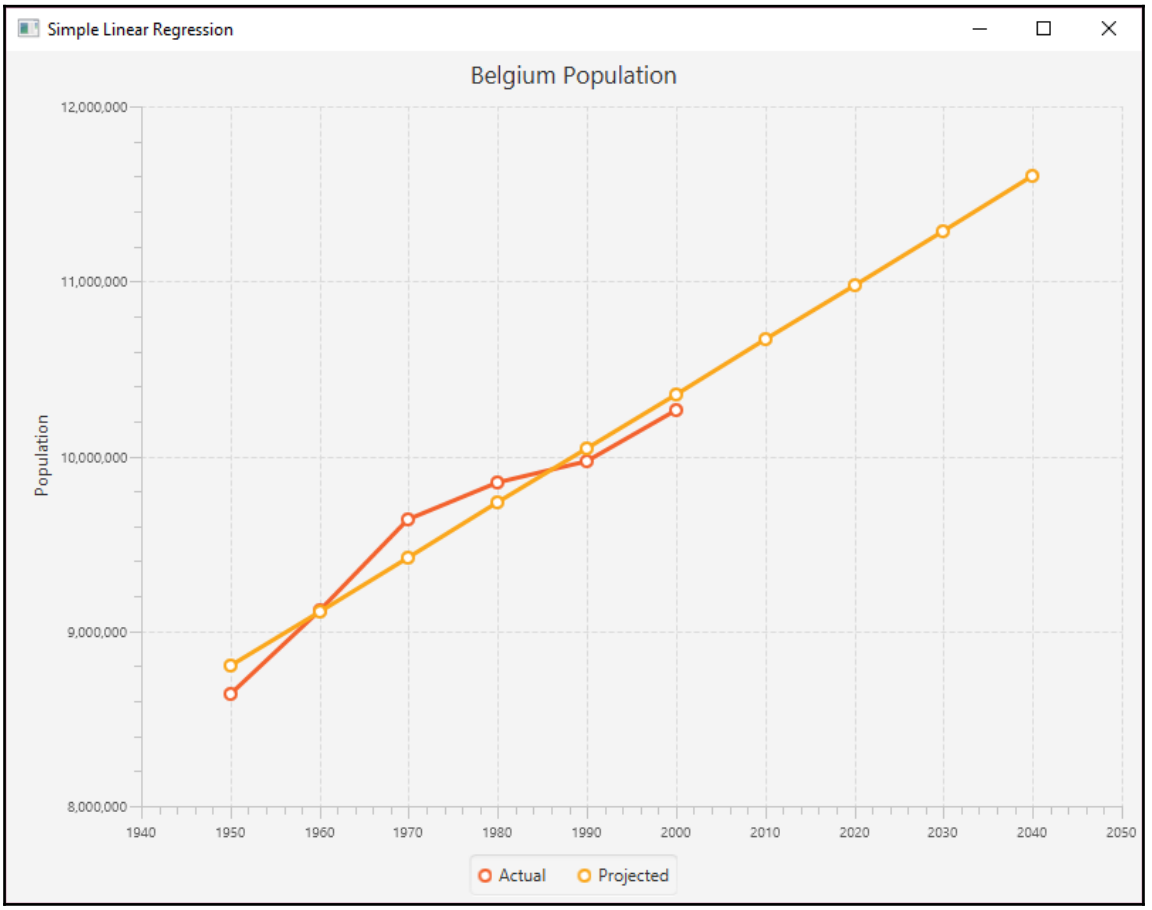
The `SimpleRegression` class possesses a `predict` method that is passed a value, a year in this case, and returns the estimated population. We use this method in a loop and call the method for each year:

```
for (int i = 0; i < predictionYears.length; i++) {
    out.println(nf.format(predictionYears[i]) + "-"
            + nf.format(regression.predict(predictionYears[i])));
}
```

When the program is executed, we get the following output:

```
1950-8,801,975
1960-9,112,892
1970-9,423,808
1980-9,734,724
1990-10,045,641
2000-10,356,557
2010-10,667,474
2020-10,978,390
2030-11,289,307
2040-11,600,223
```

To see the results graphically, we generated the following index chart. The line matches the actual population values fairly well and shows the projected populations in the future.



Simple Linear Regression

The `SimpleRegession` class supports a number of methods that provide additional information about the regression. These methods are summarized next:

Method	**Meaning**
`getR`	Returns Pearson's product moment correlation coefficient
`getRSquare`	Returns the coefficient of determination (R-square)
`getMeanSquareError`	Returns the MSE

`getSlope`	The slope of the line
`getIntercept`	The intercept

We used the helper method, `displayAttribute`, to display various attribute values as shown here:

```
displayAttribute(String attribute, double value) {
    NumberFormat numberFormat = NumberFormat.getNumberInstance();
    numberFormat.setMaximumFractionDigits(2);
    out.println(attribute + ": " + numberFormat.format(value));
}
```

We called the previous methods for our model as shown next:

```
displayAttribute("Slope",regression.getSlope());
displayAttribute("Intercept", regression.getIntercept());
displayAttribute("MeanSquareError",
    regression.getMeanSquareError());
displayAttribute("R", + regression.getR());
displayAttribute("RSquare", regression.getRSquare());
```

The output follows:

```
Slope: 31,091.64
Intercept: -51,826,728.48
MeanSquareError: 24,823,028,973.4
R: 0.97
RSquare: 0.94
```

As you can see, the model fits the data nicely.

Using multiple regression

Our intent is not to provide a detailed explanation of multiple linear regression as that would be beyond the scope of this section. A more through treatment can be found at `http://www.biddle.com/documents/bcg_comp_chapter4.pdf`. Instead, we will explain the basics of the approach and show how we can use Java to perform multiple regression.

Multiple regression works with data where multiple independent variables exist. This happens quite often. Consider that the fuel efficiency of a car can be dependent on the octane level of the gas being used, the size of the engine, the average cruising speed, and the ambient temperature. All of these factors can influence the fuel efficiency, some to a larger degree than others.

The independent variable is normally represented as Y where there are multiple dependent variables represented using different *X*s. A simplified equation for a regression using three dependent variables follows where each variable has a coefficient. The first term is the intercept. These coefficients are not intended to represent real values but are only used for illustrative purposes.

$Y = 11 + 0.75\ X1 + 0.25\ X2 - 2\ X3$

The intercept and coefficients are generated using a multiple regression model based on sample data. Once we have these values, we can create an equation to predict other values.

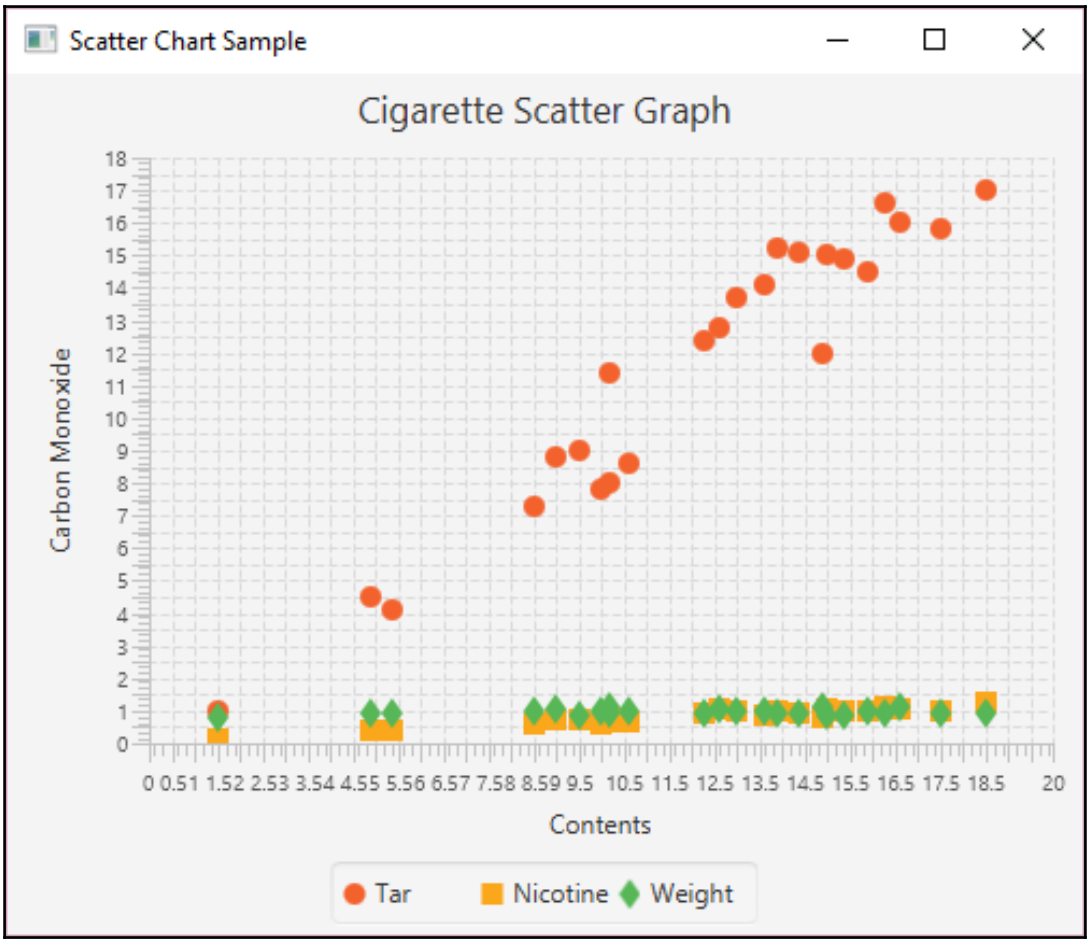
We will use the Apache Commons `OLSMultipleLinearRegression` class to perform multiple regression using cigarette data. The data has been adapted from `http://www.amstat.org/publications/jse/v2n1/datasets.mcintyre.html`. The data consists of 25 entries for different brands of cigarettes with the following information:

- Brand name
- Tar content (mg)
- Nicotine content (mg)
- Weight (g)
- Carbon monoxide content (mg)

The data has been stored in a file called `data.csv` as shown in the following partial listing of its contents where the columns values match the order of the previous list:

```
Alpine,14.1,.86,.9853,13.6
Benson&Hedges,16.0,1.06,1.0938,16.6
BullDurham,29.8,2.03,1.1650,23.5
CamelLights,8.0,.67,.9280,10.2
...
```

The following is a scatter plot chart showing the relationship of the data:



Multiple Regression Scatter plot

We will use a JavaFX program to create the scatter plot and to perform the analysis. We start with the `MainApp` class as shown next. In this example we will focus on the multiple regression code and we do not include the JavaFX code used to create the scatter plot. The complete program can be downloaded from `http://www.packtpub.com/support`.

The data is held in a one-dimensional array and a `NumberFormat` instance will be used to format the values. The array size reflects the *25* entries and the *4* values per entry. We will not be using the brand name in this example.

```
    public class MainApp extends Application {
    private final double[] data = new double[100];
    private final NumberFormat numberFormat =
        NumberFormat.getNumberInstance();
    ...
    public static void main(String[] args) {
        launch(args);
    }

}
```

The data is read into the array using a `CSVReader` instance as shown next:

```
        int i = 0;
try (CSVReader dataReader = new CSVReader(
        new FileReader("data.csv"), ',')) {
    String[] nextLine;
    while ((nextLine = dataReader.readNext()) != null) {
        String brandName = nextLine[0];
        double tarContent = Double.parseDouble(nextLine[1]);
        double nicotineContent = Double.parseDouble(nextLine[2]);
        double weight = Double.parseDouble(nextLine[3]);
        double carbonMonoxideContent =
            Double.parseDouble(nextLine[4]);

        data[i++] = carbonMonoxideContent;
        data[i++] = tarContent;
        data[i++] = nicotineContent;
        data[i++] = weight;
        ...
    }
}
```

Apache Commons possesses two classes that perform multiple regression:

- `OLSMultipleLinearRegression`- **Ordinary Least Square (OLS)** regression
- `GLSMultipleLinearRegression`- **Generalized Least Squared (GLS)** regression

When the latter technique is used, the correlation within elements of the model impacts the results addversely. We will use the `OLSMultipleLinearRegression` class and start with its instantiation:

```
OLSMultipleLinearRegression ols = new OLSMultipleLinearRegression();
```

We will use the `newSampleData` method to initialize the model. This method needs the number of observations in the dataset and the number of independent variables. It may throw an `IllegalArgumentException` exception which needs to be handled.

```
    int numberOfObservations = 25;
int numberOfIndependentVariables = 3;
try {
    ols.newSampleData(data, numberOfObservations,
        numberOfIndependentVariables);
} catch (IllegalArgumentException e) {
     // Handle exceptions
}
```

Next, we set the number of digits that will follow the decimal point to two and invoke the `estimateRegressionParameters` method. This returns an array of values for our equation, which are then displayed:

```
    numberFormat.setMaximumFractionDigits(2);
double[] parameters = ols.estimateRegressionParameters();
for (int i = 0; i < parameters.length; i++) {
    out.println("Parameter " + i +": " +
        numberFormat.format(parameters[i]));
}
```

When executed we will get the following output which gives us the parameters needed for our regression equation:

```
Parameter 0: 3.2
Parameter 1: 0.96
Parameter 2: -2.63
Parameter 3: -0.13
```

To predict a new dependent value based on a set of independent variables, the `getY` method is declared, as shown next. The `parameters` parameter contains the generated equation coefficients. The `arguments` parameter contains the value for the dependent variables. These are used to calculate the new dependent value which is returned:

```
    public double getY(double[] parameters, double[] arguments) {
    double result = 0;
    for(int i=0; i<parameters.length; i++) {
        result += parameters[i] * arguments[i];
    }
    return result;
}
```

We can test this method by creating a series of independent values. Here we used the same values as used for the `SalemUltra` entry in the data file:

```
    double arguments1[] = {1, 4.5, 0.42, 0.9106};
out.println("X: " + 4.9 + "  y: " +
        numberFormat.format(getY(parameters,arguments1)));
```

This will give us the following values:

```
X: 4.9 y: 6.31
```

The return value of `6.31` is different from the actual value of `4.9`. However, using the values for `VirginiaSlims`:

```
    double arguments2[] = {1, 15.2, 1.02, 0.9496};
out.println("X: " + 13.9 + "  y: " +
        numberFormat.format(getY(parameters,arguments2)));
```

We get the following result:

```
X: 13.9 y: 15.03
```

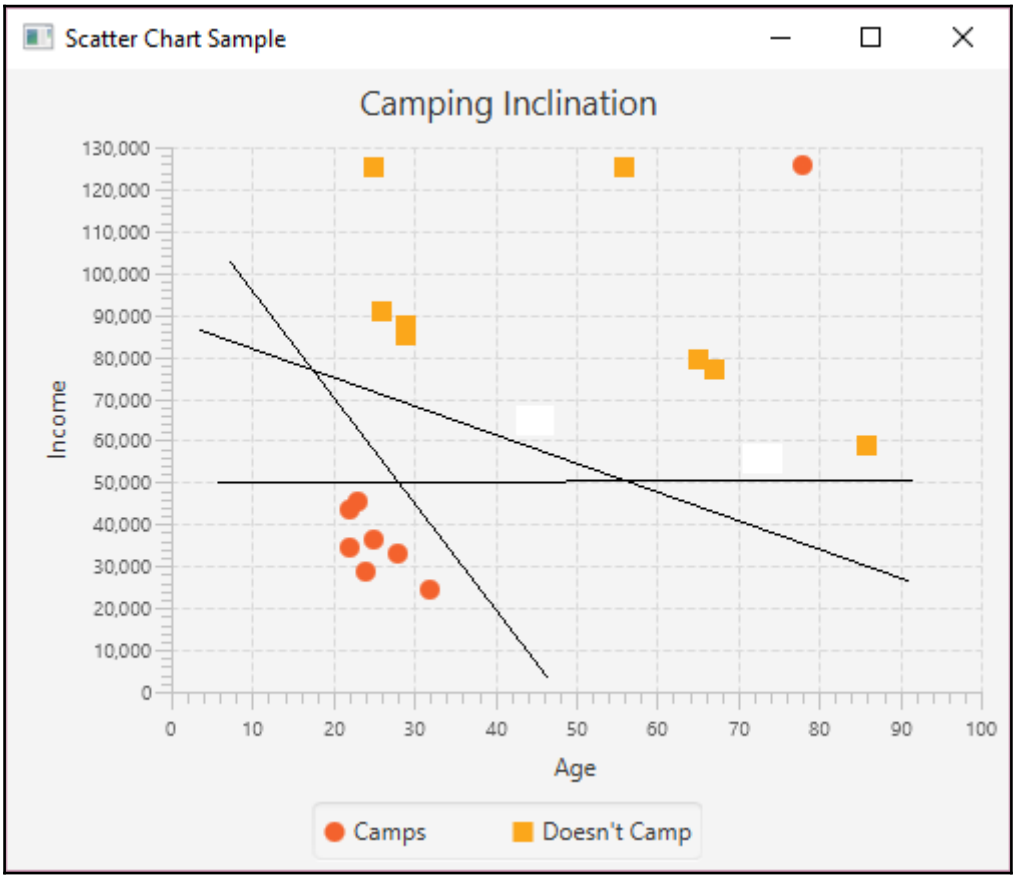
This is close to the actual value of `13.9`. Next, we use a different set of values than found in the dataset:

```
        double arguments3[] = {1, 12.2, 1.65, 0.86};
    out.println("X: " + 9.9 + "  y: " +
            numberFormat.format(getY(parameters,arguments3)));
```

The result follows:

```
X: 9.9 y: 10.49
```

The values differ but are still close. The following figure shows the predicted data in relation to the original data:



Multiple regression projected

The `OLSMultipleLinearRegression` class also possesses several methods to evaluate how well the models fits the data. However, due to the complex nature of multiple regression, we have not discussed them here.

Summary

In this chapter, we provided a brief introduction to the basic statistical analysis techniques you may encounter in data science applications. We started with simple techniques to calculate the mean, median, and mode of a set of numerical data. Both standard Java and third-party Java APIs were used to show how these attributes are calculated. While these techniques are relatively simple, there are some issues that need to be considered when calculating them.

Next, we examined linear regression. This technique is more predictive in nature and attempts to calculate other values in the future or past based on a sample dataset. We examine both simple linear regression and multiple regression and used Apache Commons classes to perform the regression and JavaFX to draw graphs.

Simple linear regression uses a single independent variable to predict a dependent variable. Multiple regression uses more than one independent variable. Both of these techniques have statistical attributes used to assess how well they match the data.

We demonstrated the use of the Apache Commons `OLSMultipleLinearRegression` class to perform multiple regression using cigarette data. We were able to use multiple attributes to create an equation that predicts the carbon monoxide output.

With these statistical techniques behind us, we can now examine basic machine learning techniques in the next chapter. This will include a detailed discussion of multilayer perceptrons and various other neural networks.

6
Machine Learning

Machine learning is a broad topic with many different supporting algorithms. It is generally concerned with developing techniques that allow applications to learn without having to be explicitly programmed to solve a problem. Typically, a model is built to solve a class of problems and then is trained using sample data from the problem domain. In this chapter, we will address a few of the more common problems and models used in data science.

Many of these techniques use training data to teach a model. The data consists of various representative elements of the problem space. Once the model has been trained, it is tested and evaluated using testing data. The model is then used with input data to make predictions.

For example, the purchases made by customers of a store can be used to train a model. Subsequently, predictions can be made about customers with similar characteristics. Due to the ability to predict customer behavior, it is possible to offer special deals or services to entice customers to return or facilitate their visit.

There are several ways of classifying machine learning techniques. One approach is to classify them according to the learning style:

- **Supervised learning**: With supervised learning, the model is trained with data that matches input characteristic values to the correct output values
- **Unsupervised learning**: In unsupervised learning, the data does not contain results, but the model is expected to determine relationships on its own.
- **Semi-supervised**: This technique uses a small amount of labeled data containing the correct response with a larger amount of unlabeled data. The combination can lead to improved results.
- **Reinforcement learning**: This is similar to supervised learning but a reward is provided for good results.

- **Deep learning**: This approach models high-level abstractions using a graph that contains multiple processing levels.

In this chapter, we will only be able to touch on a few of these techniques. Specifically, we will illustrate three techniques that use supervised learning:

- **Decision trees**: A tree is constructed using the features of the problem as internal nodes and the results as leaves
- **Support vector machines**: Generally used for classification by creating a hyperplane that separates the dataset and then making predictions
- **Bayesian networks**: Models used to depict probabilistic relationships between events within an environment

For unsupervised learning, we will show how **association rule learning** can be used to find relationships between elements of a dataset. However, we will not address unsupervised learning in this chapter.

We will discuss the elements of reinforcement learning and discuss a few specific variations of this technique. We will also provide links to resources for further exploration.

The discussion of deep learning is postponed to `Chapter 8`, *Deep Learning*. This technique builds upon neural networks, which will be discussed in `Chapter 7`, *Neural Networks*.

In this chapter, we will cover the following specific topics:

- Decision trees
- Support vector machines
- Bayesian networks
- Association rule learning
- Reinforcement learning

Supervised learning techniques

There are a large number of supervised machine learning algorithms available. We will examine three of them: decision trees, support vector machines, and Bayesian networks. They all use annotated datasets that contain attributes and a correct response. Typically, a training and a testing dataset is used.

We start with a discussion of decision trees.

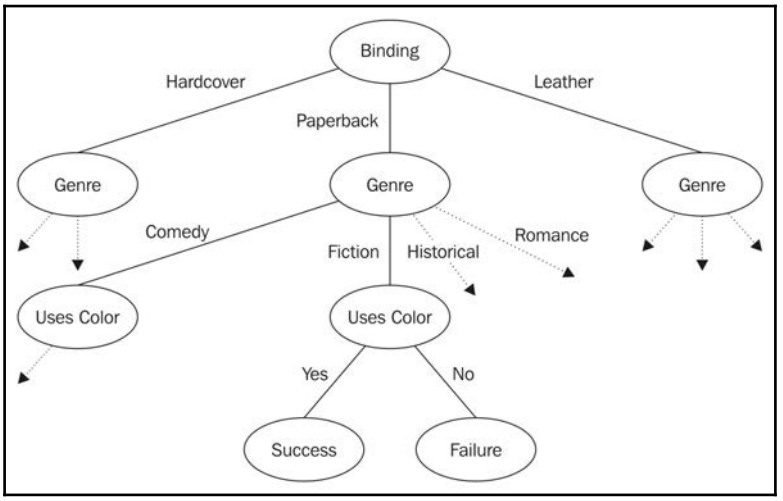
Decision trees

A machine learning **decision tree** is a model used to make predictions. It effectively maps certain observations to conclusions about a target. The term **tree** comes from the branches that reflect different states or values. The leaves of a tree represent results and the branches represent features that lead to the results. In data mining, a decision tree is a description of data used for classification. For example, we can use a decision tree to determine whether an individual is likely to buy an item based on certain attributes such as income level and postal code.

We want to create a decision tree that predicts results based on other variables. When the target variable takes on continuous values such as real numbers, the tree is called a **regression tree**.

A tree consists of internal nodes and leaves. Each internal node represents a feature of the model such as the number of years of education or whether a book is paperback or hardcover. The edges leading out of an internal node represent the values of these features. Each leaf is called a **class** and has an **associated probability distribution**.

For example, we will be using a dataset that deals with the success of a book based on its binding type, use of color, and genre. One possible decision tree based on this dataset follows:



Decision tree

Decision trees are useful and easy to understand. Preparing data for a model is straightforward even for large datasets.

Decision tree types

A tree can be *taught* by dividing an input dataset by the features. This is often done in a recursive fashion and is called **recursive partitioning** or **Top-Down Induction of Decision Trees** (**TDIDT**). The recursion is bounded when node's values are all of the same type as the target or the recursion no longer adds value.

Classification and Regression Tree (**CART**) analysis refers to two different types of decision tree types:

- **Classification tree analysis**: The leaf corresponds to a target feature
- **Regression tree analysis**: The leaf possesses a real number representing a feature

During the process of analysis, multiple trees may be created. There are several techniques used to create trees. The techniques are called **ensemble methods**:

- **Bagging decision trees**: The data is resampled and frequently used to obtain a prediction based on consensus
- **Random forest classifier**: Used to improve the classification rate
- **Boosted trees**: This can be used for regression or classification problems
- **Rotation forest**: Uses a technique called **Principal Component Analysis** (**PCA**)

With a given set of data, it is possible that more than one tree models the data. For example, the root of a tree may specify whether a bank has an ATM machine and a subsequent internal node may specify the number of tellers. However, the tree could be created where the number of tellers is at the root and the existence of an ATM is an internal node. The difference in the structure of the tree can determine how efficient the tree is.

There are a number of ways of determining the order of the nodes of a tree. One technique is to select an attribute that provides the most information gain; that is, choose an attribute that better helps narrow down the possible decisions fastest.

Decision tree libraries

There are several Java libraries that support decision trees:

- **Weka**: `http://www.cs.waikato.ac.nz/ml/weka/`

- **Apache Spark**: `https://spark.apache.org/docs/1.2.0/mllib-decision-tree.html`
- **JBoost**: `http://jboost.sourceforge.net`
- **MAchine Learning for LanguagE Toolkit** (**MALLET**): `http://mallet.cs.umass.edu`

We will use **Waikato Environment for Knowledge Analysis** (**Weka**) to demonstrate how to create a decision tree in Java. Weka is a tool that has a GUI interface that permits analysis of data. It can also be invoked from the command line or through a Java API, which we will use.

As a tree is being built, a variable is selected to split the tree. There are several techniques used to select a variable. The one we use is determined by how much information is gained by choosing a variable. Specifically, we will use the **C4.5 algorithm** as supported by Weka's `J48` class.

Weka uses an `.arff` file to hold a dataset. This file is human readable and consists of two sections. The first is a header section; it describes the data in the file. This section uses the ampersand to specify the relation and attributes of the data. The second section is the data section; it consists of a comma-delimited set of data.

Using a decision tree with a book dataset

For this example, we will use a file called `books.arff`. It is shown next and uses four features called attributes. The features specify how a book is bound, whether it uses multiple colors, its genre, and a result indicating whether the book was purchased or not. The header section is shown here:

```
@RELATION book_purchases
@ATTRIBUTE Binding {Hardcover, Paperback, Leather}
@ATTRIBUTE Multicolor {yes, no}
@ATTRIBUTE Genre {fiction, comedy, romance, historical}
@ATTRIBUTE Result {Success, Failure}
```

The data section follows and consists of 13 book entries:

```
@DATA
Hardcover,yes,fiction,Success
Hardcover,no,comedy,Failure
Hardcover,yes,comedy,Success
Leather,no,comedy,Success
Leather,yes,historical,Success
Paperback,yes,fiction,Failure
Paperback,yes,romance,Failure
```

```
Leather,yes,comedy,Failure
Paperback,no,fiction,Failure
Paperback,yes,historical,Failure
Hardcover,yes,historical,Success
Paperback,yes,comedy,Success
Hardcover,yes,comedy,Success
```

We will use the `BookDecisionTree` class as defined next to process this file. It uses one constructor and three methods:

- `BookDecisionTree`: Reads in the trainer data and creates an `Instance` object used to process the data
- `main`: Drives the application
- `performTraining`: Trains the model using the dataset
- `getTestInstance`: Creates a test case

The `Instances` class holds elements representing the individual dataset elements:

```
public class BookDecisionTree {
    private Instances trainingData;

    public static void main(String[] args) {
        ...
    }

    public BookDecisionTree(String fileName) {
        ...
    }

    private J48 performTraining() {
        ...
    }

    private Instance getTestInstance(
        ...
    }
}
```

The constructor opens a file and uses the `BufferReader` instance to create an instance of the `Instances` class. Each element of the dataset will be either a feature or a result. The `setClassIndex` method specifies the index of the result class. In this case, it is the last index of the dataset and corresponds to success or failure:

```
public BookDecisionTree(String fileName) {
    try {
        BufferedReader reader = new BufferedReader(
            new FileReader(fileName));
        trainingData = new Instances(reader);
        trainingData.setClassIndex(
            trainingData.numAttributes() - 1);
    } catch (IOException ex) {
        // Handle exceptions
    }
}
```

We will use the `J48` class to generate a decision tree. This class uses the C4.5 decision tree algorithm for generating a pruned or unpruned tree. The `setOptions` method specifies that an unpruned tree be used. The `buildClassifier` method actually creates the classifier based on the dataset used:

```
private J48 performTraining() {
    J48 j48 = new J48();
    String[] options = {"-U"};
    try {
        j48.setOptions(options);
        j48.buildClassifier(trainingData);
    } catch (Exception ex) {
        ex.printStackTrace();
    }
    return j48;
}
```

We will want to test the model, so we will create an object that implements the `Instance` interface for each test case. A `getTestInstance` helper method is passed three arguments representing the three features of a data element. The `DenseInstance` class is a class that implements the `Instance` interface. The values passed are assigned to the instance and the instance is returned:

```
private Instance getTestInstance(
        String binding, String multicolor, String genre) {
    Instance instance = new DenseInstance(3);
    instance.setDataset(trainingData);
    instance.setValue(trainingData.attribute(0), binding);
    instance.setValue(trainingData.attribute(1), multicolor);
    instance.setValue(trainingData.attribute(2), genre);
    return instance;
}
```

The `main` method uses all the previous methods to process and test our book dataset. First, a `BookDecisionTree` instance is created using the name of the book dataset file:

```
public static void main(String[] args) {
    try {
        BookDecisionTree decisionTree =
            new BookDecisionTree("books.arff");
        ...
    } catch (Exception ex) {
        // Handle exceptions
    }
}
```

Next, the `performTraining` method is invoked to train the model. We also display the tree:

```
J48 tree = decisionTree.performTraining();
System.out.println(tree.toString());
```

When executed, the following will be displayed:

```
J48 unpruned tree
------------------
Binding = Hardcover: Success (5.0/1.0)
Binding = Paperback: Failure (5.0/1.0)
Binding = Leather: Success (3.0/1.0)
Number of Leaves : 3
Size of the tree : 4
```

Testing the book decision tree

We will test the model with two different test cases. Both use identical code to set up the instance. We use the `getTestInstance` method with test-case-specific values and then use the instance with `classifyInstance` to get results. To get something that is more readable, we generate a string, which is then displayed:

```
Instance testInstance = decisionTree.
        getTestInstance("Leather", "yes", "historical");
int result = (int) tree.classifyInstance(testInstance);
String results = decisionTree.trainingData.attribute(3).value(result);
System.out.println(
        "Test with: " + testInstance + "  Result: " + results);

testInstance = decisionTree.
        getTestInstance("Paperback", "no", "historical");
result = (int) tree.classifyInstance(testInstance);
results = decisionTree.trainingData.attribute(3).value(result);
System.out.println(
        "Test with: " + testInstance + "  Result: " + results);
```

The result of executing this code is as follows:

```
Test with: Leather,yes,historical Result: Success
Test with: Paperback,no,historical Result: Failure
```

This matches our expectations. This technique is based on the amount of information gain before and after an ordering decision has been made. This can be measured based on the entropy as calculated here:

```
Entropy = -portionPos * log2(portionPos) - portionNeg* log2(portionNeg)
```

In this example, `portionPos` is the portion of the data that is positive and `portionNeg` is the portion of the data that is negative. Based on the books file, we can calculate the entropy for the binding as shown in the following table. The information gain is calculated by subtracting the entropy for binding from *1.0*:

	Portion		
Binding	**Positive**	**Negative**	**Entropy**
Hardcover	0.8	0.2	0.72
Leather	0.66	0.33	0.92
Paperback	0.2	0.8	0.72
Entropy for Binding			0.76
Information Gain			0.24

We can calculate the entropy for the use of color and genre in a similar manner. The information gain for color is *0.05*, and it is *0.15* for the genre. Thus, it makes more sense to use the binding type for the first level of the tree.

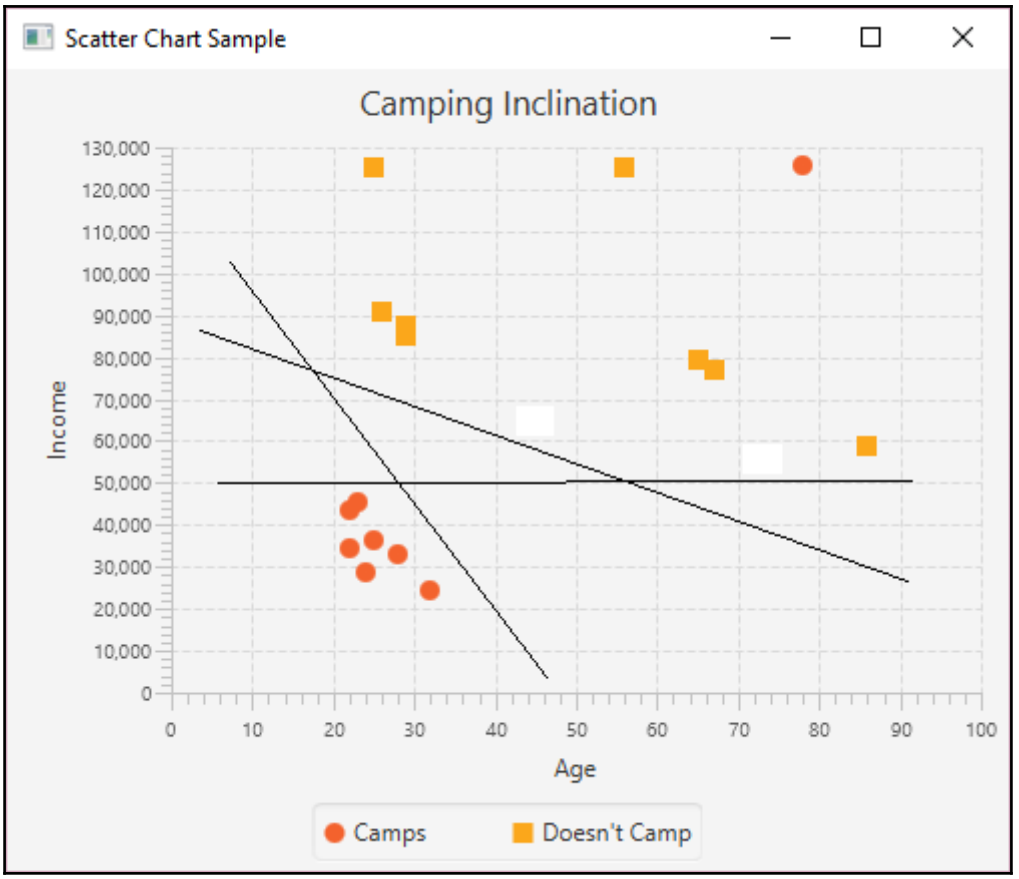
The resulting tree from the example consists of two levels, because the C4.5 algorithm determines that the remaining features do not provide any additional information gain.

Information gain can be problematic when a feature that has a large number of values is chosen, such as a customer's credit card number. Using this type of attribute will quickly narrow down the field, but it is too selective to be of much value.

Support vector machines

A **Support Vector Machine** (**SVM**) is a supervised machine learning algorithm used for both classification and regression problems. It is mostly used for classification problems. The approach creates a hyperplane to categorize the training data. A hyperplane can be envisioned as a geometric plane that separates two regions. In a two-dimensional space, it will be a line. In a three-dimensional space, it will be a two-dimensional plane. For higher dimensions, it is harder to conceptualize, but they do exist.

Consider the following figure depicting a distribution of two types of data points. The lines represent possible hyperplanes that separate these points. Part of the SVM process is to find the best hyperplane for the problem dataset. We will elaborate on this figure in the coding example.



Hyperplane example

Support vectors are data points that lie near the hyperplane. An SVM model uses the concept of a kernel to map input data to a higher order dimensional space to make the data more easily structured. A mapping function for doing this could lead to an infinite-dimensional space; that is, there could be an unbounded number of possible mappings.

However, what is known as the **kernel trick**, a kernel function is an approach that avoids this mapping and avoids possibly infeasible computations that might otherwise occur. SVMs support different types of kernels. A list of kernels can be found at `http://crsouza.com/2010/03/kernel-functions-for-machine-learning-applications/`. Choosing the right kernel depends on the problem. Commonly used kernels include:

- **Linear**: Uses a linear hyperplane
- **Polynomial**: Uses a polynomial equation for the hyperplane
- **Radial Basis Function (RBF)**: Uses a non-linear hyperplane
- **Sigmoid**: The sigmoid kernel, also known as the **Hyperbolic Tangent kernel**, comes from the neural networks field and is equivalent to a two-layer perceptron neural network

These kernels support different algorithms for analyzing data.

SVMs are useful for higher dimensional spaces that humans have a harder time visualizing. In the previous figure, two attributes were used to predict a third. An SVM can be used when many more attributes are present. The SVM needs to be trained and this can take longer with larger datasets.

We will use the Weka class `SMO` to demonstrate SVM analysis. The class supports John Platt's sequential minimal optimization algorithm. More information about this algorithm is found at `https://www.microsoft.com/en-us/research/publication/fast-training-of-support-vector-machines-using-sequential-minimal-optimization/`.

The `SMO` class supports the following kernels, which can be specified when using the class:

- **Puk**: The Pearson VII-function-based universal kernel
- **PolyKernel**: The polynomial kernel
- **RBFKernel**: The RBF kernel

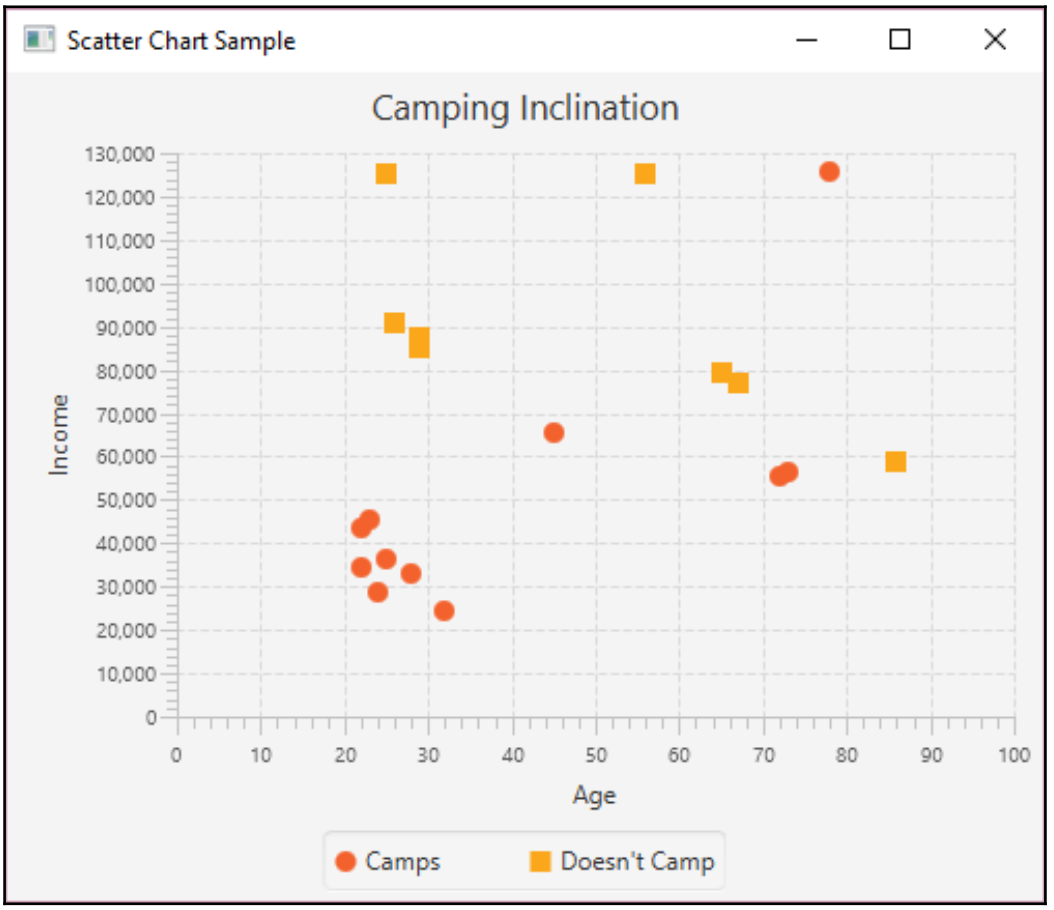
The algorithm uses training data to create a classification model. Test data can then be used to evaluate the model. We can also evaluate individual data elements.

Using an SVM for camping data

For illustration purposes, we will be using a dataset consisting of age, income, and whether someone camps. We would like to be able to predict whether someone is inclined to camp based on their age and income. The data we use is stored in `.arff` format and is not based on a survey but has been created to explain the SVM process. The input data is found in the `camping.txt` file, as shown next. The file extension does not need to be `.arff`:

```
@relation camping
@attribute age numeric
@attribute income numeric
@attribute camps {1, 0}
@data
23,45600,1
45,65700,1
72,55600,1
24,28700,1
22,34200,1
28,32800,1
32,24600,1
25,36500,1
26,91000,0
29,85300,0
67,76800,0
86,58900,0
56,125300,0
25,125000,0
22,43600,1
78,125700,1
73,56500,1
29,87600,0
65,79300,0
```

The following shows how the data is distributed graphically. Notice the outlier found in the upper-right corner. The JavaFX code that produces this graph is found at `http://www.packtpub.com/support`:



Camping Graph

We will start by reading in the data and handling exceptions:

```
try {
    BufferedReader datafile;
    datafile = readDataFile("camping.txt");
    ...
} catch (Exception ex) {
    // Handle exceptions
}
```

The `readDataFile` method follows:

```
public BufferedReader readDataFile(String filename) {
    BufferedReader inputReader = null;
    try {
        inputReader = new BufferedReader(
            new FileReader(filename));
    } catch (FileNotFoundException ex) {
        // Handle exceptions
    }
    return inputReader;
}
```

The `Instances` class holds a series of data instances, where each instance is an age, income, and camping value. The `setClassIndex` method indicates which of the attributes is to be predicted. In this case, it is the `camps` attribute:

```
Instances data = new Instances(datafile);
data.setClassIndex(data.numAttributes() - 1);
```

To train the model, we will split the dataset into two sets. The first `14` instances are used to train the model and the last `5` instances are used to test the model. The second argument of the `Instances` constructor specifies the beginning index in the dataset, and the last argument specifies how many instances to include:

```
Instances trainingData = new Instances(data, 0, 14);
Instances testingData = new Instances(data, 14, 5);
```

An `Evaluation` class instance is created to evaluate the model. An instance of the `SMO` class is also created. The `SMO` class's `buildClassifier` method builds the classifier using the dataset:

```
Evaluation evaluation = new Evaluation(trainingData);
Classifier smo = new SMO();
smo.buildClassifier(data);
```

The `evaluateModel` method evaluates the model using the testing data. The results are then displayed:

```
evaluation.evaluateModel(smo, testingData);
System.out.println(evaluation.toSummaryString());
```

The output follows. Notice one incorrectly classified instance. This corresponds to the outlier mentioned earlier:

```
Correctly Classified Instances 4 80 %
Incorrectly Classified Instances 1 20 %
Kappa statistic 0.6154
Mean absolute error 0.2
Root mean squared error 0.4472
Relative absolute error 41.0256 %
Root relative squared error 91.0208 %
Coverage of cases (0.95 level) 80 %
Mean rel. region size (0.95 level) 50 %
Total Number of Instances 5
```

Testing individual instances

We can also test an individual instance using the `classifyInstance` method. In the following sequence, we create a new instance using the `DenseInstance` class. It is then populated using the attributes of the camping dataset:

```
Instance instance = new DenseInstance(3);
instance.setValue(data.attribute("age"), 78);
instance.setValue(data.attribute("income"), 125700);
instance.setValue(data.attribute("camps"), 1);
```

The instance needs to be associated with the dataset using the `setDataset` method:

```
instance.setDataset(data);
```

The `classifyInstance` method is then applied to the `smo` instance and the results are displayed:

```
System.out.println(smo.classifyInstance(instance));
```

When executed, we get the following output:

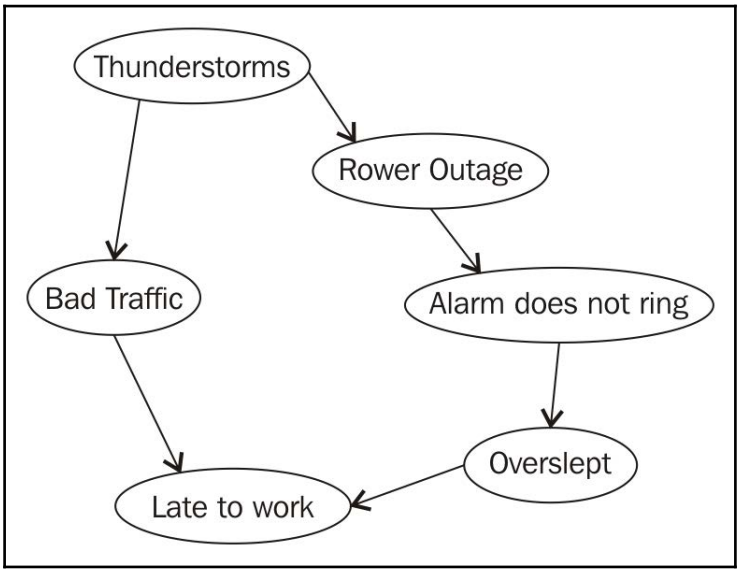
```
1.0
```

There are also alternate testing approaches. A common one is called **cross-validation folds**. This approach divides the dataset into *folds*, which are partitions of the dataset. Frequently, 10 partitions are created. Nine of the partitions are used for training and one for testing. This is repeated 10 times using a different partition of the dataset each time, and the average of the results is used. This technique is described at `https://weka.wikispaces.com/Generating+cross-validation+folds+(Java+approach)`.

We will now examine the purpose and use of Bayesian networks.

Bayesian networks

Bayesian networks, also known as **Bayes nets** or **belief networks**, are models designed to reflect a particular world or environment by depicting the states of different attributes of the world and their statistical relationships. The models can be used to show a wide variety of real-world scenarios. In the following diagram, we model a system depicting the relationship between various factors and our likelihood of being late to work:



Bayesian network

Each circle on the diagram represents a node or part of the system, which can have various values and probabilities for each value. For example, **Power Outage** might be true or false---either the power went out or it did not. The probability of the power going out affects the probability that your alarm will not ring, you might oversleep, and thus be late to work.

The nodes at the top of the diagram tend to imply a higher level of causality than those at the bottom. The higher nodes are called **parent nodes**, and they may have one or more child nodes. Bayesian networks only relate nodes that have a causal dependency and therefore allow more efficient computation of probabilities. Unlike other models, we would not have to store and analyze every possible combination of states of each node. Instead, we can calculate and store probabilities of related nodes. Additionally, Bayesian networks are easily adaptable and can grow as more knowledge about a particular world is acquired.

Using a Bayesian network

To model this type of network using Java, we will create a network using JBayes (`https://github.com/vangj/jbayes`). JBayes is an open source library for creating a simple **Bayesian Belief Network** (**BBN**). It is available at no cost for personal or commercial use. In our next example, we will perform approximate inference, a technique considered less accurate but allowing for decreased computational time. This technique is often used when working with big data because it produces a reliable model in a reasonable amount of time. We conduct approximate inference by using weight sampling of each node. JBayes also provides support for exact inference. Exact inference is most often used with smaller datasets or in situations where accuracy is of most importance. JBayes performs exact inference using the junction tree algorithm.

To begin our approximate inference model, we will first create our nodes. We will use the preceding diagram depicting attributes affecting on-time arrival to work to build our network. In the following code example, we use method chaining to create our nodes. Three of the methods take a `String` parameter. The `name` method is the name associated with each node. For brevity, we are only using the first initial, so *s* represents `storms`, `t` represents `traffic`, and so on. The `value` method allows us to set values for the node. In each case, our nodes can only have two values: `t` for true or `f` for false:

```
Node storms = Node.newBuilder().name("s").value("t").value("f").build();
Node traffic = Node.newBuilder().name("t").value("t").value("f").build();
Node powerOut = Node.newBuilder().name("p").value("t").value("f").build();
Node alarm = Node.newBuilder().name("a").value("t").value("f").build();
Node overslept = Node.newBuilder().name("o").value("t").value("f").build();
Node lateToWork =
Node.newBuilder().name("l").value("t").value("f").build();
```

Next, we assign parents to each of our child nodes. Notice that `storms` is a parent node to both `traffic` and `powerOut`. The `lateToWork` node has two parent nodes, `traffic` and `overslept`:

```
traffic.addParent(storms);
powerOut.addParent(storms);
lateToWork.addParent(traffic);
alarm.addParent(powerOut);
overslept.addParent(alarm);
lateToWork.addParent(overslept);
```

Then we define the **conditional probability tables** (**CPTs**) for each of our nodes. These tables are basically two-dimensional arrays representing the probabilities of each attribute of each node. If we have more than one parent node, as in the case of the `lateToWork` node, we need a row for each. We have used arbitrary values for our probabilities in this example, but note that each row must sum to `1.0`:

```
storms.setCpt(new double[][] {{0.7, 0.3}});
traffic.setCpt(new double[][] {{0.8, 0.2}});
powerOut.setCpt(new double[][] {{0.5, 0.5}});
alarm.setCpt(new double[][] {{0.7, 0.3}});
overslept.setCpt(new double[][] {{0.5, 0.5}});
lateToWork.setCpt(new double[][] {
   {0.5, 0.5},
   {0.5, 0.5}
});
```

Finally, we create a `Graph` object and add each node to our graph structure. We then use this graph to perform our sampling:

```
Graph bayesGraph = new Graph();
bayesGraph.addNode(storms);
bayesGraph.addNode(traffic);
bayesGraph.addNode(powerOut);
bayesGraph.addNode(alarm);
bayesGraph.addNode(overslept);
bayesGraph.addNode(lateToWork);
bayesGraph.sample(1000);
```

At this point, we may be interested in the probabilities of each event. We can use the `prob` method to check the probabilities of a `True` or `False` value for each node:

```
double[] stormProb = storms.probs();
double[] trafProb = traffic.probs();
double[] powerProb = powerOut.probs();
double[] alarmProb = alarm.probs();
double[] overProb = overslept.probs();
```

```
double[] lateProb = lateToWork.probs();

out.println("nStorm Probabilities");
out.println("True: " + stormProb[0] + " False: " + stormProb[1]);
out.println("nTraffic Probabilities");
out.println("True: " + trafProb[0] + " False: " + trafProb[1]);
out.println("nPower Outage Probabilities");
out.println("True: " + powerProb[0] + " False: " + powerProb[1]);
out.println("vAlarm Probabilities");
out.println("True: " + alarmProb[0] + " False: " + alarmProb[1]);
out.println("nOverslept Probabilities");
out.println("True: " + overProb[0] + " False: " + overProb[1]);
out.println("nLate to Work Probabilities");
out.println("True: " + lateProb[0] + " False: " + lateProb[1]);
```

Our output contains the probabilities of each value for each node. For example, the probability of a storm occurring is 71% while the probability of a storm not occurring is 29%:

```
Storm Probabilities
True: 0.71 False: 0.29
Traffic Probabilities
True: 0.726 False: 0.274
Power Outage Probabilities
True: 0.442 False: 0.558
Alarm Probabilities
True: 0.543 False: 0.457
Overslept Probabilities
True: 0.556 False: 0.444
Late to Work Probabilities
True: 0.469 False: 0.531
```



Notice in this example that we have used numbers that produce a very high likelihood of being late to work, roughly 47%. This is due to the fact that we have set the probabilities of our parent nodes fairly high as well. This data would vary dramatically if the chances of a storm were lower or if we changed some of the other child nodes as well.

If we would like to save information about our sample, we can save the data to a CSV file using the following code:

```
try {
   CsvUtil.saveSamples(bayesGraph, new FileWriter(
        new File("C://JBayesInfo.csv")));
} catch (IOException e) {
   // Handle exceptions
}
```

With this discussion of supervised learning complete, we will now move on to unsupervised learning.

Unsupervised machine learning

Unsupervised machine learning does not use annotated data; that is, the dataset does to contain anticipated results. While there are several unsupervised learning algorithms, we will demonstrate the use of association rule learning to illustrate this learning approach.

Association rule learning

Association rule learning is a technique that identifies relationships between data items. It is part of what is called **market basket analysis**. When a shopper makes purchases, these purchases are likely to consist of more than one item, and when it does, there are certain items that tend to be bought together. Association rule learning is one approach for identifying these related items. When an association is found, a rule can be formulated for it.

For example, if a customer buys diapers and lotion, they are also likely to buy baby wipes. An analysis can find these associations and a rule stating the observation can be formed. The rule would be expressed as *{diapers, lotion} => {wipes}*. Being able to identify these purchasing patterns allows a store to offer special coupons, arrange their products to be easier to get, or effect any number of other market-related activities.

One of the problems with this technique is that there are a large number of possible associations. One efficient method that is commonly used is the **apriori** algorithm. This algorithm works on a collection of transactions defined by a set of items. These items can be thought of as purchases and a transaction as a set of items bought together. The collection is often referred to as a database.

Consider the following set of transactions where a *1* indicates that the item was purchased as part of a transaction and *0* means that it was not purchased:

Transaction ID	Diapers	Lotion	Wipes	Formula
1	1	1	1	0
2	1	1	1	1
3	0	1	1	0
4	1	0	0	0
5	0	1	1	1

There are several analysis terms used with the apriori model:

- **Support**: This is the proportion of the items in a database that contain a subset of items. In the previous database, the item *{diapers, lotion}* occurs *2/5* times or *20%*.
- **Confidence**: This is a measure of the frequency of the rule being true. It is calculated as $conf(X- > Y) = sup(X \cup Y)/sup(X)$.
- **Lift**: This measures the degree to which the items are dependent upon each other. It is defined as lift(X -> Y) = $sup(X \cup Y) / (sup(X) * sup(Y))$.
- **Leverage**: Leverage is a measure of the number of transactions that are covered by both *X* and *Y* if *X* and *Y* are independent of each other. A value above *0* is a good indicator. It is calculated as $lev(X->Y) = sup(X ,Y) - sup(X) * sup(Y)$.
- **Conviction**: A measure of how often the rule makes an incorrect decision. It is defined as $conv(X -> Y) = 1 - sup(Y) / (1 - conf(X -> Y))$.

These definitions and sample values can be found at `https://en.wikipedia.org/wiki/Association_rule_learning`.

Using association rule learning to find buying relationships

We will be using the `Apriori` Weka class to demonstrate Java support for the algorithm using two datasets. The first is the data discussed previously and the second deals with what a person may take on a hike.

The following is the data file, `babies.arff`, for baby information:

```
@relation TEST_ITEM_TRANS
@attribute Diapers {1, 0}
@attribute Lotion {1, 0}
@attribute Wipes {1, 0}
@attribute Formula {1, 0}
@data
1,1,1,0
1,1,1,1
0,1,1,0
1,0,0,0
0,1,1,1
```

We start by reading in the file using a `BufferedReader` instance. This object is used as the argument of the `Instances` class, which will hold the data:

```
try {
    BufferedReader br;
    br = new BufferedReader(new FileReader("babies.arff"));
   Instances data = new Instances(br);
    br.close();
    ...
} catch (Exception ex) {
    // Handle exceptions
}
```

Next, an `Apriori` instance is created. We set the number of rules to be generated and a minimum confidence for the rules:

```
Apriori apriori = new Apriori();
apriori.setNumRules(100);
apriori.setMinMetric(0.5);
```

The `buildAssociations` method generates the associations using the `Instances` variable. The associations are then displayed:

```
apriori.buildAssociations(data);
System.out.println(apriori);
```

There will be 100 rules displayed. The following is the abbreviated output. Each rule is followed by various measures of the rule:



Note that rule 8 and 100 reflect the previous examples.

```
Apriori
=======
Minimum support: 0.3 (1 instances)
Minimum metric <confidence>: 0.5
Number of cycles performed: 14
Generated sets of large itemsets:
Size of set of large itemsets L(1): 8
Size of set of large itemsets L(2): 18
Size of set of large itemsets L(3): 16
Size of set of large itemsets L(4): 5
Best rules found:
1. Wipes=1 4 ==> Lotion=1 4 <conf:(1)> lift:(1.25) lev:(0.16) [0]
conv:(0.8)
2. Lotion=1 4 ==> Wipes=1 4 <conf:(1)> lift:(1.25) lev:(0.16) [0]
conv:(0.8)
3. Diapers=0 2 ==> Lotion=1 2 <conf:(1)> lift:(1.25) lev:(0.08) [0]
conv:(0.4)
4. Diapers=0 2 ==> Wipes=1 2 <conf:(1)> lift:(1.25) lev:(0.08) [0]
conv:(0.4)
5. Formula=1 2 ==> Lotion=1 2 <conf:(1)> lift:(1.25) lev:(0.08) [0]
conv:(0.4)
6. Formula=1 2 ==> Wipes=1 2 <conf:(1)> lift:(1.25) lev:(0.08) [0]
conv:(0.4)
7. Diapers=1 Wipes=1 2 ==> Lotion=1 2 <conf:(1)> lift:(1.25) lev:(0.08) [0]
conv:(0.4)
8. Diapers=1 Lotion=1 2 ==> Wipes=1 2 <conf:(1)> lift:(1.25) lev:(0.08) [0]
conv:(0.4)
...
62. Diapers=0 Lotion=1 Formula=1 1 ==> Wipes=1 1 <conf:(1)> lift:(1.25)
lev:(0.04) [0] conv:(0.2)
...
99. Lotion=1 Formula=1 2 ==> Diapers=1 1 <conf:(0.5)> lift:(0.83)
lev:(-0.04) [0] conv:(0.4)
100. Diapers=1 Lotion=1 2 ==> Formula=1 1 <conf:(0.5)> lift:(1.25)
lev:(0.04) [0] conv:(0.6)
```

This provides us with a list of relationships, which we can use to identify patterns in activities such as purchasing behavior.

Reinforcement learning

Reinforcement learning is a type of learning at the cutting edge of current research into neural networks and machine learning. Unlike unsupervised and supervised learning, reinforcement learning makes decisions based upon the results of an action. It is a goal-oriented learning process, similar to that used by many parents and teachers across the world. We teach children to study and perform well on tests so that they receive high grades as a reward. Likewise, reinforcement learning can be used to teach machines to make choices that will result in the highest reward.

There are four main components to reinforcement learning: the actor or agent, the state or scenario, the chosen action, and the reward. The actor is the object or vehicle making the decisions within the application. The state is the world the actor exists within. Any decision the actor makes occurs within the parameters of the state. The action is simply the choice the actor makes when given a set of options. The reward is the result of each action and influences the likelihood of choosing that particular action in the future.

It is essential to note that the action and the state where the action occurred are not independent. In fact, the correct, or highest rewarding, action can often depend upon the state in which the action occurs. If the actor is trying to decide how to cross a body of water, swimming might be a good option if the body of water is calm and rather small. Swimming would be a terrible choice for the actor to choose if he wanted to cross the Pacific Ocean.

To handle this problem, we can consider the **Q** function. This function results from the mapping of a particular state to an action within the state. The Q function would link a lower reward to swimming across the Pacific that it might for swimming across a small river. Rather than saying swimming is a low reward activity, the Q function allows for swimming to sometimes have a low reward and other times a higher reward.

Reinforcement learning always begins with a blank slate. The actor does not know the best path or sequence of decisions when the iteration first begins. However, after many iterations through a given problem, considering the results of each particular state-action pair choice, the algorithm improves and learns to make the highest rewarding choices.

The algorithms used to achieve reinforcement learning involve maximization of reward amidst a complex set of processes and choices. Though currently being tested in video games and other discrete environments, the ultimate goal is success of these algorithms in unpredictable, real-world scenarios. Within the topic of reinforcement learning, there are three main flavors or types: temporal difference learning, Q`-learning, and **State-Action-Reward-State-Action** (**SARSA**).

Temporal difference learning takes into account previously learned information to inform future decisions. This type of learning assumes a correlation between past and future decisions. Before an action is taken, a prediction is made. This prediction is compared to other known information about the environment and similar decisions before an action is chosen. This process is known as bootstrapping and is thought to create more accurate and useful results.

Q-learning uses the Q function mentioned above to select not only the best action for one particular step in a given state, but the action that will lead to the highest reward *from that point forward*. This is known as the optimum policy. One great advantage Q-learning offers is the ability to make decisions without requiring a full model of the state. This allows it to function in states with random changes in actions and rewards.

SARSA is another algorithm used in reinforcement learning. Its name is fairly self-explanatory: the Q value depends upon the current **State**, the current chosen **Action**, the **Reward** for that action, the State the agent will exist in after the action is completed, and the subsequent Action taken in the new state. This algorithm looks further ahead one step to make the best possible decision.

There are limited tools currently available for performing reinforcement learning using Java. One popular tool is **Platform for Implementing Q-Learning Experiments** (**Piqle**). This Java framework aims to provide tools for fast designing and testing or reinforcement learning experiments. Piqle can be downloaded from `http://piqle.sourceforge.net`. Another robust tool is called **Brown-UMBC Reinforcement Learning and Planning** (**BURPLAP**). Found at `http://burlap.cs.brown.edu`, this library is also designed for development of algorithms and domains for reinforcement learning. This particular resource boasts in the flexibility of states and actions and supports a wide range of planning and learning algorithms. BURLAP also includes analysis tools for visualization purposes.

Summary

Machine learning is concerned with developing techniques that allow the applications to learn without having to be explicitly programmed to solve a problem. This flexibility allows such applications to be used in more varied settings with little to no modifications.

We saw how training data is used to create a model. Once the model has been trained, the model is evaluated using testing data. Both the training data and testing data come from the problem domain. Once trained, the model is used with other input data to make predictions.

We learned how the Weka Java API is used to create decision trees. This tree consists of internal nodes that represent different attributes of the problem. The leaves of the tree represent results. Since there are many ways of constructing a tree, part of the job of a decision tree is to create the best tree.

Support vector machines divide a dataset into sections thus classifying elements in the dataset. This classification s based on the attributes of the data such as age, hair color, or weight. With the model, it is possible to predict outcomes based on attributes of a data instance.

Bayesian networks are used to make predictions based on parent-child relationships between nodes. The probability of one event directly affects the probability of a child event, and we can use this information to predict outcomes of complex real-world environments.

In the section on association rule learning, we learned how the relationships between elements of a dataset can be identified. The more significant relationships allow us to create rules that are applied to solve various problems.

In our discussion of reinforcement learning, we discussed the elements of agent, state, action, and reward and their relationship to one another. We also discussed specific types of reinforcement learning and provided resources for further inquiry.

Having introduced the elements of machine learning, we are now ready to explore neural networks, found in the next chapter.

7
Neural Networks

While neural networks have been around for a number of years, they have grown in popularity due to improved algorithms and more powerful machines. Some companies are building hardware systems that explicitly mimic neural networks (`https://www.wired.com/2016/05/google-tpu-custom-chips/`). The time has come to use this versatile technology to address data science problems.

In this chapter, we will explore the ideas and concepts behind neural networks and then demonstrate their use. Specifically, we will:

- Define and illustrate neural networks
- Describe how they are trained
- Examine various neural network architectures
- Discuss and demonstrate several different neural networks, including:
    - A simple Java example
    - A **Multi Layer Perceptron** (**MLP**) network
    - The **k-Nearest Neighbor** (**k-NN**) algorithm and others

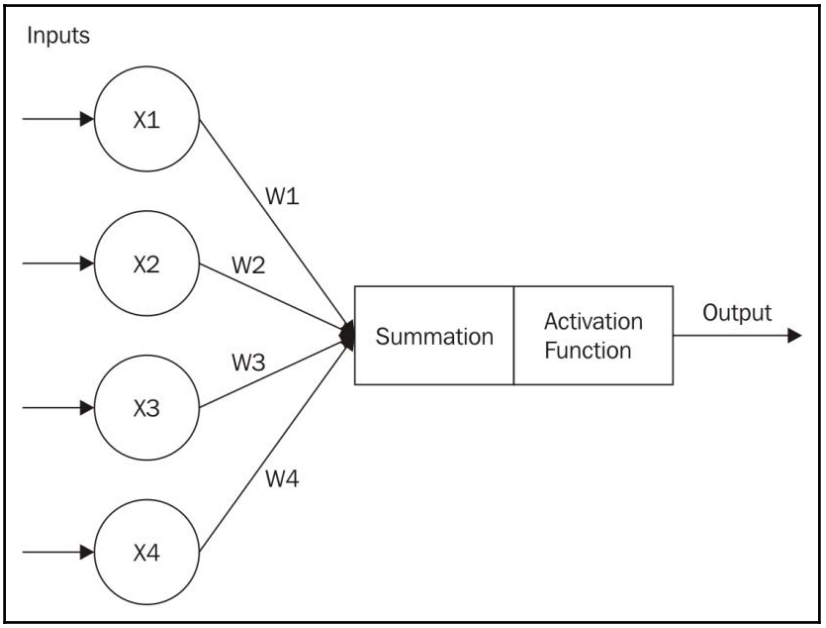
The idea for an **Artificial Neural Network** (**ANN**), which we will call a neural network, originates from the neuron found in the brain. A **neuron** is a cell that has **dendrites** connecting it to input sources and other neurons. It receives stimulus from multiple sources through the **dendrites**. Depending on the source, the weight allocated to a source, the neuron is activated and **fires** a signal down a dendrite to another neuron. A collection of neurons can be trained and will respond to a particular set of input signals.

An artificial neuron is a node that has one or more inputs and a single output. Each input has a weight associated with it. By weighting inputs, we can amplify or de-amplify an input.



Artificial neurons are alternately called **perceptrons**.

This is depicted in the following diagram, where the weights are summed and then sent to an **Activation Function** that determines the **Output.**



The neuron, and ultimately a collection of neurons, operate in one of two modes:

- **Training mode** - The neuron is trained to fire when a certain set of inputs are received
- **Testing mode** - Input is provided to the neuron, which responds as trained to a known set of inputs

A dataset is frequently split into two parts. A larger part is used to train a model. The second part is used to test and verify the model.

The output of a neuron is determined by the sum of the weighted inputs. Whether a neuron fires or not is determined by an **activation function**. There are several different types of activations functions, including:

- **Step function** - This linear function is computed using the summation of the weighted inputs as follows:

$$Net_i = \sum W_i X_i$$

The *f(Net)* designates the output of a function. It is *1*, if the *Net* input is greater than the activation threshold. When this happens the neuron fires. Otherwise it returns *0* and doesn't fire. The value is calculated based on all of the dendrite inputs.

- **Sigmoid** - This is a nonlinear function and is computed as follows:

$$f\left(Net\right) = \frac{1}{1 + e^{-Net_i}}$$

As the neuron is trained, the weights with each input can be adjusted.

In contrast to the step function, the sigmoid function is non-linear. This better matches some problem domains. We will find the sigmoid function used in multi-layer neural networks.

Training a neural network

There are three basic training approaches:

- **Supervised learning** - With supervised learning the model is trained with data that matches input sets to output values
- **Unsupervised learning** - In unsupervised learning, the data does not contain results, but the model is expected to determine relationships on its own
- **Reinforcement learning** - Similar to supervised learning, but a reward is provided for good results

These datasets differ in the information they contain. Supervised and reinforcement learning contain correct output for a set of input. The unsupervised learning does not contain correct results.

A neural network learns (at least with supervised learning) by feeding an input into a network and comparing the results, using the activation function, to the expected outcome. If they match, then the network has been trained correctly. If they don't match then the network is modified.

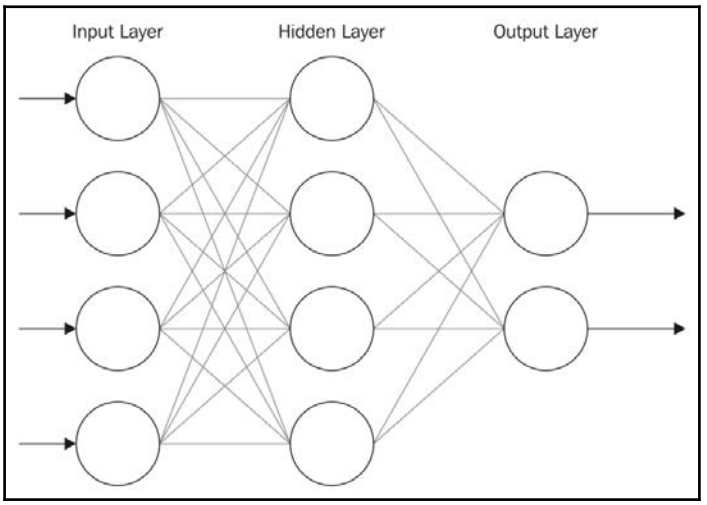
When we modify the weights we need to be careful not to change them too drastically. If the change is too large, then the results may change too much and we may miss the desired output. If the change is too little, then training the model will take too long. There are times when we may not want to change some weights.

A **bias** unit is a neuron that has a constant output. It is always one and is sometimes referred to as a fake node. This neuron is similar to an offset and is essential for most networks to function properly. You could compare the bias neuron to the *y*-intercept of a linear function in slope-intercept form. Just as adjusting the *y*-intercept value changes the location of the line, but not the shape/slope, the bias neuron can change the output values without adjusting the shape or function of the network. You can adjust the outputs to fit the particular needs of your problem.

Getting started with neural network architectures

Neural networks are usually created using a series of layers of neurons. There is typically an **Input Layer**, one or more middle layers (**Hidden Layer**), and an **Output Layer**.

The following is the depiction of a feedforward network:



The number of nodes and layers will vary. A feedforward network moves the information forward. There are also feedback networks where information is passed backwards. Multiple hidden layers are needed to handle the more complicated processing required for most analysis.

We will discuss several architectures and algorithms related to different types of neural networks throughout this chapter. Due to the complexity and length of explanation required, we will only provide in-depth analysis of a few key network types. Specifically, we will demonstrate a simple neural network, MLPs, and **Self-Organizing Maps** (**SOMs**).

We will, however, provide an overview of many different options. The type of neural network and algorithm implementation appropriate for any particular model will depend upon the problem being addressed.

Understanding static neural networks

Static neural networks are ANNs that undergo a training or learning phase and then do not change when they are used. They differ from dynamic neural networks, which learn constantly and may undergo structural changes after the initial training period. Static neural networks are useful when the results of a model are relatively easy to reproduce or are more predictable. We will look at dynamic neural networks in a moment, but we will begin by creating our own basic static neural network.

A basic Java example

Before we examine various libraries and tools available for constructing neural networks, we will implement our own basic neural network using standard Java libraries. The next example is an adaptation of work done by Jeff Heaton (`http://www.informit.com/articles/article.aspx?p=30596`). We will construct a feed-forward backpropagation neural network and train it to recognize the XOR operator pattern. Here is the basic truth table for XOR:

X	Y	Result
0	0	0
0	1	1
1	0	1
1	1	0

This network needs only two input neurons and one output neuron corresponding to the *X* and *Y* input and the result. The number of input and output neurons needed for models is dependent upon the problem at hand. The number of hidden neurons is often the sum of the number of input and output neurons, but the exact number may need to be changed as training progresses.

We are going to demonstrate how to create and train the network next. We first provide the network with an input and observe the output. The output is compared to the expected output and then the weight matrix, called `weightChanges`, is adjusted. This adjustment ensures that the subsequent output will be closer to the expected output. This process is repeated until we are satisfied that the network can produce results significantly close enough to the expected output. In this example, we present the input and output as arrays of doubles where each input or output neuron is an element of the array.



The input and output are sometimes referred to as **patterns**.

First, we will create a `SampleNeuralNetwork` class to implement the network. Begin by adding the variables listed underneath to the class. We will discuss and demonstrate their purposes later in this section. Our class contains the following instance variables:

```
double errors;
int inputNeurons;
int outputNeurons;
int hiddenNeurons;
int totalNeurons;
int weights;
double learningRate;
double outputResults[];
double resultsMatrix[];
double lastErrors[];
double changes[];
double thresholds[];
double weightChanges[];
double allThresholds[];
double threshChanges[];
double momentum;
double errorChanges[];
```

Next, let's take a look at our constructor. We have four parameters, representing the number of inputs to our network, the number of neurons in hidden layers, the number of output neurons, and the rate and momentum at which we wish for learning to occur. The `learningRate` is a parameter that specifies the magnitude of changes in weight and bias during the training process. The `momentum` parameter specifies what fraction of a previous weight should be added to create a new weight. It is useful to prevent convergence at **local minimums** or **saddle points**. A high momentum increases the speed of convergence in a system, but can lead to an unstable system if it is too high. Both the momentum and learning rate should be values between `0` and `1`:

```
public SampleNeuralNetwork(int inputCount,
         int hiddenCount,
         int outputCount,
         double learnRate,
         double momentum) {
   ...
}
```

Within our constructor we initialize all private instance variables. Notice that `totalNeurons` is set to the sum of all inputs, outputs, and hidden neurons. This sum is then used to set several other variables. Also notice that the `weights` variable is calculated by finding the product of the number of inputs and hidden neurons, the product of the hidden neurons and the outputs, and adding these two products together. This is then used to create new arrays of length weight:

```
learningRate = learnRate;
momentum = momentum;

inputNeurons = inputCount;
hiddenNeurons = hiddenCount;
outputNeurons = outputCount;
totalNeurons = inputCount + hiddenCount + outputCount;
weights = (inputCount * hiddenCount)
   + (hiddenCount * outputCount);

outputResults    = new double[totalNeurons];
resultsMatrix   = new double[weights];
weightChanges = new double[weights];
thresholds = new double[totalNeurons];
errorChanges = new double[totalNeurons];
lastErrors    = new double[totalNeurons];
allThresholds = new double[totalNeurons];
changes = new double[weights];
threshChanges = new double[totalNeurons];
reset();
```

Notice that we call the `reset` method at the end of the constructor. This method resets the network to begin training with a random weight matrix. It initializes the thresholds and results matrices to random values. It also ensures that all matrices used for tracking changes are set back to zero. Using random values ensures that different results can be obtained:

```
public void reset() {
   int loc;
   for (loc = 0; loc < totalNeurons; loc++) {
         thresholds[loc] = 0.5 - (Math.random());
         threshChanges[loc] = 0;
         allThresholds[loc] = 0;
   }
   for (loc = 0; loc < resultsMatrix.length; loc++) {
         resultsMatrix[loc] = 0.5 - (Math.random());
         weightChanges[loc] = 0;
         changes[loc] = 0;
   }
}
```

We also need a method called `calcThreshold`. The **threshold** value specifies how close a value has to be to the actual activation threshold before the neuron will fire. For example, a neuron may have an activation threshold of `1`. The threshold value specifies whether a number such as `0.999` counts as `1`. This method will be used in subsequent methods to calculate the thresholds for individual values:

```
public double threshold(double sum) {
   return 1.0 / (1 + Math.exp(-1.0 * sum));
}
```

Next, we will add a method to calculate the output using a given set of inputs. Both our input parameter and the data returned by the method are arrays of `double` values. First, we need two position variables to use in our loops, `loc` and `pos`. We also want to keep track of our position within arrays based upon the number of input and hidden neurons. The index for our hidden neurons will start after our input neurons, so its position is the same as the number of input neurons. The position of our output neurons is the sum of our input neurons and hidden neurons. We also need to initialize our `outputResults` array:

```
public double[] calcOutput(double input[]) {
   int loc, pos;
   final int hiddenIndex = inputNeurons;
   final int outIndex = inputNeurons + hiddenNeurons;

   for (loc = 0; loc < inputNeurons; loc++) {
         outputResults[loc] = input[loc];
   }
...
```

```
}
```

Then we calculate outputs based upon our input neurons for the first layer of our network. Notice our use of the `threshold` method within this section. Before we can place our sum in the `outputResults` array, we need to utilize the `threshold` method:

```
int rLoc = 0;
for (loc = hiddenIndex; loc < outIndex; loc++) {
      double sum = thresholds[loc];
      for (pos = 0; pos < inputNeurons; pos++) {
            sum += outputResults[pos] * resultsMatrix[rLoc++];
      }
      outputResults[loc] = threshold(sum);
}
```

Now we take into account our hidden neurons. Notice the process is similar to the previous section, but we are calculating outputs for the hidden layer rather than the input layer. At the end, we return our result. This result is in the form of an array of doubles containing the values of each output neuron. In our example, there is only one output neuron:

```
double result[] = new double[outputNeurons];
for (loc = outIndex; loc < totalNeurons; loc++) {
      double sum = thresholds[loc];

      for (pos = hiddenIndex; pos < outIndex; pos++) {
            sum += outputResults[pos] * resultsMatrix[rLoc++];
      }
      outputResults[loc] = threshold(sum);
      result[loc-outIndex] = outputResults[loc];
}

return result;
```

It is quite likely that the output does not match the expected output, given our XOR table. To handle this, we use error calculation methods to adjust the weights of our network to produce better output. The first method we will discuss is the `calcError` method. This method will be called every time a set of outputs is returned by the `calcOutput` method. It does not return data, but rather modifies arrays containing weight and threshold values. The method takes an array of doubles representing the ideal value for each output neuron. Notice we begin as we did in the `calcOutput` method and set up indexes to use throughout the method. Then we clear out any existing hidden layer errors:

```
public void calcError(double ideal[]) {
   int loc, pos;
   final int hiddenIndex = inputNeurons;
   final int outputIndex = inputNeurons + hiddenNeurons;

      for (loc = inputNeurons; loc < totalNeurons; loc++) {
            lastErrors[loc] = 0;
      }
```

Next we calculate the difference between our expected output and our actual output. This allows us to determine how to adjust the weights for further training. To do this, we loop through our arrays containing the expected outputs, `ideal`, and the actual outputs, `outputResults`. We also adjust our errors and change in errors in this section:

```
      for (loc = outputIndex; loc < totalNeurons; loc++) {
         lastErrors[loc] = ideal[loc - outputIndex] -
            outputResults[loc];
         errors += lastErrors[loc] * lastErrors[loc];
         errorChanges[loc] = lastErrors[loc] * outputResults[loc]
            *(1 - outputResults[loc]);
      }

      int locx = inputNeurons * hiddenNeurons;
      for (loc = outputIndex; loc < totalNeurons; loc++) {
            for (pos = hiddenIndex; pos < outputIndex; pos++) {
                  changes[locx] += errorChanges[loc] *
                        outputResults[pos];
                  lastErrors[pos] += resultsMatrix[locx] *
                        errorChanges[loc];
                  locx++;
            }
            allThresholds[loc] += errorChanges[loc];
      }
```

Next we calculate and store the change in errors for each neuron. We use the `lastErrors` array to modify the `errorChanges` array, which contains total errors:

```
for (loc = hiddenIndex; loc < outputIndex; loc++) {
      errorChanges[loc] = lastErrors[loc] *outputResults[loc]
            * (1 - outputResults[loc]);
}
```

We also fine tune our system by making changes to the `allThresholds` array. It is important to monitor the changes in errors and thresholds so the network can improve its ability to produce correct output:

```
   locx = 0;
   for (loc = hiddenIndex; loc < outputIndex; loc++) {
         for (pos = 0; pos < hiddenIndex; pos++) {
               changes[locx] += errorChanges[loc] *
                     outputResults[pos];
               lastErrors[pos] += resultsMatrix[locx] *
                     errorChanges[loc];
               locx++;
         }
         allThresholds[loc] += errorChanges[loc];
   }
}
```

We have one other method used for calculating errors in our network. The `getError` method calculates the root mean square for our entire set of training data. This allows us to identify our average error rate for the data:

```
public double getError(int len) {
   double err = Math.sqrt(errors / (len * outputNeurons));
   errors = 0;
   return err;
}
```

Now that we can initialize our network, compute outputs, and calculate errors, we are ready to train our network. We accomplish this through the use of the `train` method. This method makes adjustments first to the weights based upon the errors calculated in the previous method, and then adjusts the thresholds:

```
public void train() {
   int loc;
   for (loc = 0; loc < resultsMatrix.length; loc++) {
      weightChanges[loc] = (learningRate * changes[loc]) +
         (momentum * weightChanges[loc]);
      resultsMatrix[loc] += weightChanges[loc];
      changes[loc] = 0;
   }
   for (loc = inputNeurons; loc < totalNeurons; loc++) {
      threshChanges[loc] = learningRate * allThresholds[loc] +
```

```
            (momentum * threshChanges[loc]);
         thresholds[loc] += threshChanges[loc];
         allThresholds[loc] = 0;
      }
   }
```

Finally, we can create a new class to test our neural network. Within the `main` method of another class, add the following code to represent the XOR problem:

```
double xorIN[][] ={
               {0.0,0.0},
               {1.0,0.0},
               {0.0,1.0},
               {1.0,1.0}};

double xorEXPECTED[][] = { {0.0},{1.0},{1.0},{0.0}};
```

Next we want to create our new `SampleNeuralNetwork` object. In the following example, we have two input neurons, three hidden neurons, one output neuron (the XOR result), a learn rate of `0.7`, and a momentum of `0.9`. The number of hidden neurons is often best determined by trial and error. In subsequent executions, consider adjusting the values in this constructor and examine the difference in results:

```
SampleNeuralNetwork network = new
               SampleNeuralNetwork(2,3,1,0.7,0.9);
```



The learning rate and momentum should usually fall between zero and one.

We then repeatedly call our `calcOutput`, `calcError`, and `train` methods, in that order. This allows us to test our output, calculate the error rate, adjust our network weights, and then try again. Our network should display increasingly accurate results:

```
for (int runCnt=0;runCnt<10000;runCnt++) {
   for (int loc=0;loc<xorIN.length;loc++) {
         network.calcOutput(xorIN[loc]);
         network.calcError(xorEXPECTED[loc]);
         network.train();
   }
   System.out.println("Trial #" + runCnt + ",Error:" +
               network.getError(xorIN.length));
}
```

Execute the application and notice that the error rate changes with each iteration of the loop. The acceptable error rate will depend upon the particular network and its purpose. The following is some sample output from the preceding code. For brevity we have included the first and the last training output. Notice that the error rate is initially above 50%, but falls to close to 1% by the last run:

```
Trial #0,Error:0.5338334002845255
Trial #1,Error:0.5233475199946769
Trial #2,Error:0.5229843653785426
Trial #3,Error:0.5226263062497853
Trial #4,Error:0.5226916275713371
...
Trial #994,Error:0.014457034704806316
Trial #995,Error:0.01444865096401158
Trial #996,Error:0.01444028142777395
Trial #997,Error:0.014431926056394229
Trial #998,Error:0.01442358481032747
Trial #999,Error:0.014415257650182488
```

In this example, we have used a small scale problem and we were able to train our network rather quickly. In a larger scale problem, we would start with a training set of data and then use additional datasets for further analysis. Because we really only have four inputs in this scenario, we will not test it with any additional data.

This example demonstrates some of the inner workings of a neural network, including details about how errors and output can be calculated. By exploring a relatively simple problem we are able to examine the mechanics of a neural network. In our next examples, however, we will use tools that hide these details from us, but allow us to conduct robust analysis.

Understanding dynamic neural networks

Dynamic neural networks differ from static networks in that they continue learning after the training phase. They can make adjustments to their structure independently of external modification. A **feedforward neural network (FNN)** is one of the earliest and simplest dynamic neural networks. This type of network, as its name implies, only feeds information forward and does not form any cycles. This type of network formed the foundation for much of the later work in dynamic ANNs. We will show in-depth examples of two types of dynamic networks in this section, MLP networks and SOMs.

Multilayer perceptron networks

A MLP network is a FNN with multiple layers. The network uses supervised learning with backpropagation where feedback is sent to early layers to assist in the learning process. Some of the neurons use a nonlinear activation function mimicking biological neurons. Every nodes of one layer is fully connected to the following layer.

We will use a dataset called `dermatology.arff` that can be downloaded from `http://repository.seasr.org/Datasets/UCI/arff/`. This dataset contains 366 instances used to diagnosis erythemato-squamous diseases. It uses 34 attributes to classify the disease into one of five different categories. The following is a sample instance:

```
2,2,0,3,0,0,0,0,1,0,0,0,0,0,0,3,2,0,0,0,0,0,0,0,0,0,3,0,0,0,1,0,55,2
```

The last field represents the disease category. This dataset has been partitioned into two files: `dermatologyTrainingSet.arff` and `dermatologyTestingSet.arff`. The training set uses the first 80% (292 instances) of the original set and ends with line 456. The testing set is the last 20% (74 instances) and starts with line 457 of the original set (lines 457-530).

Building the model

Before we can make any predictions, it is necessary that we train the model on a representative set of data. We will use the Weka class, `MultilayerPerceptron`, for training and eventually to make predictions. First, we declare strings for the training and testing of filenames and the corresponding `FileReader` instances for them. The instances are created and the last field is specified as the field to use for classification:

```
String trainingFileName = "dermatologyTrainingSet.arff";
String testingFileName = "dermatologyTestingSet.arff";

try (FileReader trainingReader = new FileReader(trainingFileName);
        FileReader testingReader =
            new FileReader(testingFileName)) {
    Instances trainingInstances = new Instances(trainingReader);
    trainingInstances.setClassIndex(
        trainingInstances.numAttributes() - 1);
    Instances testingInstances = new Instances(testingReader);
    testingInstances.setClassIndex(
        testingInstances.numAttributes() - 1);
    ...
} catch (Exception ex) {
    // Handle exceptions
}
```

An instance of the `MultilayerPerceptron` class is then created:

```
MultilayerPerceptron mlp = new MultilayerPerceptron();
```

There are several model parameters that we can set, as shown here:

Parameter	**Method**	**Description**
Learning rate	`setLearningRate`	Affects the training speed
Momentum	`setMomentum`	Affects the training speed
Training time	`setTrainingTime`	The number of training epochs used to train the model
Hidden layers	`setHiddenLayers`	The number of hidden layers and perceptrons to use

As mentioned previously, the learning rate will affect the speed in which your model is trained. A large value can increase the training speed. If the learning rate is too small, then the training time may take too long. If the learning rate is too large, then the model may move past a local minimum and become divergent. That is, if the increase is too large, we might skip over a meaningful value. You can think of this a graph where a small dip in a plot along the *Y* axis is missed because we incremented our *X* value too much.

Momentum also affects the training speed by effectively increasing the rate of learning. It is used in addition to the learning rate to add momentum to the search for the optimal value. In the case of a local minimum, the momentum helps get out of the minimum in its quest for a global minimum.

When the model is learning it performs operations iteratively. The term, **epoch** is used to refer to the number of iterations. Hopefully, the total error encounter with each epoch will decrease to a point where further epochs are not useful. It is ideal to avoid too many epochs.

A neural network will have one or more hidden layers. Each of these layers will have a specific number of perceptrons. The `setHiddenLayers` method specifies the number of layers and perceptrons using a string. For example, *3,5* would specify two hidden layers with three and five perceptrons per layer, respectively.

For this example, we will use the following values:

```
mlp.setLearningRate(0.1);
mlp.setMomentum(0.2);
mlp.setTrainingTime(2000);
mlp.setHiddenLayers("3");
```

The `buildClassifier` method uses the training data to build the model:

```
mlp.buildClassifier(trainingInstances);
```

Evaluating the model

The next step is to evaluate the model. The `Evaluation` class is used for this purpose. Its constructor takes the training set as input and the `evaluateModel` method performs the actual evaluation. The following code illustrates this using the testing dataset:

```
Evaluation evaluation = new Evaluation(trainingInstances);
evaluation.evaluateModel(mlp, testingInstances);
```

One simple way of displaying the results of the evaluation is using the `toSummaryString` method:

```
System.out.println(evaluation.toSummaryString());
```

This will display the following output:

```
Correctly Classified Instances 73 98.6486 %
Incorrectly Classified Instances 1 1.3514 %
Kappa statistic 0.9824
Mean absolute error 0.0177
Root mean squared error 0.076
Relative absolute error 6.6173 %
Root relative squared error 20.7173 %
Coverage of cases (0.95 level) 98.6486 %
Mean rel. region size (0.95 level) 18.018 %
Total Number of Instances 74
```

Frequently, it will be necessary to experiment with these parameters to get the best results. The following are the results of varying the number of perceptrons:

Number of perceptrons	Correctly classified instances		Incorrectly classified instances	
	Number	Percentage	Number	Percentage
2	55	74.3243%	19	25.6757%
3	73	98.6486%	1	1.3514%
4	72	97.2973%	2	2.7027%
5	72	97.2973%	2	2.7027%

Predicting other values

Once we have a model trained, we can use it to evaluate other data. In the previous testing dataset there was one instance which failed. In the following code sequence, this instance is identified and the predicted and actual results are displayed.

Each instance of the testing dataset is used as input to the `classifyInstance` method. This method tries to predict the correct result. This result is compared to the last field of the instance that contains the actual value. For mismatches, the predicted and actual values are displayed:

```
for (int i = 0; i < testingInstances.numInstances(); i++) {
    double result = mlp.classifyInstance(
        testingInstances.instance(i));
    if (result != testingInstances
            .instance(i)
            .value(testingInstances.numAttributes() - 1)) {
        out.println("Classify result: " + result
                + " Correct: " + testingInstances.instance(i)
                .value(testingInstances.numAttributes() - 1));
        ...
    }
}
```

For the testing set we get the following output:

```
Classify result: 1.0 Correct: 3.0
```

We can get the likelihood of the prediction being correct using the `MultilayerPerceptron` class' `distributionForInstance` method. Place the following code into the previous loop. It will capture the incorrect instance, which is easier than instantiating an instance based on the 34 attributes used by the dataset. The `distributionForInstance` method takes this instance and returns a two element array of doubles. The first element is the probability of the result being positive and the second is the probability of it being negative:

```
Instance incorrectInstance = testingInstances.instance(i);
incorrectInstance.setDataset(trainingInstances);
double[] distribution = mlp.distributionForInstance(incorrectInstance);
out.println("Probability of being positive: " + distribution[0]);
out.println("Probability of being negative: " + distribution[1]);
```

The output for this instance is as follows:

```
Probability of being positive: 0.00350515156929017
Probability of being negative: 0.9683660500711128
```

This can provide a more quantitative feel for the reliability of the prediction.

Saving and retrieving the model

We can also save and retrieve a model for later use. To save the model, build the model and then use the `SerializationHelper` class' static method `write`, as shown in the following code snippet. The first argument is the name of the file to hold the model:

```
SerializationHelper.write("mlpModel", mlp);
```

To retrieve the model, use the corresponding `read` method as shown here:

```
mlp = (MultilayerPerceptron)SerializationHelper.read("mlpModel");
```

Next, we will learn how to use another useful neural network approach, SOMs.

Learning vector quantization

Learning Vector Quantization (**LVQ**) is another special type of a dynamic ANN. SOMs, which we will discuss in a moment, are a by-product of LVQ networks. This type of network implements a competitive type of algorithm in which the winning neuron gains the weight. These types of networks are used in many different applications and are considered to be more natural and intuitive than some other ANNs. In particular, LVQ is effective for classification of text-based data.

The basic algorithm begins by setting the number of neurons, the weight for each neuron, how fast the neurons can learn, and a list of input vectors. In this context, a vector is similar to a vector in physics and represents the values provided to the input layer neurons. As the network is trained, a vector is used as input, a winning neuron is selected, and the weight of the winning neuron is updated. This model is iterative and will continue to run until a solution is found.

Self-Organizing Maps

SOMs is a technique that takes multidimensional data and reducing it to one or two dimensions. This compression technique is called **vector quantization**. The technique usually involves a visual component that allows a human to better see how the data has been categorized. SOM learns without supervision.

The SOM is good for finding clusters, which is not to be confused with classification. With classification we are interested in finding the best fit for a data instance among predefined categories. With clustering we are interested in grouping instances where the categories are unknown.

A SOM uses a lattice of neurons, usually a two-dimensional array or a hexagonal grid, representing neurons that are assigned weights. The input sources are connected to each of these neurons. The technique then adjusts the weights assigned to each lattice member through several iterations until the best fit is found. When finished, the lattice members will have grouped the input dataset into categories. The SOM results can be viewed to identify categories and map new input to one of the identified categories.

Using a SOM

We will use the Weka to demonstrate SOM. However, it is does not come installed with standard Weka. Instead, we will need to download a set of Weka classification algorithms from `https://sourceforge.net/projects/wekaclassalgos/files/` and the actual SOM class from `http://www.cis.hut.fi/research/som_pak/`. The classification algorithms include support for LVQ. More details about the classification algorithms can be found at `http://wekaclassalgos.sourceforge.net/`.

To use the SOM class, called `SelfOrganizingMap`, the source code needs to be in your project. The Javadoc for this class is found at `http://jsalatas.ictpro.gr/weka/doc/SelfOrganizingMap/`.

We start with the creation of an instance of the `SelfOrganizingMap` class. This is followed by code to read in data and create an `Instances` object to hold the data. In this example, we will use the `iris.arff` file, which can be found in the Weka data directory. Notice that once the `Instances` object is created we do not specify the class index as we did with previous Weka examples since SOM uses unsupervised learning:

```
SelfOrganizingMap som = new SelfOrganizingMap();
String trainingFileName = "iris.arff";
try (FileReader trainingReader =
        new FileReader(trainingFileName)) {
```

```
        Instances trainingInstances = new Instances(trainingReader);
        ...
    } catch (IOException ex) {
        // Handle exceptions
    } catch (Exception ex) {
        // Handle exceptions
    }
```

The `buildClusterer` method will execute the SOM algorithm using the training dataset:

```
        som.buildClusterer(trainingInstances);
```

Displaying the SOM results

We can now display the results of the operation as follows:

```
        out.println(som.toString());
```

The `iris` dataset uses five attributes: `sepallength`, `sepalwidth`, `petallength`, `petalwidth`, and `class`. The first four attributes are numeric and the fifth has three possible values: `Iris-setosa`, `Iris-versicolor`, and `Iris-virginica`. The first part of the abbreviated output that follows identified four clusters and the number of instances in each cluster. This is followed by statistics for each of the attributes:

```
Self Organized Map
==================
Number of clusters: 4
Cluster
Attribute 0 1 2 3
(50) (42) (29) (29)
=============================================
sepallength
value 5.0036 6.2365 5.5823 6.9513
min 4.3 5.6 4.9 6.2
max 5.8 7 6.3 7.9
mean 5.006 6.25 5.5828 6.9586
std. dev. 0.3525 0.3536 0.3675 0.5046
...
class
value 0 1.5048 1.0787 2
min 0 1 1 2
max 0 2 2 2
mean 0 1.4524 1.069 2
std. dev. 0 0.5038 0.2579 0
```

These statistics can provide insight into the dataset. If we are interested in determining which dataset instance is found in a cluster, we can use the `getClusterInstances` method to return the array that groups the instances by cluster. As shown next, this method is used to list the instance by cluster:

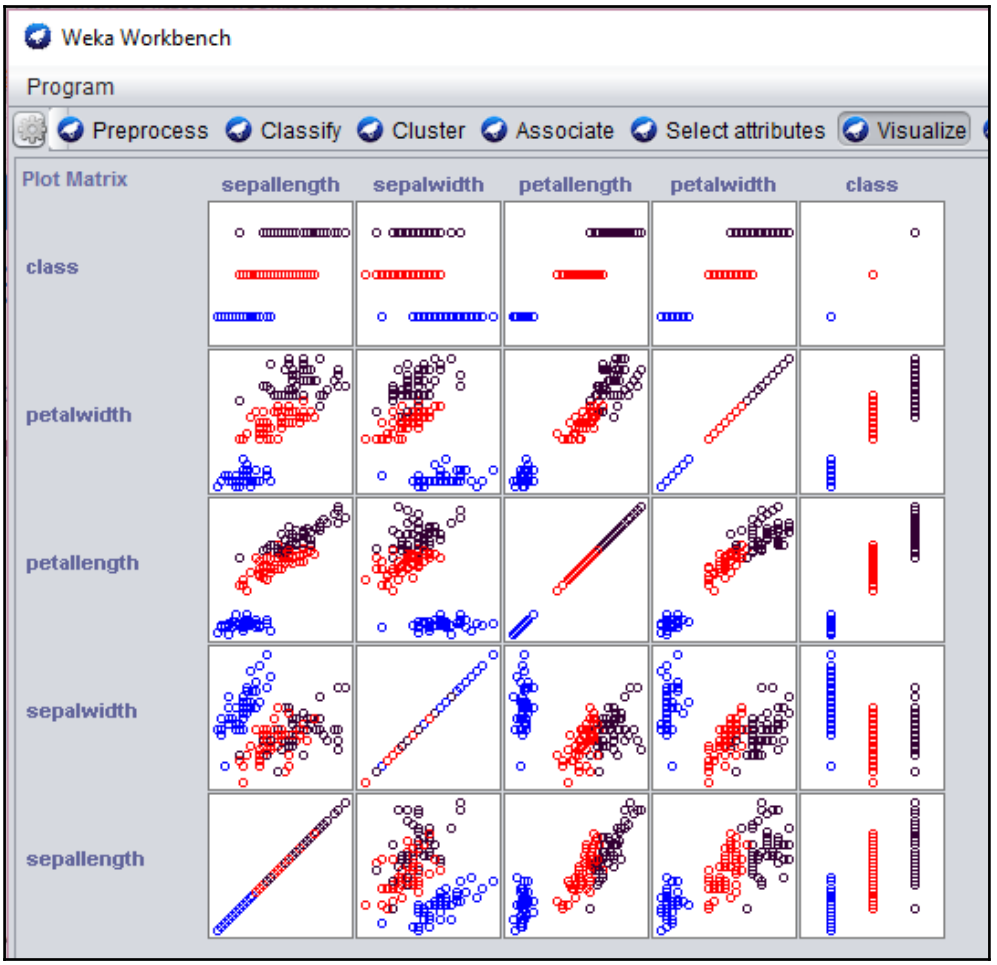
```
Instances[] clusters = som.getClusterInstances();
int index = 0;
for (Instances instances : clusters) {
    out.println("-------Custer " + index);
    for (Instance instance : instances) {
        out.println(instance);
    }
    out.println();
    index++;
}
```

As we can see with the abbreviated output of this sequence, different `iris` classes are grouped into the different clusters:

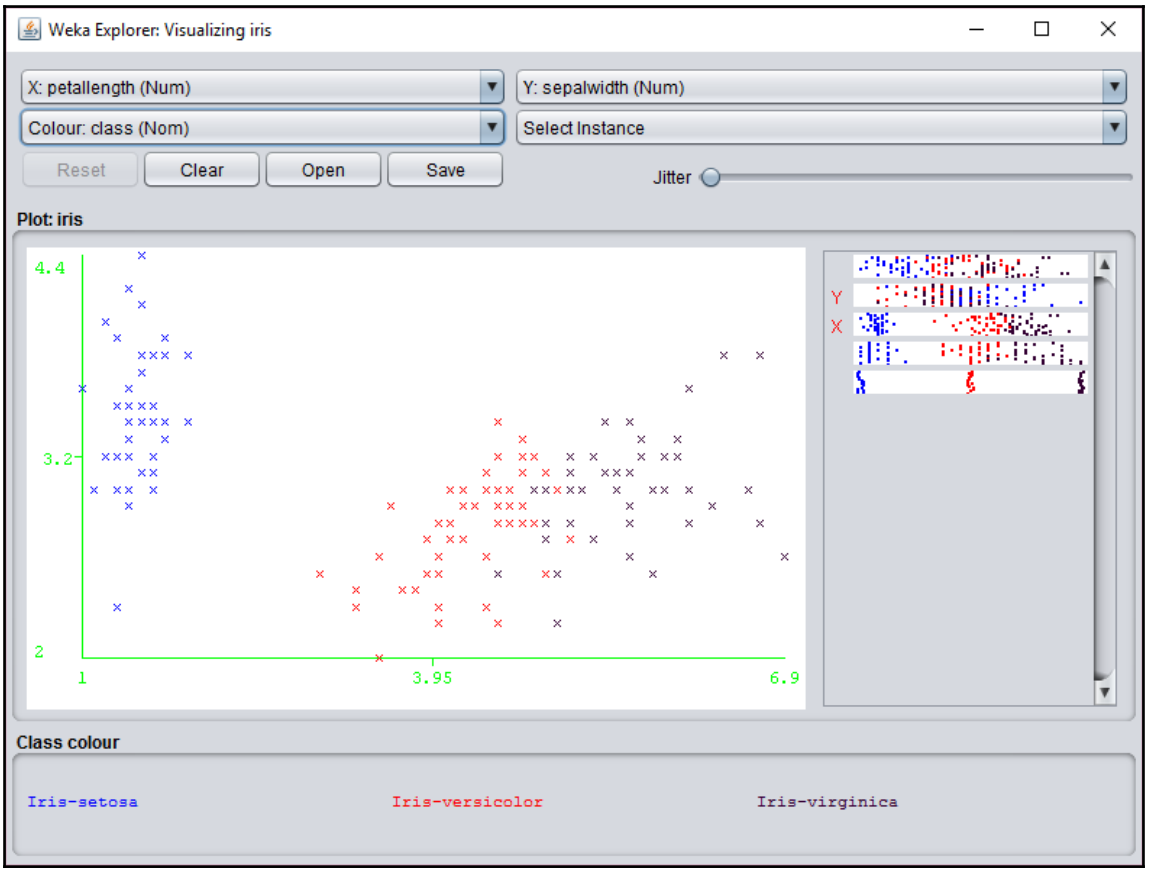
```
-------Custer 0
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
...
5.3,3.7,1.5,0.2,Iris-setosa
5,3.3,1.4,0.2,Iris-setosa

-------Custer 1
7,3.2,4.7,1.4,Iris-versicolor
6.4,3.2,4.5,1.5,Iris-versicolor
6.9,3.1,4.9,1.5,Iris-versicolor
...
6.5,3,5.2,2,Iris-virginica
5.9,3,5.1,1.8,Iris-virginica

-------Custer 2
5.5,2.3,4,1.3,Iris-versicolor
5.7,2.8,4.5,1.3,Iris-versicolor
4.9,2.4,3.3,1,Iris-versicolor
...
4.9,2.5,4.5,1.7,Iris-virginica
6,2.2,5,1.5,Iris-virginica

-------Custer 3
6.3,3.3,6,2.5,Iris-virginica
7.1,3,5.9,2.1,Iris-virginica
```

```
6.5,3,5.8,2.2,Iris-virginica
...
```

The cluster results can be displayed visually using the Weka GUI interface. In the following screenshot, we have used the **Weka Workbench** to analyze and visualize the result of the SOM analysis:



An individual section of the graph can be selected, customized, and analyzed as follows:



However, before you can use the `SOM` class, the `WekaPackageManager` must be used to add the `SOM` package to Weka. This process is discussed at `https://weka.wikispaces.com/How+do+I+use+the+package+manager%3F`.

If a new instance needs to be mapped to a cluster, the `distributionForInstance` method can be used as illustrated in *Predicting other values* section.

Additional network architectures and algorithms

We have discussed a few of the most common and practical neural networks. At this point, we would also like to consider some specialized neural networks and their application in various fields of study. These types of networks do not fit neatly into one particular category, but may still be of interest.

The k-Nearest Neighbors algorithm

An artificial neural network implementing the k-NN algorithm is similar to MLP networks, but it provides significant reduction in time compared to the winner takes all strategy. This type of network does not require a training algorithm after the initial weights are set and has fewer connections among its neurons. We have chosen not to provide an example of this algorithm's implementation because its use in Weka is very similar to the MLP example.

This type of network is best suited to classification tasks. Because it utilizes lazy learning techniques, reserving all computation until after information has been classified, it is considered to be one of the simpler models. In this model, the neurons are weighted based upon their distance from their neighbors. The classification of the neighbors is already known and therefore no specific training is required.

Instantaneously trained networks

Instantaneously Trained Neural Networks (**ITNNs**) are feedforward ANNs. They are special because they add a new hidden neuron for every unique set of training data. The main advantage to this type of network is the ability to provide generalization to other problems.

ITNNs are especially useful in short term learning situations. In particular, this type of network is useful for web searches and other pattern recognition functions with large datasets. These networks are suited for time series prediction and other deep learning purposes.

Spiking neural networks

A **Spiking Neural Network** (**SNN**) is a more complex ANN due to the fact it takes into account not only the neuron and information propagation, but also the timing of each event. In these networks, every neuron does not fire during every propagation of information, but rather only when the **membrane potential** for a particular neuron reaches a specific threshold. The membrane potential refers to the activation level of a neuron and closely resembles the way biological neurons fire.

Due to the close mimicry of biological neural networks, SNNs are especially suited to biological study and application. They have been used to model the nervous system of animals and insects and are useful for predicting the outcome of various stimuli. These networks have the ability to create very complex models with significant detail, but sacrifice time to accomplish this goal.

Cascading neural networks

Cascading Neural Networks (**CNNs**) is a specialized supervised learning algorithm. In this type, the network is initially very small and simplistic. As the network learns, it gradually adds new hidden units. Once the node is added, its input weight is constant and cannot be changed or removed.

This type of neural network is praised for its quick learning rate and ability to dynamically build itself. The user of such a network does not have to worry about topological design. Additionally, these networks do not require backpropagation of error information to make adjustments.

Holographic associative memory

Holographic Associative Memory (**HAM**) is a special type of complex neural network. This is a specialized type of network related to natural human memory and visual analysis. This network is especially useful for pattern recognition and associative memory tasks and can be applied to optical computations.

HAM attempts to closely mimic human visualization and pattern recognition. In this network, stimulus-response patterns are learned without iteration and backpropagation of errors is not required. Unlike other networks discussed in this chapter, HAM does not exhibit the same type of connected behaviour. Instead, the stimulus-response patterns can be stored within a single neuron.

Backpropagation and neural networks

Backpropagation algorithms are another supervised learning techniques used to train neural networks. As the name suggests, this algorithm calculates the computed output error and then changes the weights of each neuron in a backwards manner. Backpropagation is primarily used with MLP networks. It is important to note forward propagation must occur before backward propagation can be used.

In its most basic form, this algorithm consists of four steps:

1. Perform forward propagation for a given set of inputs.
2. Calculate the error value for each output.
3. Change the weights based upon the calculated error for each node.
4. Perform forward propagation again.

This algorithm completes when the output matches the expected output.

Summary

In this chapter, we have provided a broad overview of artificial neural networks, as well as a detailed examination of a few specific implementations. We began with a discussion of the basic properties of neural networks, training algorithms, and neural network architectures.

Next we provided an example of a simple static neural network implementing the XOR problem using Java. This example provided detailed explanation of the code used to build and train the network, including some of the math behind the weight adjustments during the training process. We then discussed dynamic neural networks and provided two in-depth examples, the MLP and SOM networks. These used the Weka tools to create and train the networks.

Finally, we concluded our chapter with a discussion of additional network architectures and algorithms. We chose some of the more popular networks to summarize and explored situations where each type would be most useful. We also included a discussion of backpropagation in this section.

In the next chapter, we will expand upon this introduction and take a look at deep learning with neural networks.

8
Deep Learning

In this chapter, we will focus on neural networks, often referred to as **Deep Learning Networks** (**DLNs**). This type of network is characterized as a multiple-layer neural network. Each of these layers are rained on the output of the previous layer, potentially identifying features and sub-features of the dataset. A feature hierarchy is created in this manner.

DLNs typically work with unstructured and unlabeled data, which constitute the vast bulk of data found in the world today. DLN will take this unstructured data, identify features, and try to reconstruct the original input. This approach is illustrated with **Restricted Boltzmann Machines** (**RBMs**) in *Restricted Boltzmann Machines* and with autoencoders in *Deep autoencoders*. An autoencoder takes a dataset and effectively compresses it. It then decompresses it to reconstruct the original dataset.

DLN can also be used for predictive analysis. The last step of a DLN will use an activation function to generate output represented by one of several categories. When used with new data, the model will attempt to classify the input based on the previously trained model.

An important DLN task is ensuring that the model is accurate and minimizes error. As with simple neural networks, weights and biases are used at each layer. As weight values are adjusted, errors can be introduced. A technique to adjust weights uses **gradient descent**. This can be thought of as the slope of the change. The idea is to modify the weight so as to minimize the error. It is an optimization technique that speeds up the learning process.

Later in the chapter, we will examine **Convolutional Neural Networks** (**CNNs**) and briefly discuss **Recurrent Neural Networks** (**RNN**). Convolution networks mimic the visual cortex in that each neuron can interact with and make decisions **based** on a region of information. Recurrent networks process information based on not only the output of the previous layer but also the calculations performed in previous layers.

There are several libraries that support deep learning, including these:

- **N-Dimensional Arrays for Java** (**ND4J**) `http://nd4j.org/`: A scientific computing library intended for production use
- **Deeplearning4j** (`http://deeplearnin4j.org/`): An open source, distributed deep-learning library
- **Encog** (`http://www.heatonresearch.com/encog/`): This library supports several deep learning algorithms

ND4J is a lower level library that is actually used in other projects, including DL4J. Encog is perhaps not as well supported as DL4J, but does provide support for deep learning.

The examples used in this chapter are all based on the **Deep Learning for Java** (**DL4J**) (`http://deeplearning4j.org`) API with support from ND4J. This library provides good support for many of the algorithms associated with deep learning. As a result, the next section explains the basic tasks found in common with many of the deep learning algorithms, such as loading data, training a model, and testing the model.

Deeplearning4j architecture

In this section, we will discuss its architecture and address several of the common tasks performed when using the API. DLN typically starts with the creation of a `MultiLayerConfiguration` instance, which defines the network, or model. The network is composed of multiple layers. **Hyperparameters** are used to configure the network and are variables that affect such things as learning speed, activation functions to use for a layer, and how weights are to be initialized.

As with neural networks, the basic DLN process consists of:

- Acquiring and manipulating data
- Configuring and building a model
- Training the model
- Testing the model

We will investigate each of these tasks in the next sections.



The code examples in this section are not intended to be entered and executed here. Instead, these examples are snippets out of later models that we will be using.

Acquiring and manipulating data

The DL4J API has a number of techniques for acquiring data. We will focus on those specific techniques that we will use in our examples. The dataset used by a DL4J project is often modified using either **binarization** or **normalization**. Binarization converts data to ones and zeroes. Normalization converts data to a value between *1* and *0*.

Data feed to DLN is transformed to a set of numbers. These numbers are referred to as **vectors**. These vectors consist of a one-column matrix with a variable number of rows. The process of creating a vector is called **vectorization**.

Canova (`http://deeplearning4j.org/canova.html`) is a DL4J library that supports vectorization. It works with many different types of datasets. It has been merged with **DataVec** (`http://deeplearning4j.org/datavec`), a vectorization and **Extract, Transform, and Load** (**ETL**) library.

In this section, we will focus on how to read in CSV data.

Reading in a CSV file

ND4J provides the `CSVRecordReader` class, which is useful for reading CSV data. It has three overloaded constructors. The one we will demonstrate is passed two arguments. The first is the number of lines to skip when first reading a file and the second is a string holding the delimiters used to parse the text.

In the following code, we create a new instance of the class, where we do not skip any lines and use only a comma for a delimiter:

```
RecordReader recordReader = new CSVRecordReader(0, ",");
```

The class implements the `RecordReader` interface. It has an `initialize` method that is passed an instance of the `FileSplit` class. One of its constructors is passed an instance of a `File` object that references a dataset. The `FileSplit` class assists in splitting the data for training and testing. In this example, we initialize the reader for a file called `car.txt` that we will use in the *Preparing the data* section:

```
recordReader.initialize(new FileSplit(new File("car.txt")));
```

To process the data, we need an iterator such as the `DataSetIterator` instance shown next. This class possesses a multitude of overloaded constructors. In the following example, the first argument is the `RecordReader` instance. This is followed by three arguments. The first is the batch size, which is the number of records to retrieve at a time. The next one is the index of the last attribute of the record. The last argument is the number of classes represented by the dataset:

```
DataSetIterator iterator =
    new RecordReaderDataSetIterator(recordReader, 1728, 6, 4);
```

The file's record's last attribute will hold a class value if we use a dataset for regression. This is precisely how we will use it later. The number of the class's parameter is only used with regression.

In the next code sequence, we will split the dataset into two sets: one for training and one for testing. Starting with the `next` method, this method returns the next dataset from the source. The size of the dataset is dependent on the batch size used earlier. The `shuffle` method randomizes the input while the `splitTestAndTrain` method returns an instance of the `SplitTestAndTrain` class, which we use to get the training and testing datasets. The `splitTestAndTrain` method's argument specifies the percentage of the data to be used for training.

```
DataSet dataset = iterator.next();
dataset.shuffle();
SplitTestAndTrain testAndTrain = dataset.splitTestAndTrain(0.65);
DataSet trainingData = testAndTrain.getTrain();
DataSet testData = testAndTrain.getTest();
```

We can then use these datasets with a model.

Configuring and building a model

Frequently, DL4J uses the `MultiLayerConfiguration` class to define the configuration of the model and the `MultiLayerNetwork` class to represent a model. These classes provide a flexible way of building models.

In the following example, we will demonstrate the use of these classes. Starting with the `MultiLayerConfiguration` class, we find that several methods are used in a fluent style. We will provide more details about these methods shortly. However, notice that two layers are defined for this model:

```
MultiLayerConfiguration conf =
    new NeuralNetConfiguration.Builder()
```

```
.iterations(1000)
.activation("relu")
.weightInit(WeightInit.XAVIER)
.learningRate(0.4)
.list()
.layer(0, new DenseLayer.Builder()
        .nIn(6).nOut(3)
        .build())
.layer(1, new OutputLayer
        .Builder(LossFunctions.LossFunction
                .NEGATIVELOGLIKELIHOOD)
        .activation("softmax")
        .nIn(3).nOut(4).build())
.backprop(true).pretrain(false)
.build();
```

The `nIn` and `nOut` methods specify the number of inputs and outputs for a layer.

Using hyperparameters in ND4J

Builder classes are common in DL4J. In the previous example, the `NeuralNetConfiguration.Builder` class is used. The methods used here are but a few of the many that are available. In the following table, we describe several of them:

Method	**Usage**
`iterations`	Controls the number of optimization iterations performed
`activation`	This is the activation function used
`weightInit`	Used to initialize the initial weights for the model
`learningRate`	Controls the speed the model learns
`List`	Creates an instance of the `NeuralNetConfiguration.ListBuilder` class so that we can add layers
`Layer`	Creates a new layer
`backprop`	When set to true, it enables backpropagation
`pretrain`	When set to true, it will pretrain the model
`Build`	Performs the actual build process

Let's examine how a layer is created more closely. In the example, the `list` method returns a `NeuralNetConfiguration.ListBuilder` instance. Its `layer` method takes two arguments. The first is the number of the layer, a zero-based numbering scheme. The second is the `Layer` instance.

There are two different layers used here with two different builders: a `DenseLayer.Builder` and an `OutputLayer.Builder` instance. There are several types of layers available in DL4J. The argument of a builder's constructor may be a **loss function**, as is the case with the output layer, and is explained next.

In a feedback network, the neural network's guess is compared to what is called the **ground truth**, which is the error. This error is used to update the network through the modification of weights and biases. The loss function, also called an **objective** or **cost function**, measures the difference.

There are several loss functions supported by DL4J:

- `MSE`: In linear regression MSE stands for mean squared error
- `EXPLL`: In poisson regression EXPLL stands for exponential log likelihood
- `XENT`: In binary classification XENT stands for cross entropy
- `MCXENT`: This stands for multiclass cross entropy
- `RMSE_XENT`: This stands for RMSE cross entropy
- `SQUARED_LOSS`: This stands for squared loss
- `RECONSTRUCTION_CROSSENTROPY`: This stands for reconstruction cross entropy
- `NEGATIVELOGLIKELIHOOD`: This stands for negative log likelihood
- `CUSTOM`: Define your own loss function

The remaining methods used with the builder instance are the activation function, the number of inputs and outputs for the layer, and the `build` method, which creates the layer.

Each layer of a multi-layer network requires the following:

- **Input**: Usually in the form of an input vector
- **Weights**: Also called coefficients
- **Bias**: Used to ensure that at least some nodes in a layer are activated
- **Activation function**: Determines whether a node fires

There are many different types of activation functions, each of which can address a particular type of problem.

The activation function is used to determine whether the neuron fires. There are several functions supported, including `relu` (rectified linear), `tanh`, `sigmoid`, `softmax`, `hardtanh`, `leakyrelu`, `maxout`, `softsign`, and `softplus`.



An interesting list of activation functions along with graphs is found at `http://stats.stackexchange.com/questions/115258/comprehensive-list-of-activation-functions-in-neural-networks-with-pros-cons` and `https://en.wikipedia.org/wiki/Activation_function`.

Instantiating the network model

Next, a `MultiLayerNetwork` instance is created using the defined configuration. The model is initialized, and its listeners are set. The `ScoreIterationListener` instance will display information as the model trains, which we will see shortly. Its constructor's argument specifies how often that information should be displayed:

```
MultiLayerNetwork model = new MultiLayerNetwork(conf);
model.init();
model.setListeners(new ScoreIterationListener(100));
```

We are now ready to train the model.

Training a model

This is actually a fairly simple step. The `fit` method performs the training:

```
model.fit(trainingData);
```

When executed, the output will be generated using any listeners associated with the model, as is the preceding case, where a `ScoreIterationListener` instance is used.

Another example of how the `fit` method is used is through the process of iterating through a dataset, as shown next. In this example, a sequence of datasets is used. This is the part of an autoencoder where the output is intended to match the input, as explained in *Deep autoencoders* section. The dataset used as the argument to the `fit` method uses both the input and the expected output. In this case, they are the same as provided by the `getFeatureMatrix` method:

```
while (iterator.hasNext()) {
```

```
        DataSet dataSet = iterator.next();
        model.fit(new DataSet(dataSet.getFeatureMatrix(),
                dataSet.getFeatureMatrix()));
    }
```

For larger datasets, it is necessary to pretrain the model several times to get accurate results. This is often performed in parallel to reduce training time. This option is set with a layer class's `pretrain` method.

Testing a model

The evaluation of a model is performed using the `Evaluation` class and the training dataset. An `Evaluation` instance is created using an argument specifying the number of classes. The test data is fed into the model using the `output` method. The `eval` method takes the output of the model and compares it against the test data classes to generate statistics:

```
Evaluation evaluation = new Evaluation(4);
INDArray output = model.output(testData.getFeatureMatrix());
evaluation.eval(testData.getLabels(), output);
out.println(evaluation.stats());
```

The output will look similar to the following:

```
==========================Scores========================================
Accuracy: 0.9273
Precision: 0.854
Recall: 0.8323
F1 Score: 0.843
```

These statistics are detailed here:

- `Accuracy`: This is a measure of how often the correct answer was returned.
- `Precision`: This is a measure of the probability that a positive response is correct.
- `Recall`: This measures how likely the result will be classified correctly if given a positive example.
- `F1 Score`: This is the probability that the network's results are correct. It is the harmonic mean of recall and precision. It is calculated by dividing the number of true positives by the sum of true positives and false negatives.

We will use the `Evaluation` class to determine the quality of our model. A measure called **f1** is used, whose values range from *0* to *1*, where *1* represents the best quality.

Deep learning and regression analysis

Neural networks can be used to perform regression analysis. However, other techniques (see the early chapters) may offer a more effective solution. With regression analysis, we want to predict a result based on several input variables.

We can perform regression analysis using an output layer that consists of a single neuron that sums the weighted input plus bias of the previous hidden layer. Thus, the result is a single value representing the regression.

Preparing the data

We will use a car evaluation database to demonstrate how to predict the acceptability of a car based on a series of attributes. The file containing the data we will be using can be downloaded from: `http://archive.ics.uci.edu/ml/machine-learning-databases/car/car.data`. It consists of car data such as price, number of passengers, and safety information, and an assessment of its overall quality. It is this latter element, quality, that we will try to predict. The comma-delimited values in each attribute are shown next, along with substitutions. The substitutions are needed because the model expects numeric data:

Attribute	**Original value**	**Substituted value**
Buying price	`vhigh, high, med, low`	`3,2,1,0`
Maintenance price	`vhigh, high, med, low`	`3,2,1,0`
Number of doors	`2, 3, 4, 5-more`	`2,3,4,5`
Seating	`2, 4, more`	`2,4,5`
Cargo space	`small, med, big`	`0,1,2`
Safety	`low, med, high`	`0,1,2`

There are 1,728 instances in the file. The cars are marked with four classes:

Class	Number of instances	Percentage of instances	Original value	Substituted value
Unacceptable	1210	70.023%	unacc	0
Acceptable	384	22.222%	acc	1
Good	69	3.99%	good	2
Very good	65	3.76%	v-good	3

Setting up the class

We start with the definition of a `CarRegressionExample` class, as shown next, where an instance of the class is created and where the work is performed within its default constructor:

```
public class CarRegressionExample {

    public CarRegressionExample() {
        try {
            ...
        } catch (IOException | InterruptedException ex) {
            // Handle exceptions
        }
    }

    public static void main(String[] args) {
        new CarRegressionExample();
    }
}
```

Reading and preparing the data

The first task is to read in the data. We will use the `CSVRecordReader` class to get the data, as explained in *Reading in a CSV file*:

```
RecordReader recordReader = new CSVRecordReader(0, ",");
recordReader.initialize(new FileSplit(new File("car.txt")));
DataSetIterator iterator = new
  RecordReaderDataSetIterator(recordReader, 1728, 6, 4);
```

With this dataset, we will split the data into two sets. Sixty five percent of the data is used for training and the rest for testing:

```
DataSet dataset = iterator.next();
dataset.shuffle();
SplitTestAndTrain testAndTrain = dataset.splitTestAndTrain(0.65);
DataSet trainingData = testAndTrain.getTrain();
DataSet testData = testAndTrain.getTest();
```

The data now needs to be normalized:

```
DataNormalization normalizer = new NormalizerStandardize();
normalizer.fit(trainingData);
normalizer.transform(trainingData);
normalizer.transform(testData);
```

We are now ready to build the model.

Building the model

A `MultiLayerConfiguration` instance is created using a series of `NeuralNetConfiguration.Builder` methods. The following is the dice used. We will discuss the individual methods following the code. Note that this configuration uses two layers. The last layer uses the `softmax` activation function, which is used for regression analysis:

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .iterations(1000)
        .activation("relu")
        .weightInit(WeightInit.XAVIER)
        .learningRate(0.4)
        .list()
        .layer(0, new DenseLayer.Builder()
                .nIn(6).nOut(3)
                .build())
        .layer(1, new OutputLayer
                .Builder(LossFunctions.LossFunction
                        .NEGATIVELOGLIKELIHOOD)
                .activation("softmax")
                .nIn(3).nOut(4).build())
        .backprop(true).pretrain(false)
        .build();
```

Two layers are created. The first is the input layer. The `DenseLayer.Builder` class is used to create this layer. The `DenseLayer` class is a feed-forward and fully connected layer. The created layer uses the six car attributes as input. The output consists of three neurons that are fed into the output layer and is duplicated here for your convenience:

```
.layer(0, new DenseLayer.Builder()
        .nIn(6).nOut(3)
        .build())
```

The second layer is the output layer created with the `OutputLayer.Builder` class. It uses a loss function as the argument of its constructor. The `softmax` activation function is used since we are performing regression as shown here:

```
.layer(1, new OutputLayer
        .Builder(LossFunctions.LossFunction
                .NEGATIVELOGLIKELIHOOD)
        .activation("softmax")
        .nIn(3).nOut(4).build())
```

Next, a `MultiLayerNetwork` instance is created using the configuration. The model is initialized, its listeners are set, and then the `fit` method is invoked to perform the actual training. The `ScoreIterationListener` instance will display information as the model trains which we will see shortly in the output of this example. The `ScoreIterationListener` constructor's argument specifies the frequency that information is displayed:

```
MultiLayerNetwork model = new MultiLayerNetwork(conf);
model.init();
model.setListeners(new ScoreIterationListener(100));
model.fit(trainingData);
```

We are now ready to evaluate the model.

Evaluating the model

In the next sequence of code, we evaluate the model against the training dataset. An `Evaluation` instance is created using an argument specifying that there are four classes. The test data is fed into the model using the `output` method. The `eval` method takes the output of the model and compares it against the test data classes to generate statistics. The `getLabels` method returns the expected values:

```
Evaluation evaluation = new Evaluation(4);
INDArray output = model.output(testData.getFeatureMatrix());
evaluation.eval(testData.getLabels(), output);
```

```
out.println(evaluation.stats());
```

The output of the training follows, which is produced by the `ScoreIterationListener` class. However, the values you get may differ due to how the data is selected and analyzed. Notice that the score improves with the iterations but levels out after about 500 iterations:

```
12:43:35.685 [main] INFO o.d.o.l.ScoreIterationListener - Score at
iteration 0 is 1.443480901811554
12:43:36.094 [main] INFO o.d.o.l.ScoreIterationListener - Score at
iteration 100 is 0.3259061845624861
12:43:36.390 [main] INFO o.d.o.l.ScoreIterationListener - Score at
iteration 200 is 0.2630572026049783
12:43:36.676 [main] INFO o.d.o.l.ScoreIterationListener - Score at
iteration 300 is 0.24061281470878784
12:43:36.977 [main] INFO o.d.o.l.ScoreIterationListener - Score at
iteration 400 is 0.22955121170274934
12:43:37.292 [main] INFO o.d.o.l.ScoreIterationListener - Score at
iteration 500 is 0.22249920540161677
12:43:37.575 [main] INFO o.d.o.l.ScoreIterationListener - Score at
iteration 600 is 0.2169898450109222
12:43:37.872 [main] INFO o.d.o.l.ScoreIterationListener - Score at
iteration 700 is 0.21271599814600958
12:43:38.161 [main] INFO o.d.o.l.ScoreIterationListener - Score at
iteration 800 is 0.2075677126088741
12:43:38.451 [main] INFO o.d.o.l.ScoreIterationListener - Score at
iteration 900 is 0.20047317735870715
```

This is followed by the results of the `stats` method as shown next. The first part reports on how examples are classified and the second part displays various statistics:

```
Examples labeled as 0 classified by model as 0: 397 times
Examples labeled as 0 classified by model as 1: 10 times
Examples labeled as 0 classified by model as 2: 1 times
Examples labeled as 1 classified by model as 0: 8 times
Examples labeled as 1 classified by model as 1: 113 times
Examples labeled as 1 classified by model as 2: 1 times
Examples labeled as 1 classified by model as 3: 1 times
Examples labeled as 2 classified by model as 1: 7 times
Examples labeled as 2 classified by model as 2: 21 times
Examples labeled as 2 classified by model as 3: 14 times
Examples labeled as 3 classified by model as 1: 2 times
Examples labeled as 3 classified by model as 3: 30 times
==========================Scores========================================Accuracy
: 0.9273
Precision: 0.854
Recall: 0.8323
F1 Score: 0.843
========================================================================
```

The regression model does a reasonable job with this dataset.

Restricted Boltzmann Machines

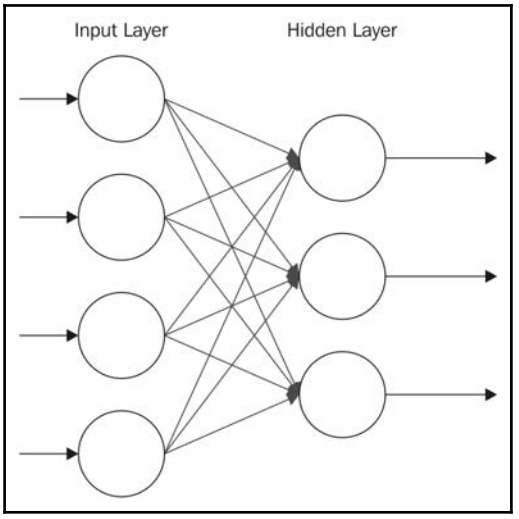
RBM is often used as part of a multi-layer deep belief network. The output of the RBM is used as an input to another layer. The use of the RBM is repeated until the final layer is reached.



Deep Belief Networks (**DBNs**) consist of several RBMs stacked together. Each hidden layer provides the input for the subsequent layer. Within each layer, the nodes cannot communicate laterally and it becomes essentially a network of other single-layer networks. DBNs are especially helpful for classifying, clustering, and recognizing image data.

The term, **continuous restricted Boltzmann machine**, refers an RBM that uses values other than integers. Input data is normalized to values between zero and one.

Each node of the input layer is connected to each node of the second layer. No nodes of the same layer are connected to each other. That is, there is no intra-layer communication. This is what restricted means.



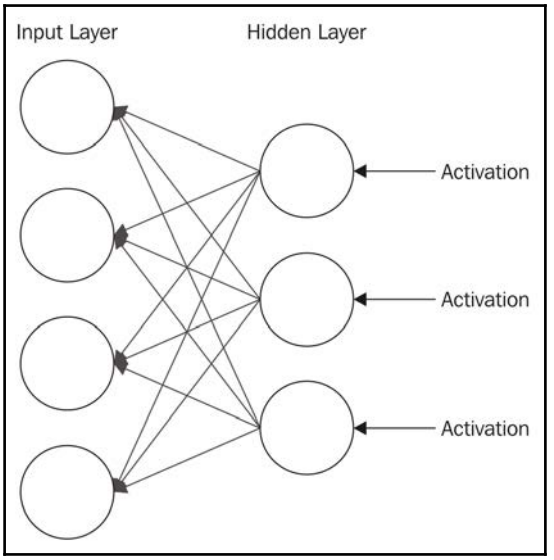
The number of input nodes for the visible layer is dependent on the problem being solved. For example, if we are looking at an image with *256* pixels, then we will need *256* input nodes. For an image, this is the number of rows times the number of columns for the image.

The **Hidden Layer** should contain fewer neurons than the **Input Layer**. Using close to the same number of neurons will sometimes result in the construction of an identity function. Too many neurons may result in overfitting. This means that datasets with a large number of inputs will require multiple layers. Smaller input sizes result in the need for fewer layers.

Stochastic, that is, random, values are assigned to each node's weights. The value for a node is multiplied by its weight and then added to a bias. This value, combined with the combined input from the other input nodes, is then fed into the activation function, where an output value is generated.

Reconstruction in an RBM

The RBM technique goes through a reconstruction phase. This is where the activations are fed back to the first layer and multiplied by the same weights used for the input. The sum of these values from each node of the second layer, plus another bias, represents an approximation of the original input. The idea is to train the model to minimize the difference between the original input values and the feedback values.



The difference in values is treated as an error. The process is repeated until an error minimum is reached. You can think of the reconstruction as guesses about the original input. These guesses are essentially a probability distribution of the original input. This is called generative learning, in contrast to discriminative learning, which occurs with classification techniques.

In a multi-layer model, each layer can be used to essentially identify a feature. In subsequent layers, a combination of features may be identified or generated. In this way, a seemingly random set of pixel values may be analyzed to identify the veins of a leaf, a leaf, a trunk, and then a tree.

The output of an RBM is a value that essentially represents a percentage. If it is not zero, then the machine has learned something about the input.

Configuring an RBM

We will examine two different RBM configurations. The first one is minimal and we will see it again in *Deep autoencoders*. The second uses several additional methods and provides more insights into the various ways it can be configured.

The following statement creates a new layer using the `RBM.Builder` class. The input is computed based on the number of rows and columns of an image. The output is large, containing `1000` neurons. The loss function is `RMSE_XENT`. This loss function works better for some classification problems:

```
.layer(0, new RBM.Builder()
    .nIn(numRows * numColumns).nOut(1000)
    .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
    .build())
```

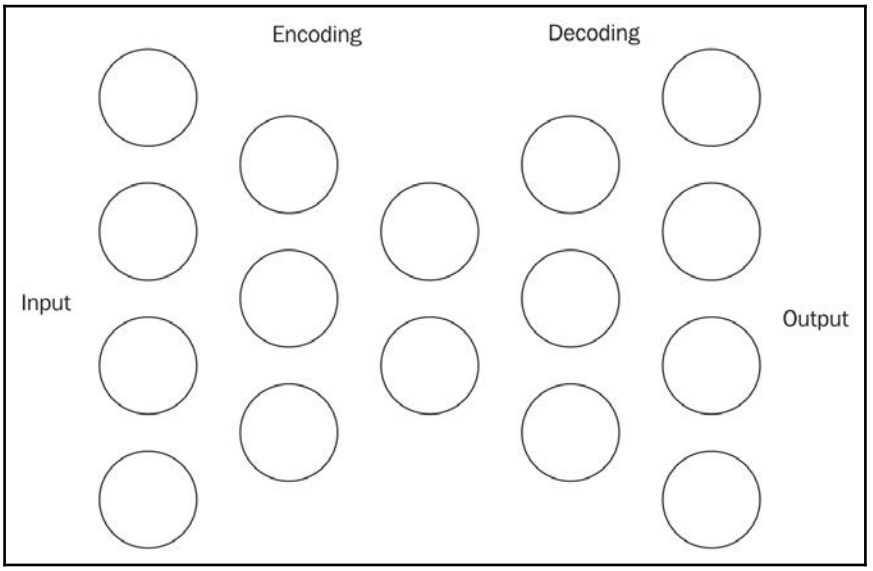
Next is a more complex RBM. We will not detail each of these methods here but will see them used in later examples:

```
.layer(new RBM.Builder()
    .l2(1e-1).l1(1e-3)
    .nIn(numRows * numColumns
    .nOut(outputNum)
    .activation("relu")
    .weightInit(WeightInit.RELU)
    .lossFunction(LossFunctions.LossFunction
        .RECONSTRUCTION_CROSSENTROPY).k(3)
    .hiddenUnit(HiddenUnit.RECTIFIED)
    .visibleUnit(VisibleUnit.GAUSSIAN)
    .updater(Updater.ADAGRAD)
        .gradientNormalization(
             GradientNormalization.ClipL2PerLayer)
    .build())
```

A single-layer `RBM` is not always useful. A multi-layer autoencoder is often required. We will look at this option in the next section.

Deep autoencoders

An autoencoder is used for feature selection and extraction. It consists of two symmetrical DBNs. The first half of the network is composed of several layers, which performs encoding. The second part of the network performs decoding. Each layer of the autoencoder is an RBM. This is illustrated in the following figure:



The purpose of the encoding sequence is to compress the original input into a smaller vector space. The middle layer of the previous figure is this compressed layer. These intermediate vectors can be thought of as possible features of the dataset. The encoding is also referred to as the pre-training half. It is the output of the intermediate RBM layer and does not perform classification.

The encoder's first layer will use more inputs than used by the dataset. This has the effect of expanding the features of the dataset. A sigmoid-belief unit is a form of non-linear transformation used with each layer. This unit is not able to accurately represent information as real values. However, using more inputs, it is able to do a better job.

The second half of the network performs decoding, effectively reconstructing the input. This is a forward-feed network, using the same weights as the corresponding layers in the encoding half. However, the weights are transposed and are not initialized randomly. The training rate needs to be set lower for the second half.

An autoencoder is useful for data compression and searching. The output of the first half of the model is compressed, thus making it useful for storage and transmission usage. Later, it can be decompressed, as we will demonstrate in `Chapter 10`, *Visual and Audio Analysis*. This is sometimes referred to as semantic hashing.

If a series of inputs, such as images or sounds, have been compressed and stored, then new input can be compressed and matched with the stored values to find the best fit. An autoencoder can also be used for other information retrieval tasks.

Building an autoencoder in DL4J

This example is adapted from `http://deeplearning4j.org/deepautoencoder`. We start with a try-catch block to handle errors that may crop up and with a few variable declarations. This example uses the `Mnist` (`http://yann.lecun.com/exdb/mnist/`) dataset, which is a set of images containing hand-written numbers. Each image consists of `28` by `28` pixels. An iterator is declared to access the data:

```
try {
    final int numberOfRows = 28;
    final int numberOfColumns = 28;
    int seed = 123;
    int numberOfIterations = 1;

    iterator = new MnistDataSetIterator(
        1000, MnistDataFetcher.NUM_EXAMPLES, true);
    ...
} catch (IOException ex) {
    // Handle exceptions
}
```

Configuring the network

The configuration of the network is created using the `NeuralNetConfiguration.Builder()` class. Ten layers are created where the input layer consists of `1000` neurons. This is larger than the 28 by 28 pixel input and is used to compensate for the sigmoid-belief units used in each layer.

Each of the subsequent layers gets smaller until layer four is reached. This layer represents the last step of the encoding process. With layer five, the decoding process starts and the subsequent layers get bigger. The last layer uses `1000` neurons.

Each layer of the model uses an RBM instance except the last layer, which is constructed using the `OutputLayer.Builder` class. The configuration code follows:

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .seed(seed)
        .iterations(numberOfIterations)
        .optimizationAlgo(
           OptimizationAlgorithm.LINE_GRADIENT_DESCENT)
        .list()
        .layer(0, new RBM.Builder()
            .nIn(numberOfRows * numberOfColumns).nOut(1000)
            .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
            .build())
        .layer(1, new RBM.Builder().nIn(1000).nOut(500)
            .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
            .build())
        .layer(2, new RBM.Builder().nIn(500).nOut(250)
            .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
            .build())
        .layer(3, new RBM.Builder().nIn(250).nOut(100)
            .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
            .build())
        .layer(4, new RBM.Builder().nIn(100).nOut(30)
            .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
            .build()) //encoding stops
        .layer(5, new RBM.Builder().nIn(30).nOut(100)
            .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
            .build()) //decoding starts
        .layer(6, new RBM.Builder().nIn(100).nOut(250)
            .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
            .build())
        .layer(7, new RBM.Builder().nIn(250).nOut(500)
            .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
            .build())
        .layer(8, new RBM.Builder().nIn(500).nOut(1000)
            .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
            .build())
        .layer(9, new OutputLayer.Builder(
                LossFunctions.LossFunction.RMSE_XENT).nIn(1000)
                .nOut(numberOfRows * numberOfColumns).build())
        .pretrain(true).backprop(true)
        .build();
```

Building and training the network

The model is then created and initialized, and score iteration listeners are set up:

```
model = new MultiLayerNetwork(conf);
model.init();
model.setListeners(Collections.singletonList(
        (IterationListener) new ScoreIterationListener()));
```

The model is trained using the `fit` method:

```
while (iterator.hasNext()) {
    DataSet dataSet = iterator.next();
    model.fit(new DataSet(dataSet.getFeatureMatrix(),
            dataSet.getFeatureMatrix()));
}
```

Saving and retrieving a network

It is useful to save the model so that it can be used for later analysis. This is accomplished using the `ModelSerializer` class's `writeModel` method. It takes the `model` instance and `modelFile` instance, along with a `boolean` parameter specifying whether the model's updater should be saved. An updater is a learning algorithm used for adjusting certain model parameters:

```
modelFile = new File("savedModel");
ModelSerializer.writeModel(model, modelFile, true);
```

The model can be retrieved using the following code:

```
modelFile = new File("savedModel");
MultiLayerNetwork model =
ModelSerializer.restoreMultiLayerNetwork(modelFile);
```

Specialized autoencoders

There are specialized versions of autoencoders. When an autoencoder uses more hidden layers than inputs, it may learn the identity function, which is a function that always returns the same value used as input to the function. To avoid this problem, an extension to the **autoencoder**, **denoising autoencoder**, is used; it randomly modifies the input introducing noise. The amount of noise introduced varies depending on the input dataset. A **Stacked Denoising Autoencoder** (**SdA**) is a series of denoising autoencoders strung together.

Convolutional networks

CNNs are feed-forward networks modeled after the visual cortex found in animals. The visual cortex is arranged with overlapping neurons, and so in this type of network, the neurons are also arranged in overlapping sections, known as receptive fields. Due to their design model, they function with minimal preprocessing or prior knowledge, and this lack of human intervention makes them especially useful.

This type of network is used frequently in image and video recognition applications. They can be used for classification, clustering, and object recognition. CNNs can also be applied to text analysis by implementing **Optical Character Recognition** (**OCR**). CNNs have been a driving force in the machine learning movement in part due to their wide applicability in practical situations.

We are going to demonstrate a CNN using DL4J. The process will closely mirror the process we used in the *Building an autoencoder in DL4J* section. We will again use the `Mnist` dataset. This dataset contains image data, so it is well-suited to a convolutional network.

Building the model

First, we need to create a new `DataSetIterator` to process the data. The parameters for the `MnistDataSetIterator` constructor are the batch size, `1000` in this case, and the total number of samples to process. We then get our next dataset, shuffle the data to randomize, and split our data to be tested and trained. As we discussed earlier in the chapter, we typically use 65% of the data to train the data and the remaining 35% is used for testing:

```
DataSetIterator iter = new MnistDataSetIterator(1000,
MnistDataFetcher.NUM_EXAMPLES);
DataSet dataset = iter.next();
dataset.shuffle();
SplitTestAndTrain testAndTrain = dataset.splitTestAndTrain(0.65);
DataSet trainingData = testAndTrain.getTrain();
DataSet testData = testAndTrain.getTest();
```

We then normalize both sets of data:

```
DataNormalization normalizer = new NormalizerStandardize();
normalizer.fit(trainingData);
normalizer.transform(trainingData);
normalizer.transform(testData);
```

Next, we can build our network. As shown earlier, we will again use a `MultiLayerConfiguration` instance with a series of `NeuralNetConfiguration.Builder` methods. We will discuss the individual methods after the following code sequence. Notice that the last layer again uses the `softmax` activation function for regression analysis:

```
MultiLayerConfiguration.Builder builder = new
          NeuralNetConfiguration.Builder()
     .seed(123)
     .iterations(1)
     .regularization(true).l2(0.0005)
     .weightInit(WeightInit.XAVIER)
     .optimizationAlgo(OptimizationAlgorithm
          .STOCHASTIC_GRADIENT_DESCENT)
     .updater(Updater.NESTEROVS).momentum(0.9)
     .list()
     .layer(0, new ConvolutionLayer.Builder(5, 5)
          .nIn(6)
          .stride(1, 1)
          .nOut(20)
          .activation("identity")
          .build())
     .layer(1, new SubsamplingLayer.Builder(
               SubsamplingLayer.PoolingType.MAX)
          .kernelSize(2, 2)
          .stride(2, 2)
          .build())
     .layer(2, new ConvolutionLayer.Builder(5, 5)
          .stride(1, 1)
          .nOut(50)
          .activation("identity")
          .build())
     .layer(3, new SubsamplingLayer.Builder(
               SubsamplingLayer.PoolingType.MAX)
          .kernelSize(2, 2)
          .stride(2, 2)
          .build())
     .layer(4, new DenseLayer.Builder().activation("relu")
          .nOut(500).build())
     .layer(5, new OutputLayer.Builder(
               LossFunctions.LossFunction.NEGATIVELOGLIKELIHOOD)
          .nOut(10)
          .activation("softmax")
          .build())
     .backprop(true).pretrain(false);
```

The first layer, layer `0`, which is duplicated next for your convenience, uses the `ConvolutionLayer.Builder` method. The input to a convolution layer is the product of the image height, width, and number of channels. In a standard RGB image, there are three channels. The `nIn` method takes the number of channels. The `nOut` method specifies that `20` outputs are expected:

```
.layer(0, new ConvolutionLayer.Builder(5, 5)
        .nIn(6)
        .stride(1, 1)
        .nOut(20)
        .activation("identity")
        .build())
```

Layers `1` and `3` are both subsampling layers. These layers follow convolution layers and do no real convolution themselves. They return a single value, the maximum value for that input region:

```
.layer(1, new SubsamplingLayer.Builder(
            SubsamplingLayer.PoolingType.MAX)
        .kernelSize(2, 2)
        .stride(2, 2)
        .build())
                        ...
.layer(3, new SubsamplingLayer.Builder(
            SubsamplingLayer.PoolingType.MAX)
        .kernelSize(2, 2)
        .stride(2, 2)
        .build())
```

Layer `2` is also a convolution layer like layer `0`. Notice that we do not specify the number of channels in this layer:

```
.layer(2, new ConvolutionLayer.Builder(5, 5)
        .nOut(50)
        .activation("identity")
        .build())
```

The fourth layer uses the `DenseLayer.Builder` class, as in our earlier example. As mentioned previously, the `DenseLayer` class is a feed-forward and fully connected layer:

```
.layer(4, new DenseLayer.Builder().activation("relu")
        .nOut(500).build())
```

The layer `5` is an `OutputLayer` instance and uses `softmax` automation:

```
.layer(5, new OutputLayer.Builder(
            LossFunctions.LossFunction.NEGATIVELOGLIKELIHOOD)
        .nOut(10)
        .activation("softmax")
        .build())
        .backprop(true).pretrain(false);
```

Finally, we create a new instance of the `ConvolutionalLayerSetup` class. We pass the builder object and the dimensions of our image (28 x 28). We also pass the number of channels, in this case, `1`:

```
new ConvolutionLayerSetup(builder, 28, 28, 1);
```

We can now configure and fit our model. We once again use the `MultiLayerConfiguration` and `MultiLayerNetwork` classes to build our network. We set up listeners and then iterate through our data. For each `DataSet`, we execute the `fit` method:

```
MultiLayerConfiguration conf = builder.build();
MultiLayerNetwork model = new MultiLayerNetwork(conf);
model.init();
model.setListeners(Collections.singletonList((IterationListener)
  new ScoreIterationListener(1/5)));

while (iter.hasNext()) {
    DataSet next = iter.next();
    model.fit(new DataSet(next.getFeatureMatrix(), next.getLabels()));
}
```

We are now ready to evaluate our model.

Evaluating the model

To evaluate our model, we use the `Evaluation` class. We get the output from our model and send it, along with the labels for our dataset, to the `eval` method. We then execute the `stats` method to get the statistical information on our network:

```
Evaluation evaluation = new Evaluation(4);
INDArray output = model.output(testData.getFeatureMatrix());
evaluation.eval(testData.getLabels(), output);
out.println(evaluation.stats());
```

The following is a sample output from the execution of this code, for we are only showing the results of the `stats` method. The first part reports on how examples are classified and the second part displays various statistics:

```
Examples labeled as 0 classified by model as 0: 19 times
Examples labeled as 1 classified by model as 1: 41 times
Examples labeled as 2 classified by model as 1: 4 times
Examples labeled as 2 classified by model as 2: 30 times
Examples labeled as 2 classified by model as 3: 1 times
Examples labeled as 3 classified by model as 2: 1 times
Examples labeled as 3 classified by model as 3: 28 times
==========================Scores========================================Accuracy
: 0.3371
Precision: 0.8481
Recall: 0.8475
F1 Score: 0.8478
========================================================================
```

As in our previous model, the evaluation demonstrates decent accuracy and success with our network.

Recurrent Neural Networks

RNN differ from feed-forward networks in that their input includes the input from the previous iteration or step. They still process the current input but use a feedback loop to take into consideration the inputs to the prior step, also called the recent past, for context. This step effectively gives the network memory. One popular type of recurrent network involves **Long Short-Term Memory** (**LSTM**). This type of memory improves the processing power of the network.

RNNs are designed to process sequential data and are especially useful for analysis and prediction with text data. Given a sequence of words, an RNN can predict the probability of each word being the next in the sequence. This also allows for text generation by the network. RNNs are versatile and also process image data well, especially image labeling applications. The flexibility in design and purpose and ease in training make RNNs popular choices for many data science applications. DL4J also provides support for LSTM networks and other RNNs.

Summary

In this chapter, we examined deep learning techniques for neural networks. All API support in this chapter was provided by Deeplearning4j. We began by demonstrating how to acquire and prepare data for use with deep learning networks. We discussed how to configure and build a model. This was followed by an explanation of how to train and test a model by splitting the dataset into training and testing segments.

Our discussion continued with an examination of deep learning and regression analysis. We showed how to prepare the data and class, build the model, and evaluate the model. We used sample data and displayed output statistics to demonstrate the relative effectiveness of our model.

RBM and DBNs were then examined. DBNs are comprised of RBMs stacked together and are especially useful for classification and clustering applications. Deep autoencoders are also built using RBMs, with two symmetrical DBNs. The autoencoders are especially useful for feature selection and extraction.

Finally, we demonstrated a convolutional network. This network is modeled after the visual cortex and allows the network to use regions of information for classification. As in previous examples, we built, trained, and then evaluated the model for effectiveness. We then concluded the chapter with a brief introduction to recurrent neural networks.

We will expand upon these topics as we move into next chapter and examine text analysis techniques.

9
Text Analysis

Text analysis is a broad topic and is typically referred to as **Natural Language Processing** (**NLP**). It is used for many different tasks, including text searching, language translation, sentiment analysis, speech recognition, and classification, to mention a few. The process of analyzing can be difficult due to the particularities and ambiguity found in natural languages. However, there has been a considerable amount of work in this area and there are several Java APIs supporting this effort.

We will start with an introduction to the basic concepts and tasks used in NLP. These include the following:

- **Tokenization**: The process of splitting text into individual tokens or words.
- **Stop words**: These are words that are common and may not be necessary for processing. They include such words as the, a, and to.
- **Name Entity Recognition** (**NER**): This is the process of identifying elements of text such as people's name, locations, or things.
- **Parts of Speech** (**POS**): This identifies the grammatical parts of a sentence such as noun, verb, adjective, and so on.
- **Relationships**: Here, we are concerned with identifying how parts of text are related to each other, such as the subject and object of a sentence.

The concepts of words, sentences, and paragraphs are well known. However, extracting and analyzing these components is not always that straightforward. The term **corpus** frequently refers to a collection of text.

As with most data science problems, it is important to preprocess text. Frequently, this involves handling such tasks as these:

- Handling Unicode
- Converting text to uppercase or lowercase
- Removing stop words

We examined several techniques for tokenization and removing stop words in `Chapter 3`, *Data Cleaning*. In this chapter, we will focus on POS, NER, extracting relationships from sentence, text classification, and sentiment analysis.

There are several NLP APIs available, including these:

- **OpenNLP** (`https://opennlp.apache.org/`): An open source Apache project
- **Stanford NLP** (`http://nlp.stanford.edu/software/`) : Another open source library
- **UIMA** (`https://uima.apache.org/`): An Apache project supporting pipelines
- **LingPipe** (`http://alias-i.com/lingpipe/`): A library that uses pipelines extensively
- **DL4J** (`http://deeplearning4j.org/`): The Deep Learning for Java library supports various classes for deep learning neural networks including support for NLP

We will use OpenNLP and DL4J to demonstrate text analysis in this chapter. We chose these because they are both well-known and have good published resources for additional support.

We will use the Google **Word2Vec** and **Doc2Vec** neural networks to perform text classification. This includes feature vectors based on other words as well as using labeled information to classify documents. Finally, we will discuss sentiment analysis. This type of analysis seeks to assign meaning to text and also uses the Word2Vec network.

We start our discussion with NER.

Implementing named entity recognition

This is sometimes referred to as finding people and things. Given a text segment, we may want to identify all the names of people present. However, this is not always easy because a name such as Rob may also be used as a verb.

In this section, we will demonstrate how to use OpenNLP's `TokenNameFinderModel` class to find names and locations in text. While there are other entities we may want to find, this example will demonstrate the basics of the technique. We begin with names.

Most names occur within a single line. We do not want to use multiple lines because an entity such as a state might inadvertently be identified incorrectly. Consider the following sentences:

Jim headed north. Dakota headed south.

If we ignored the period, then the state of North Dakota might be identified as a location, when in fact it is not present.

Using OpenNLP to perform NER

We start our example with a try-catch block to handle exceptions. OpenNLP uses models that have been trained on different sets of data. In this example, the `en-token.bin` and `en-ner-person.bin` files contain the models for the tokenization of English text and for English name elements, respectively. These files can be downloaded from `http://opennlp.sourceforge.net/models-1.5/`. However, the IO stream used here is standard Java:

```
try (InputStream tokenStream =
            new FileInputStream(new File("en-token.bin"));
        InputStream personModelStream = new FileInputStream(
            new File("en-ner-person.bin"));) {
    ...
} catch (Exception ex) {
    // Handle exceptions
}
```

An instance of the `TokenizerModel` class is initialized using the token stream. This instance is then used to create the actual `TokenizerME` tokenizer. We will use this instance to tokenize our sentence:

```
TokenizerModel tm = new TokenizerModel(tokenStream);
TokenizerME tokenizer = new TokenizerME(tm);
```

The `TokenNameFinderModel` class is used to hold a model for name entities. It is initialized using the person model stream. An instance of the `NameFinderME` class is created using this model since we are looking for names:

```
TokenNameFinderModel tnfm = new
  TokenNameFinderModel(personModelStream);
NameFinderME nf = new NameFinderME(tnfm);
```

To demonstrate the process, we will use the following sentence. We then convert it to a series of tokens using the tokenizer and `tokenizer` method:

```
String sentence = "Mrs. Wilson went to Mary's house for dinner.";
String[] tokens = tokenizer.tokenize(sentence);
```

The `Span` class holds information regarding the positions of entities. The `find` method will return the position information, as shown here:

```
Span[] spans = nf.find(tokens);
```

This array holds information about person entities found in the sentence. We then display this information as shown here:

```
for (int i = 0; i < spans.length; i++) {
    out.println(spans[i] + " - " + tokens[spans[i].getStart()]);
}
```

The output for this sequence is as follows. Notice that it identifies the last name of Mrs. Wilson but not the "Mrs.":

```
[1..2) person - Wilson
[4..5) person - Mary
```

Once these entities have been extracted, we can use them for specialized analysis.

Identifying location entities

We can also find other types of entities such as dates and locations. In the following example, we find locations in a sentence. It is very similar to the previous person example, except that an `en-ner-location.bin` file is used for the model:

```
try (InputStream tokenStream =
            new FileInputStream("en-token.bin");
        InputStream locationModelStream = new FileInputStream(
            new File("en-ner-location.bin"));) {

    TokenizerModel tm = new TokenizerModel(tokenStream);
```

```
        TokenizerME tokenizer = new TokenizerME(tm);

        TokenNameFinderModel tnfm =
            new TokenNameFinderModel(locationModelStream);
        NameFinderME nf = new NameFinderME(tnfm);

        sentence = "Enid is located north of Oklahoma City.";
        String tokens[] = tokenizer.tokenize(sentence);

        Span spans[] = nf.find(tokens);

        for (int i = 0; i < spans.length; i++) {
            out.println(spans[i] + " - " +
            tokens[spans[i].getStart()]);
        }
    } catch (Exception ex) {
        // Handle exceptions
    }
```

With the sentence defined previously, the model was only able to find the second city, as shown here. This likely due to the confusion that arises with the name `Enid` which is both the name of a city and a person' name:

```
[5..7) location - Oklahoma
```

Suppose we use the following sentence:

```
sentence = "Pond Creek is located north of Oklahoma City.";
```

Then we get this output:

```
[1..2) location - Creek
[6..8) location - Oklahoma
```

Unfortunately, it has missed the town of `Pond Creek`. NER is a useful tool for many applications, but like many techniques, it is not always foolproof. The accuracy of the NER approach presented, and many of the other NLP examples, will vary depending on factors such as the accuracy of the model, the language being used, and the type of entity.

We may also be interested in how text can be classified. We will examine one approach in the next section.

Classifying text

Classifying text is an important part of machine learning and data science. We have to be able to classify text for a variety of applications, including document retrieval and web searches. It is often important to assign specific labels to the data before we can determine its usefulness for a particular application or search result.

In this chapter, we are going to demonstrate a technique involving the use of paragraph vectors and labeled data with DL4J classes. This example allows us to read in documents and, based on the text inside of the document, assign a label (or classification) to the document. We are also going to show an example of classifying text by similarity. This means we will match phrases and words that have similar structure. This example will also use DL4J.

Word2Vec and Doc2Vec

We will be using Word2Vec and Doc2Vec in a few examples in this chapter. Word2Vec is a neural network with two layers used for text processing. Given a body of text, the network will provide feature vectors for the words contained in the text. These vectors are simply mathematical representations of the word features and can be numerically compared to other vectors. This comparison is often referred to as the distance between two words.

Word2Vec operates with the understanding that words can be classified by determining the probability that two words will occur together. Because of this methodology, Word2Vec can be used for more than classification of sentences. Any object or data that can be represented by text labels can be classified with this network.

Doc2Vec is an extension of Word2Vec. Rather than building vectors representing the features of individual words compared to other words, as Word2Vec does, this network compares words to given labels. The vectors are set up to represent the theme or overall meaning of a document. Our next example shows how these feature vectors are then associated with specific documents.

Classifying text by labels

In our first example using Doc2Vec, we will associate our documents with three labels: health, finance, and science. But before we can associate the data with labels, we have to define those labels and train our model to recognize the labels. Each label represents the meaning or classification of a particular piece of text.

In this example we will use sample documents, each pre-labelled with our categories: health, finance, or science. We will use these paragraphs to train our model and then, as in previous examples, use a set of test data to test our model. We will be using the files found at `https://github.com/deeplearning4j/dl4j-examples/tree/master/dl4j-examples/src/main/resources/paravec`. We have based this example upon sample code written for DL4J, which can be found at `https://github.com/deeplearning4j/dl4j-examples/blob/master/dl4j-examples/src/main/java/org/deeplearning4j/examples/nlp/paragraphvectors/ParagraphVectorsClassifierExample.java`.

First we need to set up some instance variables to use later in our code. We will be using a `ParagraphVectors` object to create our vectors, a `LabelAwareIterator` object to iterate through our data, and a `TokenizerFactory` object to tokenize our data:

```
ParagraphVectors pVect;
LabelAwareIterator iter;
TokenizerFactory tFact;
```

Then we will set up our `ClassPathResource`. This specifies the directory within our project that contains the data files to be classified. The first resource contains our labeled data used for training purposes. We then direct our iterator and tokenizer to use the resources specified as the `ClassPathResource`. We also specify that we will use the `CommonPreprocessor` to preprocess our data:

```
ClassPathResource resource = new
         ClassPathResource("paravec/labeled");

iter = new FileLabelAwareIterator.Builder()
        .addSourceFolder(resource.getFile())
        .build();

tFact = new DefaultTokenizerFactory();
tFact.setTokenPreProcessor(new CommonPreprocessor());
```

Next, we build our `ParagraphVectors`. This is where we specify the learning rate, batch size, and number of training epochs. We include our iterator and tokenizer in the setup process as well. Once we've built our `ParagraphVectors`, we call the `fit` method to train our model using the training data in the `paravec/labeled` directory:

```
pVect = new ParagraphVectors.Builder()
        .learningRate(0.025)
        .minLearningRate(0.001)
        .batchSize(1000)
        .epochs(20)
        .iterate(iter)
```

```
            .trainWordVectors(true)
            .tokenizerFactory(tFact)
            .build();

    pVect.fit();
```

Now that we have trained our model, we can use our unlabeled data to test. We create a new `ClassPathResource` for our unlabeled data and create a new `FileLabelAwareIterator` as well:

```
    ClassPathResource unlabeledText =
             new ClassPathResource("paravec/unlabeled");
    FileLabelAwareIterator unlabeledIter =
             new FileLabelAwareIterator.Builder()
                   .addSourceFolder(unlabeledText.getFile())
                   .build();
```

The next step involves iterating through our unlabeled data and identifying the correct label for each document. We can generally expect that each document will fall into multiple labels but have a different weight, or percent match, for each. So, while one article may be mostly classified as a health article, it likely has enough information to be also classified, to a lesser degree, as a science article.

Next, we set up a `MeansBuilder` and `LabelSeeker` object. These classes access tables containing the relationships between words and labels, which we will use in our `ParagraphVectors`. The `InMemoryLookupTable` class provides access to a default table for word lookup:

```
    MeansBuilder mBuilder =
       new MeansBuilder((InMemoryLookupTable<VocabWord>)
          pVect.getLookupTable(),tFact);
    LabelSeeker lSeeker =
        new LabelSeeker(iter.getLabelsSource().getLabels(),
                    (InMemoryLookupTable<VocabWord>)
        pVect.getLookupTable());
```

Finally, we iterate through our unlabeled documents. For each document, we will change the document into a vector and use our `LabelSeeker` to get the scores for each document. We log the scores for each document and print out the score with the appropriate labels:

```
    while (unlabeledIter.hasNextDocument()) {
        LabelledDocument doc = unlabeledIter.nextDocument();
        INDArray docCentroid = mBuilder.documentAsVector(doc);
        List<Pair<String, Double>> scores =
                   lSeeker.getScores(docCentroid);
        out.println("Document '" + doc.getLabel() +
```

```
            "' falls into the following categories: ");
        for (Pair<String, Double> score : scores) {
            out.println ("        " + score.getFirst() + ": " +
                  score.getSecond());
             }

    }
```

The output from our preceding print statements is as follows:

```
Document 'finance' falls into the following categories:
finance: 0.2889593541622162
health: 0.11753179132938385
science: 0.021202782168984413
Document 'health' falls into the following categories:
finance: 0.059537000954151154
health: 0.27373185753822327
science: 0.07699354737997055
```

In each instance, our documents were classified properly, as demonstrated by the higher percentage assigned to the correct label category. This classification can be used in conjunction with other data analysis techniques to draw additional conclusions about the data contained in the files. Often text classification is an initial or early step in a data analysis process as documents are classified into groups for further analysis.

Classifying text by similarity

In this next example, we will match different text samples based on their structure and similarity. We will still be using the `ParagraphVectors` class we used in the previous example. To begin, download the `raw_sentences.txt` file from GitHub (`https://github.com/deeplearning4j/dl4j-examples/tree/master/dl4j-examples/src/main/resources`) and add it to your project. This file contains a list of sentences which we will read in, label, and then compare.

First, we set up our `ClassPathResource` and assign an iterator to handle our file data. We have used a `SentenceIterator` for this example:

```
ClassPathResource srcFile = new
      ClassPathResource("/raw_sentences.txt");
File file = srcFile.getFile();
SentenceIterator iter = new BasicLineIterator(file);
```

Next, we will again use `TokenizerFactory` to tokenize our data. We also want to create a new `LabelsSource` object. This allows us to define the format of our sentence labels. We have chosen to prefix each line with `LINE_`:

```
TokenizerFactory tFact = new DefaultTokenizerFactory();
tFact.setTokenPreProcessor(new CommonPreprocessor());
LabelsSource labelFormat = new LabelsSource("LINE_");
```

Now we are ready to build our `ParagraphVectors`. Our setup process includes these methods: `minWordFrequency`, which specifies the minimum word frequency to use in the training corpus, and `iterations`, which specifies the number of iterations for each mini batch. We also set the number of epochs, the layer size, and the learning rate. Additionally, we include our `LabelsSource`, defined before, and our iterator and tokenizer. The `trainWordVectors` method specifies whether word and document representations should be built together. Finally, `sampling` determines whether subsampling should occur or not. We then call our `build` and `fit` methods:

```
ParagraphVectors vec = new ParagraphVectors.Builder()
        .minWordFrequency(1)
        .iterations(5)
        .epochs(1)
        .layerSize(100)
        .learningRate(0.025)
        .labelsSource(labelFormat)
        .windowSize(5)
        .iterate(iter)
        .trainWordVectors(false)
        .tokenizerFactory(tFact)
        .sampling(0)
        .build();

vec.fit();
```

Next, we will include some statements to evaluate the accuracy of our classifications. It is important to note that while the document itself starts at `1`, the indexing process begins at `0`. So, for example, line `9836` in the document will be associated with the label `LINE_9835`. We will first compare three sentences that should be classified as somewhat similar, and then two examples comparing dissimilar sentences. The `similarity` method takes two labels and returns the relative distance between them in the form of `double`:

```
double similar1 = vec.similarity("LINE_9835", "LINE_12492");
out.println("Comparing lines 9836 & 12493
       ('This is my house .'/'This is my world .')
       Similarity = " + similar1);
```

```
double similar2 = vec.similarity("LINE_3720", "LINE_16392");
out.println("Comparing lines 3721 & 16393
       ('This is my way .'/'This is my work .')
       Similarity = " + similar2);

double similar3 = vec.similarity("LINE_6347", "LINE_3720");
out.println("Comparing lines 6348 & 3721
       ('This is my case .'/'This is my way .')
       Similarity = " + similar3);

double dissimilar1 = vec.similarity("LINE_3720", "LINE_9852");
out.println("Comparing lines 3721 & 9853
       ('This is my way .'/'We now have one .')
       Similarity = " + dissimilar1);

double dissimilar2 = vec.similarity("LINE_3720", "LINE_3719");
out.println("Comparing lines 3721 & 3720
       ('This is my way .'/'At first he says no .')
       Similarity = " + dissimilar2);
```

The output of our print statements is shown as follows. Compare the result of the `similarity` method for the three similar sentences and the two dissimilar sentences. Of particular note, the `similarity` method result for the last example, two very dissimilar sentences, returned a negative number. This implies a more significant disparity:

```
16:56:15.423 [main] INFO o.d.m.s.SequenceVectors - Epoch: [1]; Words
vectorized so far: [3171540]; Lines vectorized so far: [485810];
learningRate: [1.0E-4]
Comparing lines 9836 & 12493 ('This is my house .'/'This is my world .')
Similarity = 0.7641470432281494
Comparing lines 3721 & 16393 ('This is my way .'/'This is my work .')
Similarity = 0.7246013879776001
Comparing lines 6348 & 3721 ('This is my case .'/'This is my way .')
Similarity = 0.8988922834396362
Comparing lines 3721 & 9853 ('This is my way .'/'We now have one .')
Similarity = 0.5840312242507935
Comparing lines 3721 & 3720 ('This is my way .'/'At first he says no .')
Similarity = -0.6491150259971619
```

Although this example uses `ParagraphVectors` like our first classification example, this demonstrates flexibility in our approach. We can use these DL4J libraries to classify data in more than one manner.

Understanding tagging and POS

POS is concerned with identifying the types of components found in a sentence. For example, this sentence has several elements, including the verb "has", several nouns such as "example" and "elements", and adjectives such as "several". Tagging, or more specifically **POS tagging**, is the process of associating element types to words.

POS tagging is useful as it adds more information about the sentence. We can ascertain the relationship between words and often their relative importance. The results of tagging are often used in later processing steps.

This task can be difficult as we are unable to rely upon a simple dictionary of words to determine their type. For example, the word `lead` can be used as both a noun and as a verb. We might use it in either of the following two sentences:

```
He took the lead in the play.
Lead the way!
```

POS tagging will attempt to associate the proper label to each word of a sentence.

Using OpenNLP to identify POS

To illustrate this process, we will be using OpenNLP (`https://opennlp.apache.org/`). This is an open source Apache project which supports many other NLP processing tasks.

We will be using the `POSModel` class, which can be trained to recognize POS elements. In this example, we will use it with a previously trained model based on the **Penn TreeBank tag-set** (`http://www.comp.leeds.ac.uk/ccalas/tagsets/upenn.html`). Various pretrained models are found at `http://opennlp.sourceforge.net/models-1.5/`. We will be using the `en-pos-maxent.bin` model. This has been trained on English text using what is called maximum entropy.

Maximum entropy refers to the amount of uncertainty in the model which it maximizes. For a given problem there is a set of probabilities describing what is known about the data set. These probabilities are used to build a model. For example, we may know that there is a 23 percent chance that one specific event may follow a certain condition. We do not want to make any assumptions about unknown probabilities so we avoid adding unjustified information. A maximum entropy approach attempts to preserve as much uncertainty as possible; hence it attempts to maximize entropy.

We will also use the `POSTaggerME` class, which is a maximum entropy tagger. This is the class that will make tag predictions. With any sentence, there may be more than one way of classifying, or tagging, its components.

We start with code to acquire the previously trained English tagger model and a simple sentence to be tagged:

```
try (InputStream input = new FileInputStream(
        new File("en-pos-maxent.bin"));) {
    String sentence = "Let's parse this sentence.";
    ...
} catch (IOException ex) {
    // Handle exceptions
}
```

The tagger uses an array of strings, where each string is a word. The following sequence takes the previous sentence and creates an array called `words`. The first part uses the `Scanner` class to parse the sentence string. We could have used other code to read the data from a file if needed. After that, the `List` class's `toArray` method is used to create the array of strings:

```
List<String> list = new ArrayList<>();
Scanner scanner = new Scanner(sentence);
while(scanner.hasNext()) {
    list.add(scanner.next());
}
String[] words = new String[1];
words = list.toArray(words);
```

The model is then built using the file containing the model:

```
POSModel posModel = new POSModel(input);
```

The tagger is then created based on the model:

```
POSTaggerME posTagger = new POSTaggerME(posModel);
```

The `tag` method does the actual work. It is passed an array of words and returns an array of tags. The words and tags are then displayed:

```
String[] posTags = posTagger.tag(words);
for(int i=0; i<posTags.length; i++) {
    out.println(words[i] + " - " + posTags[i]);
}
```

The output for this example follows:

```
Let's - NNP
parse - NN
this - DT
sentence. - NN
```

The analysis has determined that the word `let's` is a singular proper noun while the words `parse` and `sentence` are singular nouns. The word `this` is a determiner, that is, it is a word that modifies another and helps identify a phrase as general or specific. A list of tags is provided in the next section.

Understanding POS tags

The POS elements returned abbreviations. A list of **Penn TreeBankPOS** tags can be found at `https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html`. The following is a shortened version of this list:

Tag	Description	Tag	Description
DT	Determiner	RB	Adverb
JJ	Adjective	RBR	Adverb, comparative
JJR	Adjective, comparative	RBS	Adverb, superlative
JJS	Adjective, superlative	RP	Particle
NN	Noun, singular or mass	SYM	Symbol
NNS	Noun, plural	TOP	Top of the parse tree
NNP	Proper noun, singular	VB	Verb, base form
NNPS	Proper noun, plural	VBD	Verb, past tense
POS	Possessive ending	VBG	Verb, gerund or present participle
PRP	Personal pronoun	VBN	Verb, past participle
PRP$	Possessive pronoun	VBP	Verb, non-3rd person singular present
S	Simple declarative clause	VBZ	Verb, 3rd person singular present

As mentioned earlier, there may be more than one possible set of POS assignments for a sentence. The `topKSequences` method, as shown next, will return various assignment possibilities along with a score. The method returns an array of `Sequence` objects whose `toString` method returns the score and POS list:

```
Sequence sequences[] = posTagger.topKSequences(words);
for(Sequence sequence : sequences) {
    out.println(sequence);
}
```

The output for the previous sentence follows, where the last sequence is considered to be the most probable alternative:

```
-2.3264880694837213 [NNP, NN, DT, NN]
-2.6610271245387853 [NNP, VBD, DT, NN]
-2.6630142638557217 [NNP, VB, DT, NN]
```

Each line of output assigns possible tags to each word of the sentence. We can see that only the second word, `parse`, is determined to have other possible tags.

Next, we will demonstrate how to extract relationships from text.

Extracting relationships from sentences

Knowing the relationship between elements of a sentence is important in many analysis tasks. It is useful for assessing the important content of a sentence and providing insight into the meaning of a sentence. This type of analysis has been used for tasks ranging from grammar checking to speech recognition to language translations.

In the previous section, we demonstrated one approach used to extract the parts of speech. Using this technique, we were able to identify the sentence element types present in a sentence. However, the relationships between these elements is missing. We need to parse the sentence to extract these relationships between sentence elements.

Using OpenNLP to extract relationships

There are several techniques and APIs that can be used to extract this type of information. In this section we will use OpenNLP to demonstrate one way of extracting the structure of a sentence. The demonstration is centered around the `ParserTool` class, which uses a previously trained model. The parsing process will return the probabilities that the sentence's elements extracted are correct. As will many NLP tasks, there are often multiple answers possible.

We start with a try-with-resource block to open an input stream for the model. The `en-parser-chunking.bin` file contains a model that uses parses text into its POS. In this case, it is trained for English:

```
try (InputStream modelInputStream = new FileInputStream(
            new File("en-parser-chunking.bin"));) {
    ...
} catch (Exception ex) {
    // Handle exceptions
}
```

Within the try block an instance of the `ParserModel` class is created using the input stream. The actual parser is created next using the `ParserFactory` class's `create` method:

```
ParserModel parserModel = new ParserModel(modelInputStream);
Parser parser = ParserFactory.create(parserModel);
```

We will use the following sentence to test the parser. The `ParserTool` class's `parseLine` method does the actual parsing and returns an array of `Parse` objects. Each of these objects holds one parsing alternative. The last argument of the `parseLine` method specifies how many alternatives to return:

```
String sentence = "Let's parse this sentence.";
Parse[] parseTrees = ParserTool.parseLine(sentence, parser, 3);
```

The next sequence displays each of the possibilities:

```
for(Parse tree : parseTrees) {
    tree.show();
}
```

The output of the show method for this example follows. The tags were previously defined in *Understanding POS tags* section:

```
(TOP (NP (NP (NNP Let's) (NN parse)) (NP (DT this) (NN sentence.))))
(TOP (S (NP (NNP Let's)) (VP (VB parse) (NP (DT this) (NN sentence.)))))
(TOP (S (NP (NNP Let's)) (VP (VBD parse) (NP (DT this) (NN sentence.)))))
```

The following example reformats the last two outputs to better show the relationships. They differ in how they classify the verb parse:

```
(TOP
(S
(NP (NNP Let's))
(VP (VB parse)
(NP (DT this) (NN sentence.))
)
)
)
(TOP
(S
(NP (NNP Let's))
(VP (VBD parse)
(NP (DT this) (NN sentence.))
)
)
)
```

When there are multiple parse alternatives, the `Parse` class's `getProb` returns a probability that reflects the model's confidence in the alternatives. The following sequence demonstrates this method:

```
for(Parse tree : parseTrees) {
    out.println("Probability: " + tree.getProb());
}
```

The output follows:

```
Probability: -3.6810244423259078
Probability: -3.742475884515823
Probability: -4.16148634555491
```

Another interesting NLP task is sentiment analysis, which we will demonstrate next.

Sentiment analysis

Sentiment analysis involves the evaluation and classification of words based on their context, meaning, and emotional implications. Typically, if we were to look up a word in a dictionary we will find a meaning or definition for the word but, taken out of the context of a sentence, we may not be able to ascribe detailed and precise meaning to the word.

For example, the word toast could be defined as simply a slice of heated and browned bread. But in the context of the sentence *He's toast!*, the meaning changes completely. Sentiment analysis seeks to derive meanings of words based on their context and usage.

It is important to note that advanced sentiment analysis will expand beyond simple positive or negative classification and ascribe detailed emotional meaning to words. It is far simpler to classify words as positive or negative but far more useful to classify them as happy, furious, indifferent, or anxious.

This type of analysis falls into the category of effective computing, a type of computing interested in the emotional implications and uses of technological tools. This type of computing is especially significant given the growing amount of emotionally influenced data readily available for analysis on social media sites today.

Being able to determine the emotional content of text enables a more targeted, and appropriate response. For example, being able to judge the emotional response in a chat session between a customer and technical representative can allow the representative to do a better job. This can be especially important when there is a cultural or language gap between them.

This type of analysis can also be applied to visual images. It could be used to gauge someone's response to a new product, such as when conducting a taste test, or to judge how people react to scenes of s movie or commercial.

As part of our example we will be using a bag-of-words model. Bag-of-words models simplify word representation for natural language processing by containing a set, known as the **bag**, of words irrespective of grammar or word order. The words have features used for classification, most importantly the frequency of each word. Because some words such as the, a, or and will naturally have a higher frequency in any text, the words are given a weight as well. Common words with less contextual significance will have a smaller weight and factor less into the text analysis.

Downloading and extracting the Word2Vec model

To demonstrate sentiment analysis, we will use Google's Word2Vec models in conjunction with DL4J to simply classify movie reviews as either positive or negative based upon the words used in the review. This example is adapted from work done by Alex Black (`https://github.com/deeplearning4j/dl4j-examples/blob/master/dl4j-examples/src/main/java/org/deeplearning4j/examples/recurrent/word2vecsentiment/Word2VecSentimentRNN.java`). As discussed previously in this chapter, Word2Vec consists of two-layer neural networks trained to build meaning from the context of words. We will also be using a large set of movie reviews from `http://ai.stanford.edu/~amaas/data/sentiment/`.

Before we begin, you will need to download the Word2Vec data from `https://code.google.com/p/word2vec/`. The basic process includes:

- Downloading and extracting the movie reviews
- Loading the Word2Vec Google News vectors
- Loading each movie review

The words within the reviews are then broken into vectors and used to train the network. We will train the network across five epochs and evaluate the network's performance after each epoch.

To begin, we first declare three final variables. The first is the URL to retrieve the training data, the second is the location to store our extracted data, and the third is the location of the Google News vectors on the local machine. Modify this third variable to reflect the location on your local machine:

```
public static final String TRAINING_DATA_URL =
    "http://ai.stanford.edu/~amaas/" +
    "data/sentiment/aclImdb_v1.tar.gz";
public static final String EXTRACT_DATA_PATH =
    FilenameUtils.concat(System.getProperty(
    "java.io.tmpdir"), "dl4j_w2vSentiment/");
public static final String GNEWS_VECTORS_PATH =
    "C:/YOUR_PATH/GoogleNews-vectors-negative300.bin" +
    "/GoogleNews-vectors-negative300.bin";
```

Next we download and extract our model data. The next two methods are modelled after the code found in the DL4J example. We first create a new method, `getModelData`. The method is shown next in its entirety.

First we create a new `File` using the `EXTRACT_DATA_PATH` we defined previously. If the file does not already exist, we create a new directory. Next, we create two more `File` objects, one for the path to the archived TAR file and one for the path to the extracted data. Before we attempt to extract the data, we check whether these two files exist. If the archive path does not exist, we download the data from the `TRAINING_DATA_URL` and then extract the data. If the extracted file does not exist, we then extract the data:

```
private static void getModelData() throws Exception {
    File modelDir = new File(EXTRACT_DATA_PATH);
    if (!modelDir.exists()) {
        modelDir.mkdir();
    }
    String archivePath = EXTRACT_DATA_PATH + "aclImdb_v1.tar.gz";
    File archiveName = new File(archivePath);
    String extractPath = EXTRACT_DATA_PATH + "aclImdb";
    File extractName = new File(extractPath);
    if (!archiveName.exists()) {
        FileUtils.copyURLToFile(new URL(TRAINING_DATA_URL),
                archiveName);
        extractTar(archivePath, EXTRACT_DATA_PATH);
    } else if (!extractName.exists()) {
        extractTar(archivePath, EXTRACT_DATA_PATH);
    }
}
```

To extract our data, we will create another method called `extractTar`. We will provide two inputs to the method, the `archivePath` and the `EXTRACT_DATA_PATH` defined before. We also need to define our buffer size to use in the extraction process:

```
private static final int BUFFER_SIZE = 4096;
```

We first create a new `TarArchiveInputStream`. We use the `GzipCompressorInputStream` because it provides support for extracting `.gz` files. We also use the `BufferedInputStream` to improve performance in our extraction process. The compressed file is very large and may take some time to download and extract.

Next we create a `TarArchiveEntry` and begin reading in data using the `TarArchiveInputStreamgetNextEntry` method. As we process entry in the compressed file, we first check whether the entry is a directory. If it is, we create a new directory in our extraction location. Finally we create a new `FileOutputStream` and `BufferedOutputStream` and use the `write` method to write our data in the extracted location:

```
private static void extractTar(String dataIn, String dataOut)
    throws IOException {
        try (TarArchiveInputStream inStream =
            new TarArchiveInputStream(
                new GzipCompressorInputStream(
                    new BufferedInputStream(
                        new FileInputStream(dataIn))))) {
            TarArchiveEntry tarFile;
            while ((tarFile = (TarArchiveEntry) inStream.getNextEntry())
                != null) {
                if (tarFile.isDirectory()) {
                    new File(dataOut + tarFile.getName()).mkdirs();
                }else {
                    int count;
                    byte data[] = new byte[BUFFER_SIZE];
                    FileOutputStream fileInStream =
                      new FileOutputStream(dataOut + tarFile.getName());
                    BufferedOutputStream outStream =
                      new BufferedOutputStream(fileInStream,
                        BUFFER_SIZE);
                    while ((count = inStream.read(data, 0, BUFFER_SIZE))
                        != -1) {
                            outStream.write(data, 0, count);
                    }
                }
            }
        }
    }
```

Building our model and classifying text

Now that we have created methods to download and extract our data, we need to declare and initialize variables used to control the execution of our model. Our `batchSize` refers to the amount of words we process in each example, in this case `50`. Our `vectorSize` determines the size of the vectors. The Google News model has word vectors of size `300`. `nEpochs` refers to the number of times we attempt to run through our training data. Finally, `truncateReviewsToLength` specifies whether, for memory utilization purposes, we should truncate the movie reviews if they exceed a specific length. We have chosen to truncate reviews longer than `300` words:

```
int batchSize = 50;
int vectorSize = 300;
int nEpochs = 5;
int truncateReviewsToLength = 300;
```

Now we can set up our neural network. We will use a `MultiLayerConfiguration` network, as discussed in `Chapter 8`, *Deep Learning*. In fact, our example here is very similar to the model built in configuring and building a model, with a few differences. In particular, in this model we will use a faster learning rate and a `GravesLSTM` recurrent network in layer 0. We will have the same number of input neurons as we have words in our vector, in this case, `300`. We also use `gradientNormalization`, a technique used to help our algorithm find the optimal solution. Notice we are using the `softmax` activation function, which was discussed in `Chapter 8`, *Deep Learning*. This function uses regression and is especially suited for classification algorithms:

```
MultiLayerConfiguration sentimentNN =
         new NeuralNetConfiguration.Builder()
        .optimizationAlgo(OptimizationAlgorithm
                 .STOCHASTIC_GRADIENT_DESCENT).iterations(1)
        .updater(Updater.RMSPROP)
        .regularization(true).l2(1e-5)
        .weightInit(WeightInit.XAVIER)
        .gradientNormalization(GradientNormalization
                 .ClipElementWiseAbsoluteValue)
                 .gradientNormalizationThreshold(1.0)
        .learningRate(0.0018)
        .list()
        .layer(0, new GravesLSTM.Builder()
                 .nIn(vectorSize).nOut(200)
                .activation("softsign").build())
        .layer(1, new RnnOutputLayer.Builder()
                .activation("softmax")
                .lossFunction(LossFunctions.LossFunction.MCXENT)
                .nIn(200).nOut(2).build())
```

```
        .pretrain(false).backprop(true).build();
```

We can then create our `MultiLayerNetwork`, initialize the network, and set listeners.

```
MultiLayerNetwork net = new MultiLayerNetwork(sentimentNN);
net.init();
net.setListeners(new ScoreIterationListener(1));
```

Next we create a `WordVectors` object to load our Google data. We use a `DataSetIterator` to test and train our data. The `AsyncDataSetIterator` allows us to load our data in a separate thread, to improve performance. This process requires a large amount of memory and so improvements such as this are essential for optimal performance:

```
WordVectors wordVectors = WordVectorSerializer
DataSetIterator trainData = new AsyncDataSetIterator(
    new SentimentExampleIterator(EXTRACT_DATA_PATH, wordVectors,
        batchSize, truncateReviewsToLength, true), 1);
DataSetIterator testData = new AsyncDataSetIterator(
    new SentimentExampleIterator(EXTRACT_DATA_PATH, wordVectors,
        100, truncateReviewsToLength, false), 1);
```

Finally, we are ready to train and evaluate our data. We run through our data `nEpochs` times; in this case, we have five iterations. Each iteration executes the `fit` method against our training data and then creates a new `Evaluation` object to evaluate our model using `testData`. The evaluation is based on around 25,000 movie reviews and can take a significant amount to time to run. As we evaluate the data, we create `INDArray` to store information, including the feature matrix and labels from our data. This data is used later in the `evalTimeSeries` method for evaluation. Finally, we print out our evaluation statistics:

```
for (int i = 0; i < nEpochs; i++) {
    net.fit(trainData);
    trainData.reset();

    Evaluation evaluation = new Evaluation();
    while (testData.hasNext()) {
        DataSet t = testData.next();
        INDArray dataFeatures = t.getFeatureMatrix();
        INDArray dataLabels = t.getLabels();
        INDArray inMask = t.getFeaturesMaskArray();
        INDArray outMask = t.getLabelsMaskArray();
        INDArray predicted = net.output(dataFeatures, false,
            inMask, outMask);

        evaluation.evalTimeSeries(dataLabels, predicted, outMask);
    }
    testData.reset();
```

```
        out.println(evaluation.stats());
    }
```

The output from the final iteration is shown next. Our examples classified as `0` are considered negative reviews and the ones classified as `1` are considered positive reviews:

```
Epoch 4 complete. Starting evaluation:
Examples labeled as 0 classified by model as 0: 11122 times
Examples labeled as 0 classified by model as 1: 1378 times
Examples labeled as 1 classified by model as 0: 3193 times
Examples labeled as 1 classified by model as 1: 9307 times
==========================Scores========================================Accuracy
: 0.8172
Precision: 0.824
Recall: 0.8172
F1 Score: 0.8206
========================================================================
```

If compared with previous iterations, you should notice the score and accuracy improving with each evaluation. With each iteration, our model improves its accuracy in classifying movie reviews as either negative or positive.

Summary

In this chapter, we introduced a number of NLP tasks and showed how they are supported. In particular, we used OpenNLP and DL4J to illustrate how they are performed. While there are a number of other libraries available, these examples provide a good introduction to the techniques.

We started with an introduction to basic NLP terms and concepts such as named entity recognition, POS, and relationships between elements of a sentence. Named entity recognition is concerned with finding and labeling the parts of a sentence such as people, locations, and things. POS associates labels with elements of a sentence. For example, `NN` refers to a noun and `VB` to a verb.

We then included a discussion of the Word2Vec and Doc2Vec neural networks. These were used to classify text, both with labels and by similarity with other words. We demonstrated the use of DL4J resources to create feature vectors for document association with labels.

While the identification of these associations is interesting, a more useful analysis is performed when relationships are extracted from a sentence. We demonstrated how relationships are found using OpenNLP. The POS are associated with each word and the relationships between the words are shown using a set of tags and parentheses. This type of analysis can be used for more sophisticated analyses such as language translation and grammar checking.

Finally, we discussed and showed examples of sentiment analysis. This process involves classifying text based on its tone or contextual meaning. We examined a process for classifying movie reviews as positive or negative.

In this chapter, we demonstrated various techniques for text analysis and classification. In our next chapter, we will examine techniques designed for video and audio analysis.

10
Visual and Audio Analysis

The use of sound, images, and videos is becoming a more important aspect of our day-to-day lives. Phone conversations and devices reliant on voice commands are increasingly common. People regularly conduct video chats with other people around the world. There has been a rapid proliferation of photo and video sharing sites. Applications that utilize images, video, and sound from a variety of sources are becoming more common.

In this chapter, we will demonstrate several techniques available to Java to process sounds and images. The first part of the chapter addresses sound processing. Both speech recognition and **Text-To-Speech** (**TTS**) APIs will be demonstrated. Specifically, we will use the FreeTTS (`http://freetts.sourceforge.net/docs/index.php`) API to convert text to speech, followed with a demonstration of the CMU Sphinx toolkit for speech recognition.

The **Java Speech API** (**JSAPI**) (`http://www.oracle.com/technetwork/java/index-140170.html`) provides access to speech technology. It is not part of the standard JDK but is supported by third-party vendors. Its intent is to support speech recognition and speech synthesizers. There are several vendors that support JSAPI, including FreeTTS and Festival (`http://www.cstr.ed.ac.uk/projects/festival/`).

In addition, there are several cloud-based speech APIs, including IBM's support through **Watson Cloud** speech-to-text capabilities.

Next, we will examine image processing techniques, including facial recognition. This involves identifying faces within an image. This technique is easy to accomplish using OpenCV (`http://opencv.org/`) which we will demonstrate in the Identifying faces section.

We will end the chapter with a discussion of Neuroph Studio, a neural network Java-based editor, to classify images and perform image recognition. We will continue to use faces here and attempt to train a network to recognize images of human faces.

Text-to-speech

Speech synthesis generates human speech. TTS converts text to speech and is useful for a number of different applications. It is used in many places, including phone help desk systems and ordering systems. The TTS process typically consists of two parts. The first part tokenizes and otherwise processes the text into speech units. The second part converts these units into speech.

The two primary approaches for TTS uses **concatenation synthesis** and **formant synthesis**. Concatenation synthesis frequently combines prerecorded human speech to create the desired output. Formant synthesis does not use human speech but generates speech by creating electronic waveforms.

We will be using FreeTTS (`http://freetts.sourceforge.net/docs/index.php`) to demonstrate TTS. The latest version can be downloaded from `https://sourceforge.net/projects/freetts/files/`. This approach uses concatenation to generate speech.

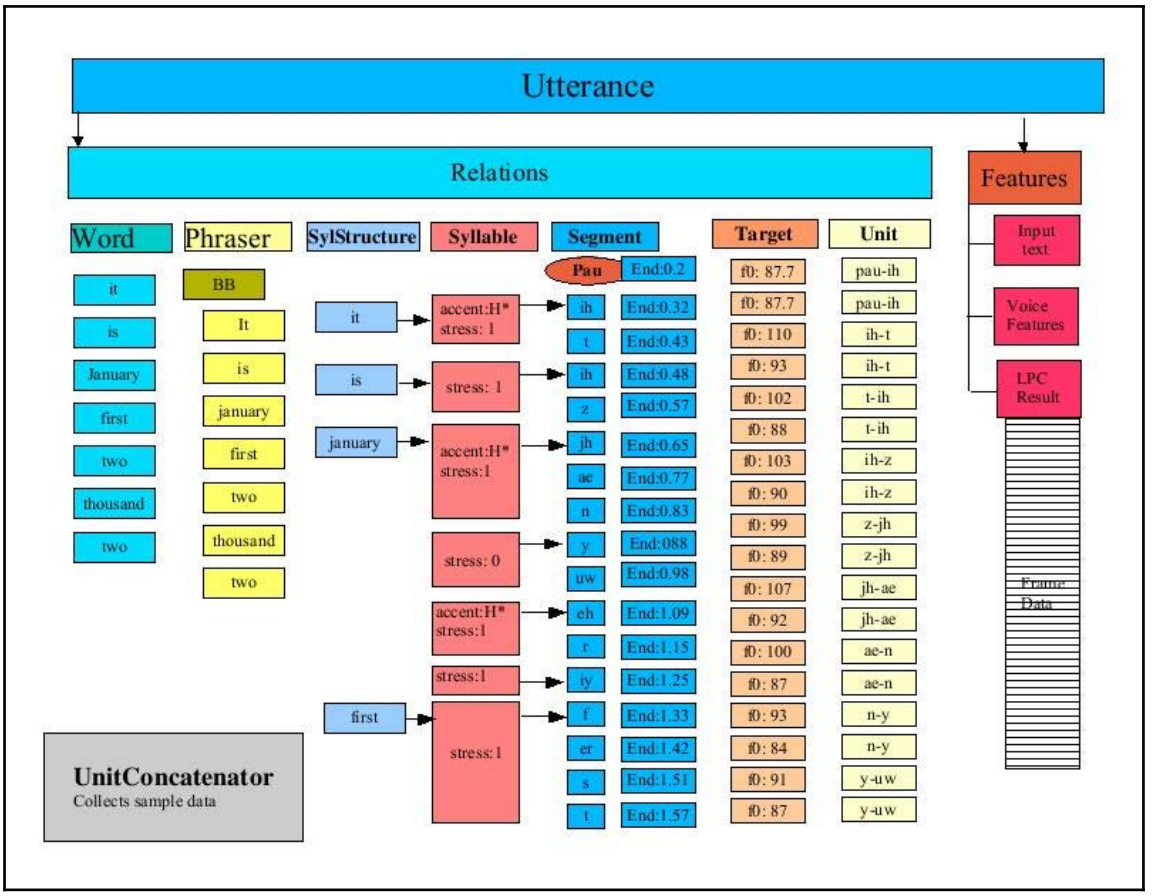
There are several important terms used in TTS/FreeTTS:

- **Utterance** - This concept corresponds roughly to the vocal sounds that make up a word or phrase
- **Items** - Sets of features (name/value pairs) that represent parts of an utterance
- **Relationship** - A list of items, used by FreeTTS to iterate back and forward through an utterance
- **Phone** - A distinct sound
- **Diphone** - A pair of adjacent phones

The FreeTTS Programmer's Guide (`http://freetts.sourceforge.net/docs/ProgrammerGuide.html`) details the process of converting text to speech. This is a multi-step process whose major steps include the following:

- **Tokenization** - Extracting the tokens from the text
- **TokenToWords** - Converting certain words, such as 1910 to nineteen ten
- **PartOfSpeechTagger** - This step currently does nothing, but is intended to identify the parts of speech
- **Phraser** - Creates a phrase relationship for the utterance
- **Segmenter** - Determines where syllable breaks occur
- **PauseGenerator** - This step inserts pauses within speech, such as before utterances
- **Intonator** - Determines the accents and tones
- **PostLexicalAnalyzer** - This step fixes problems such as a mismatch between the available diphones and the one that needs to be spoken
- **Durator** - Determines the duration of the syllables
- **ContourGenerator** - Calculates the fundamental frequency curve for an utterance, which maps the frequency against time and contributes to the generation of tones
- **UnitSelector** - Groups related diphones into a unit
- **PitchMarkGenerator** - Determines the pitches for an utterance
- **UnitConcatenator** - Concatenates the diphone data together

The following figure is from the *FreeTTS Programmer's Guide, Figure 11:* The **Utterance** after **UnitConcatenator** processing, and depicts the process. This high-level overview of the TTS process provides a hint at the complexity of the process:



Using FreeTTS

TTS system facilitates the use of different voices. For example, these differences may be in the language, the sex of the speaker, or the age of the speaker.

The **MBROLA Project's** (`http://tcts.fpms.ac.be/synthesis/mbrola.html`) objective is to support voice synthesizers for as many languages as possible. MBROLA is a speech synthesizer that can be used with a TTS system such as FreeTTS to support TTS synthesis.

Download MBROLA for the appropriate platform binary from `http://tcts.fpms.ac.be/synthesis/mbrola.html`. From the same page, download any desired MBROLA voices found at the bottom of the page. For our examples we will use `usa1`, `usa2`, and `usa3`. Further details about the setup are found at `http://freetts.sourceforge.net/mbrola/README.html`.

The following statement illustrates the code needed to access the MBROLA voices. The `setProperty` method assigns the path where the MBROLA resources are found:

```
System.setProperty("mbrola.base", "path-to-mbrola-directory");
```

To demonstrate how to use TTS, we use the following statement. We obtain an instance of the `VoiceManager` class, which will provide access to various voices:

```
VoiceManager voiceManager = VoiceManager.getInstance();
```

To use a specific voice the `getVoice` method is passed the name of the voice and returns an instance of the `Voice` class. In this example, we used `mbrola_us1`, which is a US English, young, female voice:

```
Voice voice = voiceManager.getVoice("mbrola_us1");
```

Once we have obtained the `Voice` instance, use the `allocate` method to load the voice. The `speak` method is then used to synthesize the words passed to the method as a string, as illustrated here:

```
voice.allocate();
voice.speak("Hello World");
```

When executed, the words `"Hello World"` should be heard. Try this with other voices, as described in the next section, and text to see which combination is best suited for an application.

Getting information about voices

The `VoiceManager` class' `getVoices` method is used to obtain an array of the voices currently available. This can be useful to provide users with a list of voices to choose from. We will use the method here to illustrate some of the voices available. In the next code sequence, the method returns the array, whose elements are then displayed:

```
Voice[] voices = voiceManager.getVoices();
for (Voice v : voices) {
    out.println(v);
}
```

The output will be similar to the following:

```
CMUClusterUnitVoice
CMUDiphoneVoice
CMUDiphoneVoice
MbrolaVoice
MbrolaVoice
MbrolaVoice
```

The `getVoiceInfo` method provides potentially more useful information, though it is somewhat verbose:

```
out.println(voiceManager.getVoiceInfo());
```

The first part of the output follows; the `VoiceDirectory` directory is displayed followed by the details of the voice. Notice that the directory name contains the name of the voice. The `KevinVoiceDirectory` contains two voices: `kevin` and `kevin16`:

```
VoiceDirectory
'com.sun.speech.freetts.en.us.cmu_time_awb.AlanVoiceDirectory'
Name: alan
Description: default time-domain cluster unit voice
Organization: cmu
Domain: time
Locale: en_US
Style: standard
Gender: MALE
Age: YOUNGER_ADULT
Pitch: 100.0
Pitch Range: 12.0
Pitch Shift: 1.0
Rate: 150.0
Volume: 1.0
VoiceDirectory
'com.sun.speech.freetts.en.us.cmu_us_kal.KevinVoiceDirectory'
```

```
Name: kevin
Description: default 8-bit diphone voice
Organization: cmu
Domain: general
Locale: en_US
Style: standard
Gender: MALE
Age: YOUNGER_ADULT
Pitch: 100.0
Pitch Range: 11.0
Pitch Shift: 1.0
Rate: 150.0
Volume: 1.0
Name: kevin16
Description: default 16-bit diphone voice
Organization: cmu
Domain: general
Locale: en_US
Style: standard
Gender: MALE
Age: YOUNGER_ADULT
Pitch: 100.0
Pitch Range: 11.0
Pitch Shift: 1.0
Rate: 150.0
Volume: 1.0
...
Using voices from a JAR file
```

Voices can be stored in JAR files. The `VoiceDirectory` class provides access to voices stored in this manner. The voice directories available to FreeTTs are found in the **lib** directory and include the following:

- `cmu_time_awb.jar`
- `cmu_us_kal.jar`

The name of a voice directory can be obtained from the command prompt:

```
java –jar fileName.jar
```

For example, execute the following command:

```
java –jar cmu_time_awb.jar
```

It generates the following output:

```
VoiceDirectory
'com.sun.speech.freetts.en.us.cmu_time_awb.AlanVoiceDirectory'
Name: alan
Description: default time-domain cluster unit voice
Organization: cmu
Domain: time
Locale: en_US
Style: standard
Gender: MALE
Age: YOUNGER_ADULT
Pitch: 100.0
Pitch Range: 12.0
Pitch Shift: 1.0
Rate: 150.0
Volume: 1.0
```

Gathering voice information

The `Voice` class provides a number of methods that permit the extraction or setting of speech characteristics. As we demonstrated earlier, the `VoiceManager` class' `getVoiceInfo` method provided information about the voices currently available. However, we can use the `Voice` class to get information about a specific voice.

In the following example, we will display information about the voice `kevin16`. We start by getting an instance of this `voice` using the `getVoice` method:

```
VoiceManager vm = VoiceManager.getInstance();
Voice voice = vm.getVoice("kevin16");
voice.allocate();
```

Next, we call a number of the `Voice` class' `get` method to obtain specific information about the voice. This includes previous information provided by the `getVoiceInfo` method and other information that is not otherwise available:

```
out.println("Name: " + voice.getName());
out.println("Description: " + voice.getDescription());
out.println("Organization: " + voice.getOrganization());
out.println("Age: " + voice.getAge());
out.println("Gender: " + voice.getGender());
out.println("Rate: " + voice.getRate());
out.println("Pitch: " + voice.getPitch());
out.println("Style: " + voice.getStyle());
```

The output of this example follows:

```
Name: kevin16
Description: default 16-bit diphone voice
Organization: cmu
Age: YOUNGER_ADULT
Gender: MALE
Rate: 150.0
Pitch: 100.0
Style: standard
```

These results are self-explanatory and give you an idea of the type of information available. There are additional methods that give you access to details regarding the TTS process that are not normally of interest. This includes information such as the audio player being used, utterance-specific data, and features of a specific phone.

Having demonstrated how text can be converted to speech, we will now examine how we can convert speech to text.

Understanding speech recognition

Converting speech to text is an important application feature. This ability is increasingly being used in a wide variety of contexts. Voice input is used to control smart phones, automatically handle input as part of help desk applications, and to assist people with disabilities, to mention a few examples.

Speech consists of an audio stream that is complex. Sounds can be split into **phones**, which are sound sequences that are similar. Pairs of these phones are called **diphones**. **Utterances** consist of words and various types of pauses between them.

The essence of the conversion process involves splitting sounds by silences between utterances. These utterances are then matched to the words that most closely sound like the utterance. However, this can be difficult due to many factors. For example, these differences may be in the form of variances in how words are pronounced due to the context of the word, regional dialects, the quality of the sound, and other factors.

The matching process is quite involved and often uses multiple models. A model may be used to match acoustic features with a sound. A phonetic model can be used to match phones to words. Another model is used to restrict word searches to a given language. These models are never entirely accurate and contribute to inaccuracies found in the recognition process.

We will be using CMUSphinx 4 to illustrate this process.

Using CMUPhinx to convert speech to text

Audio processed by CMUSphinx must be in **Pulse Code Modulation** (**PCM**) format. PCM is a technique that samples analog data, such as an analog wave representing speech, and produces a digital version of the signal. FFmpeg (`https://ffmpeg.org/`) is a free tool that can convert between audio formats if needed.

You will need to create sample audio files using the PCM format. These files should be fairly short and can contain numbers or words. It is recommended that you run the examples with different files to see how well the speech recognition works.

First, we set up the basic framework for the conversion by creating a try-catch block to handle exceptions. First, create an instance of the `Configuration` class. It is used to configure the recognizer to recognize standard English. The configuration models and dictionary need to be changed to handle other languages:

```
try {
    Configuration configuration = new Configuration();
    String prefix = "resource:/edu/cmu/sphinx/models/en-us/";
    configuration
            .setAcousticModelPath(prefix + "en-us");
    configuration
            .setDictionaryPath(prefix + "cmudict-en-us.dict");
    configuration
            .setLanguageModelPath(prefix + "en-us.lm.bin");
    ...
} catch (IOException ex) {
    // Handle exceptions
}
```

The `StreamSpeechRecognizer` class is then created using `configuration`. This class processes the speech based on an input stream. In the following code, we create an instance of the `StreamSpeechRecognizer` class and an `InputStream` from the speech file:

```
StreamSpeechRecognizer recognizer = new StreamSpeechRecognizer(
        configuration);
InputStream stream = new FileInputStream(new File("filename"));
```

To start speech processing, the `startRecognition` method is invoked. The `getResult` method returns a `SpeechResult` instance that holds the result of the processing. We then use the `SpeechResult` method to get the best results. We stop the processing using the `stopRecognition` method:

```
recognizer.startRecognition(stream);
SpeechResult result;
```

```
while ((result = recognizer.getResult()) != null) {
    out.println("Hypothesis: " + result.getHypothesis());
}
recognizer.stopRecognition();
```

When this is executed, we get the following, assuming the speech file contained this sentence:

```
Hypothesis: mary had a little lamb
```

When speech is interpreted there may be more than one possible word sequence. We can obtain the best ones using the `getNbest` method, whose argument specifies how many possibilities should be returned. The following demonstrates this method:

```
Collection<String> results = result.getNbest(3);
for (String sentence : results) {
    out.println(sentence);
}
```

One possible output follows:

```
<s> mary had a little lamb </s>
<s> marry had a little lamb </s>
<s> mary had a a little lamb </s>
```

This gives us the basic results. However, we will probably want to do something with the actual words. The technique for getting the words is explained next.

Obtaining more detail about the words

The individual words of the results can be extracted using the `getWords` method, as shown next. The method returns a list of `WordResult` instance, each of which represents one word:

```
List<WordResult> words = result.getWords();
for (WordResult wordResult : words) {
    out.print(wordResult.getWord() + " ");
}
```

The output for this code sequence follows `<sil>` reflects a silence found at the beginning of the speech:

```
<sil> mary had a little lamb
```

We can extract more information about the words using various methods of the `WordResult` class. In this sequence that follows, we will return the confidence and time frame associated with each word.

The `getConfidence` method returns the confidence expressed as a log. We use the `SpeechResult` class' `getResult` method to get an instance of the `Result` class. Its `getLogMath` method is then used to get a `LogMath` instance. The `logToLinear` method is passed the confidence value and the value returned is a real number between 0 and 1.0 inclusive. More confidence is reflected by a larger value.

The `getTimeFrame` method returns a `TimeFrame` instance. Its `toString` method returns two integer values, separated by a colon, reflecting the beginning and end times of the word:

```
for (WordResult wordResult : words) {
    out.printf("%s\n\tConfidence: %.3f\n\tTime Frame: %s\n",
            wordResult.getWord(), result
                    .getResult()
                    .getLogMath()
                    .logToLinear((float)wordResult
                            .getConfidence()),
            wordResult.getTimeFrame());
}
```

One possible output follows:

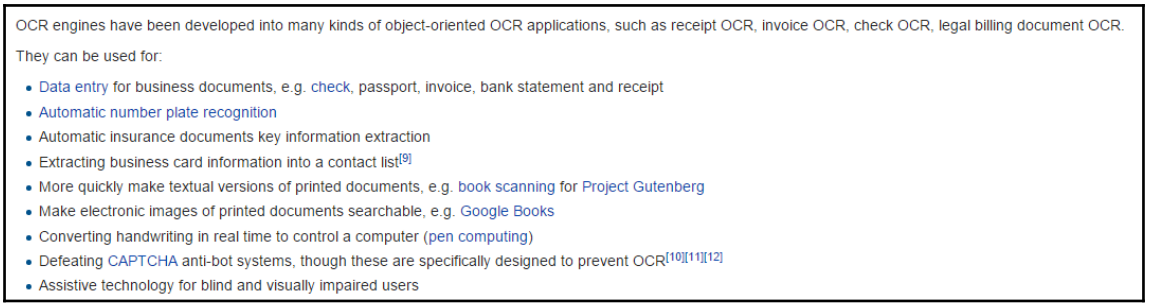
```
<sil>
Confidence: 0.998
Time Frame: 0:430
mary
Confidence: 0.998
Time Frame: 440:900
had
Confidence: 0.998
Time Frame: 910:1200
a
Confidence: 0.998
Time Frame: 1210:1340
little
Confidence: 0.998
Time Frame: 1350:1680
lamb
Confidence: 0.997
Time Frame: 1690:2170
```

Now that we have examined how sound can be processed, we will turn our attention to image processing.

Extracting text from an image

The process of extracting text from an image is called **Optical Character Recognition** (**OCR**). This can be very useful when the text data that needs to be processed is embedded in an image. For example, the information contained in license plates, road signs, and directions can be very useful at times.

We can perform OCR using Tess4j (`http://tess4j.sourceforge.net/`), a Java JNA wrapper for Tesseract OCR API. We will demonstrate how to use the API using an image captured from the Wikipedia article on OCR (`https://en.wikipedia.org/wiki/Optical_character_recognition#Applications`). The Javadoc for the API is found at `http://tess4j.sourceforge.net/docs/docs-3.0/`. The image we use is shown here:



OCR engines have been developed into many kinds of object-oriented OCR applications, such as receipt OCR, invoice OCR, check OCR, legal billing document OCR.

They can be used for:

- Data entry for business documents, e.g. check, passport, invoice, bank statement and receipt
- Automatic number plate recognition
- Automatic insurance documents key information extraction
- Extracting business card information into a contact list[9]
- More quickly make textual versions of printed documents, e.g. book scanning for Project Gutenberg
- Make electronic images of printed documents searchable, e.g. Google Books
- Converting handwriting in real time to control a computer (pen computing)
- Defeating CAPTCHA anti-bot systems, though these are specifically designed to prevent OCR[10][11][12]
- Assistive technology for blind and visually impaired users

Using Tess4j to extract text

The `ITesseract` interface contains numerous OCR methods. The `doOCR` method takes a file and returns a string containing the words found in the file, as shown here:

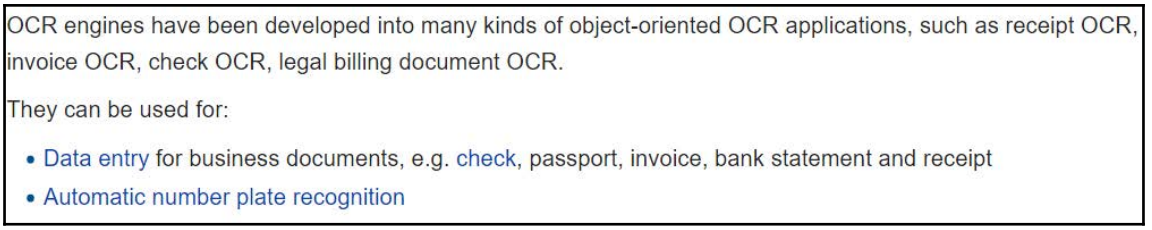
```
ITesseract instance = new Tesseract();
try {
    String result = instance.doOCR(new File("OCRExample.png"));
    out.println(result);
} catch (TesseractException e) {
    // Handle exceptions
}
```

Part of the output is shown next:

```
OCR engines nave been developed into many lunds oiobiectorlented OCR
applicatlons, sucn as reoeipt OCR, involoe OCR, check OCR, legal billing
document OCR
They can be used ior
- Data entry ior business documents, e g check, passport, involoe, bank
statement and receipt
- Automatic number plate recognnlon
```

As you can see, there are numerous errors in this example. Often the quality of an image needs to be improved before it can be processed correctly. Techniques for improving the quality of the output can be found at `https://github.com/tesseract-ocr/tesseract/wiki/ImproveQuality`. For example, we can use the `setLanguage` method to specify the language processed. Also, the method often works better on TIFF images.

In the next example, we used an enlarged portion of the previous image, as shown here:



The output is much better, as shown here:

```
OCR engines have been developed into many kinds of object-oriented OCR
applications, such as receipt OCR,
invoice OCR, check OCR, legal billing document OCR.
They can be used for:
. Data entry for business documents, e.g. check, passport, invoice, bank
statement and receipt
. Automatic number plate recognition
```

These examples highlight the need for the careful cleaning of data.

Identifying faces

Identifying faces within an image is useful in many situations. It can potentially classify an image as one containing people or find people in an image for further processing. We will use OpenCV 3.1 (`http://opencv.org/opencv-3-1.html`) for our examples.

OpenCV (`http://opencv.org/`) is an open source computer vision library that supports several programming languages, including Java. It supports a number of techniques, including machine learning algorithms, to perform computer vision tasks. The library supports such operations as face detection, tracking camera movements, extracting 3D models, and removing red eye from images. In this section, we will demonstrate face detection.

Using OpenCV to detect faces

The example that follows was adapted from `http://docs.opencv.org/trunk/d9/d52/tutorial_java_dev_intro.html`. Start by loading the native libraries added to your system when OpenCV was installed. On Windows, this requires that appropriate DLL files are available:

```
System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
```

We used a base string to specify the location of needed OpenCV files. Using an absolute path works better with many methods:

```
String base = "PathToResources";
```

The `CascadeClassifier` class is used for object classification. In this case, we will use it for face detection. An XML file is used to initialize the class. In the following code, we use the `lbpcascade_frontalface.xml` file, which provides information to assist in the identification of objects. In the OpenCV download are several files, as listed here, that can be used for specific face recognition scenarios:

- `lbpcascade_frontalcatface.xml`
- `lbpcascade_frontalface.xml`
- `lbpcascade_frontalprofileface.xml`
- `lbpcascade_silverware.xml`

The following statement initializes the class to detect faces:

```
CascadeClassifier faceDetector =
        new CascadeClassifier(base +
            "/lbpcascade_frontalface.xml");
```

The image to be processed is loaded, as shown here:

```
Mat image = Imgcodecs.imread(base + "/images.jpg");
```

For this example, we used the following image:



To find this image, perform a Google search using the term **people**. Select the **Images** category and then filter for **Labeled for reuse**. The image has the label: **Closeup portrait of a group of business people laughing by LyndaSanchez**.

When faces are detected, the location within the image is stored in a `MatOfRect` instance. This class is intended to hold vectors and matrixes for any faces found:

```
MatOfRect faceVectors = new MatOfRect();
```

At this point, we are ready to detect faces. The `detectMultiScale` method performs this task. The image and the `MatOfRect` instance to hold the locations of any images are passed to the method:

```
faceDetector.detectMultiScale(image, faceVectors);
```

The next statement shows how many faces were detected:

```
out.println(faceVectors.toArray().length + " faces found");
```

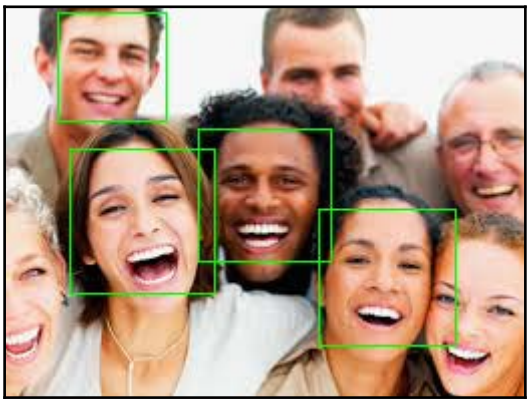
We need to use this information to augment the image. This process will draw boxes around each face found, as shown next. To do this, the `Imgproc` class' `rectangle` method is used. The method is called once for each face detected. It is passed the image to be modified and the points represented the boundaries of the face:

```
for (Rect rect : faceVectors.toArray()) {
    Imgproc.rectangle(image, new Point(rect.x, rect.y),
            new Point(rect.x + rect.width, rect.y + rect.height),
            new Scalar(0, 255, 0));
}
```

The last step writes this image to a file using the `Imgcodecs` class' `imwrite` method:

```
Imgcodecs.imwrite("faceDetection.png", image);
```

As shown in the following image, it was able to identify four images:



Using different configuration files will work better for other facial profiles.

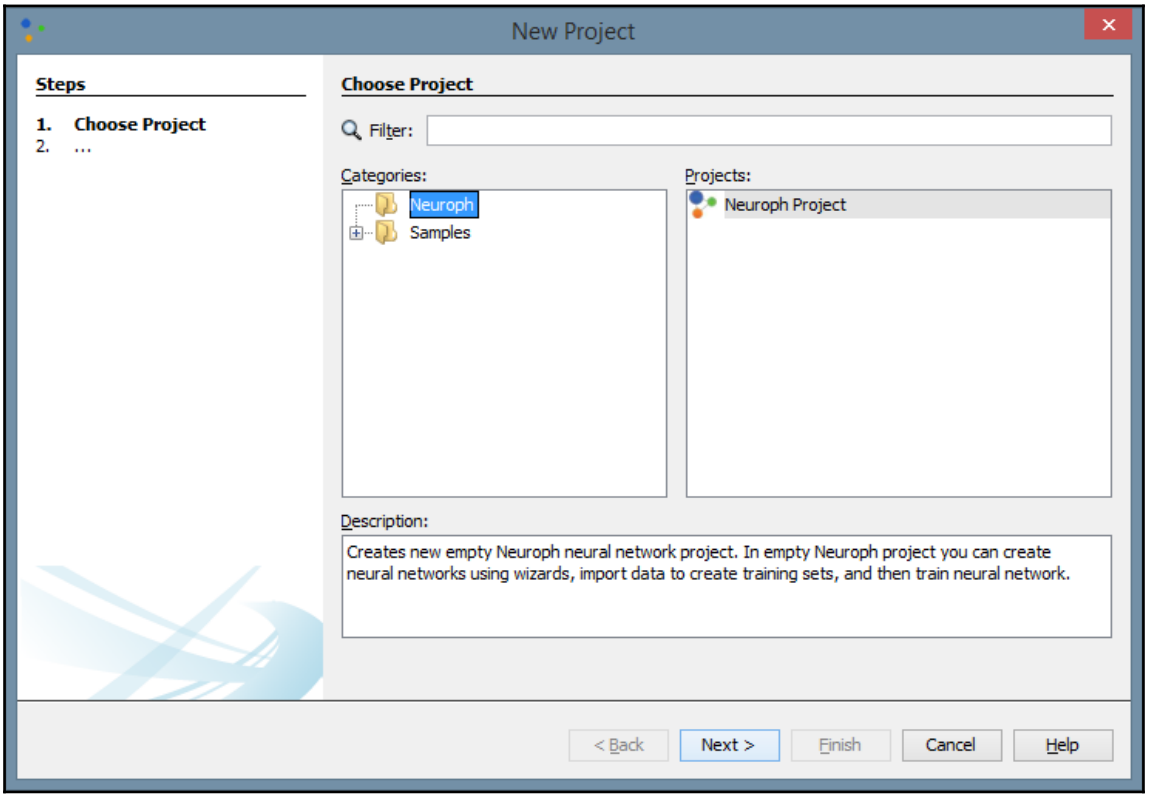
Classifying visual data

In this section, we will demonstrate one technique for classifying visual data. We will use Neuroph to accomplish this. Neuroph is a Java-based neural network framework that supports a variety of neural network architectures. Its open source library provides support and plugins for other applications. In this example, we will use its neural network editor, Neuroph Studio, to create a network. This network can be saved and used in other applications. Neuroph Studio is available for download here: `http://neuroph.sourceforge.net/download.html`. We are building upon the process shown here: `http://neuroph.sourceforge.net/image_recognition.htm`.

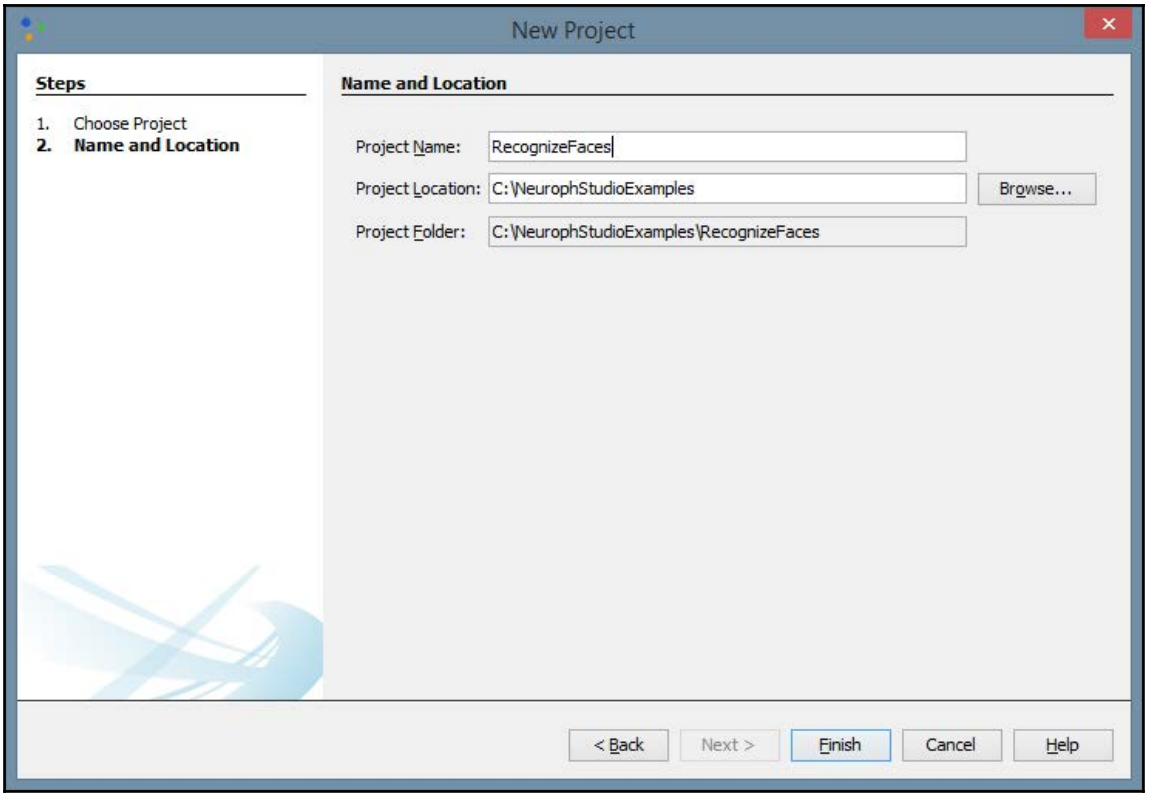
For our example, we will create a **Multi Layer Perceptron** (**MLP**) network. We will then train our network to recognize images. We can both train and test our network using Neuroph Studio. It is important to understand how MLP networks recognize and interpret image data. Every image is basically represented by three two-dimensional arrays. Each array contains information about the color components: one array contains information about the color red, one about the color green, and one about the color blue. Every element of the array holds information about one specific pixel in the image. These arrays are then flattened into a one-dimensional array to be used as an input by the neural network.

Creating a Neuroph Studio project for classifying visual images

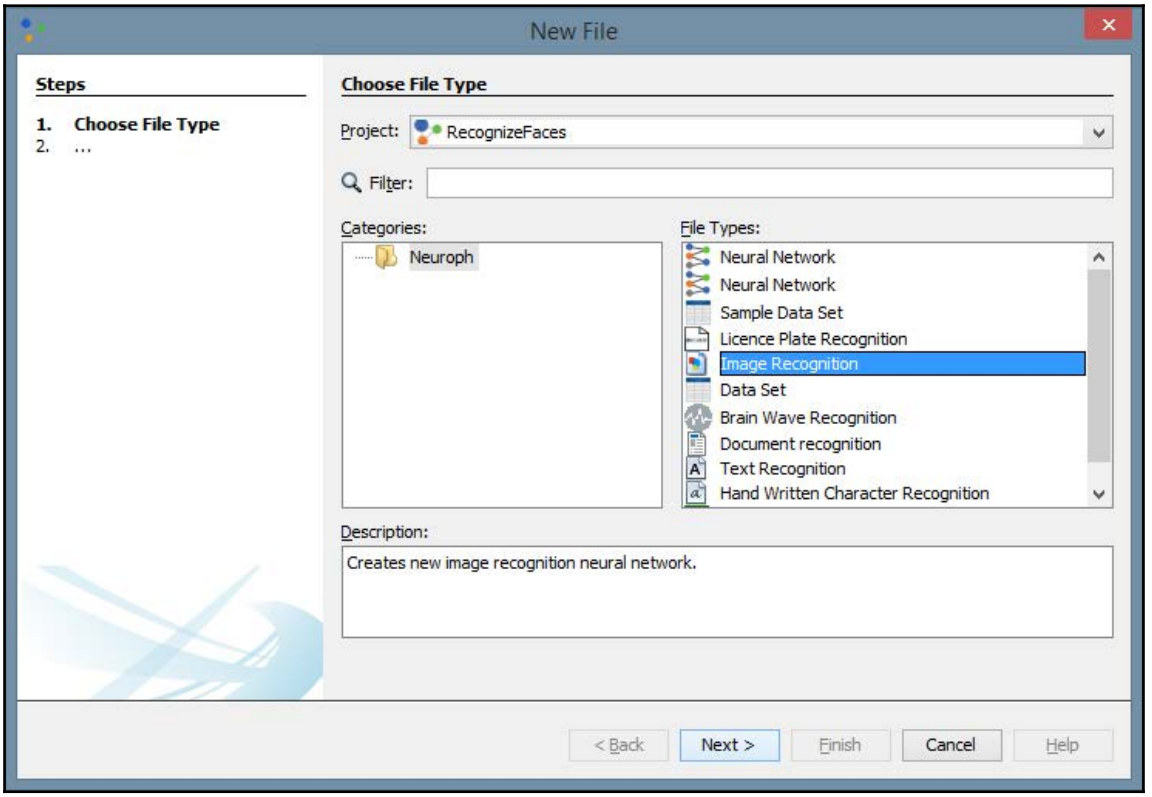
To begin, first create a new Neuroph Studio project:



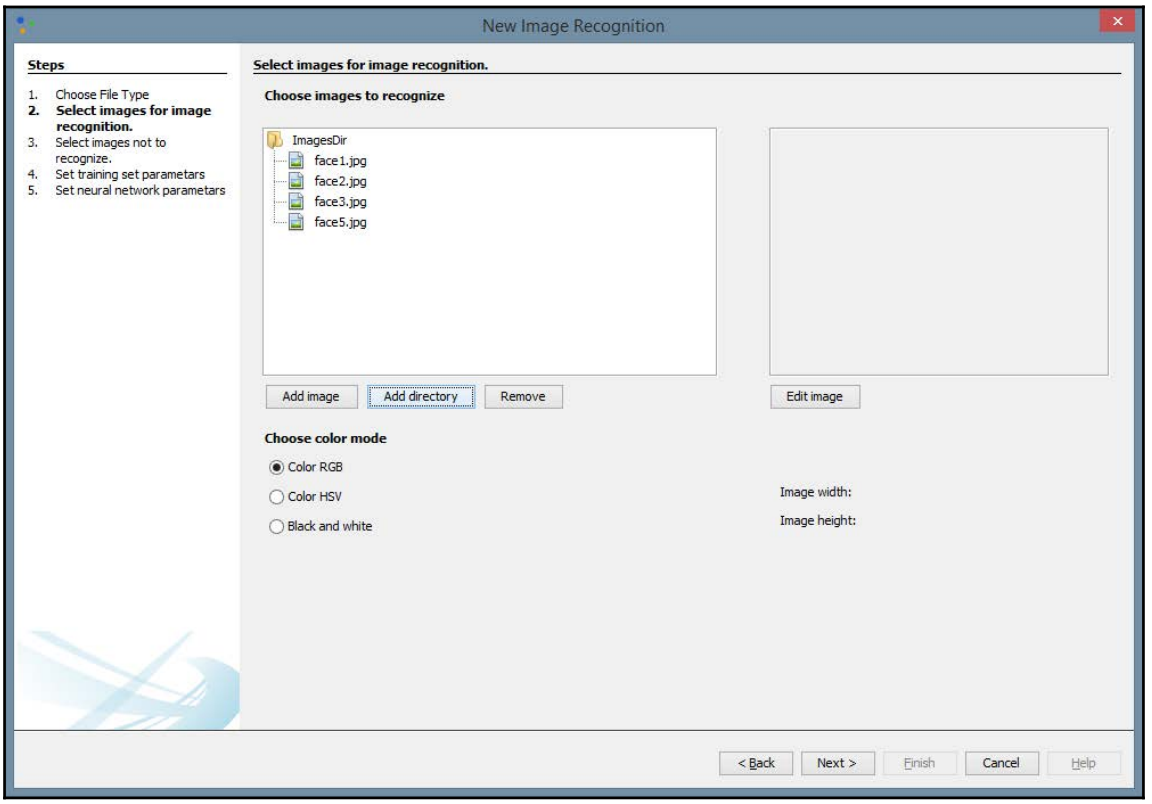
We will name our project `RecognizeFaces` because we are going train the neural network to recognize images of human faces:



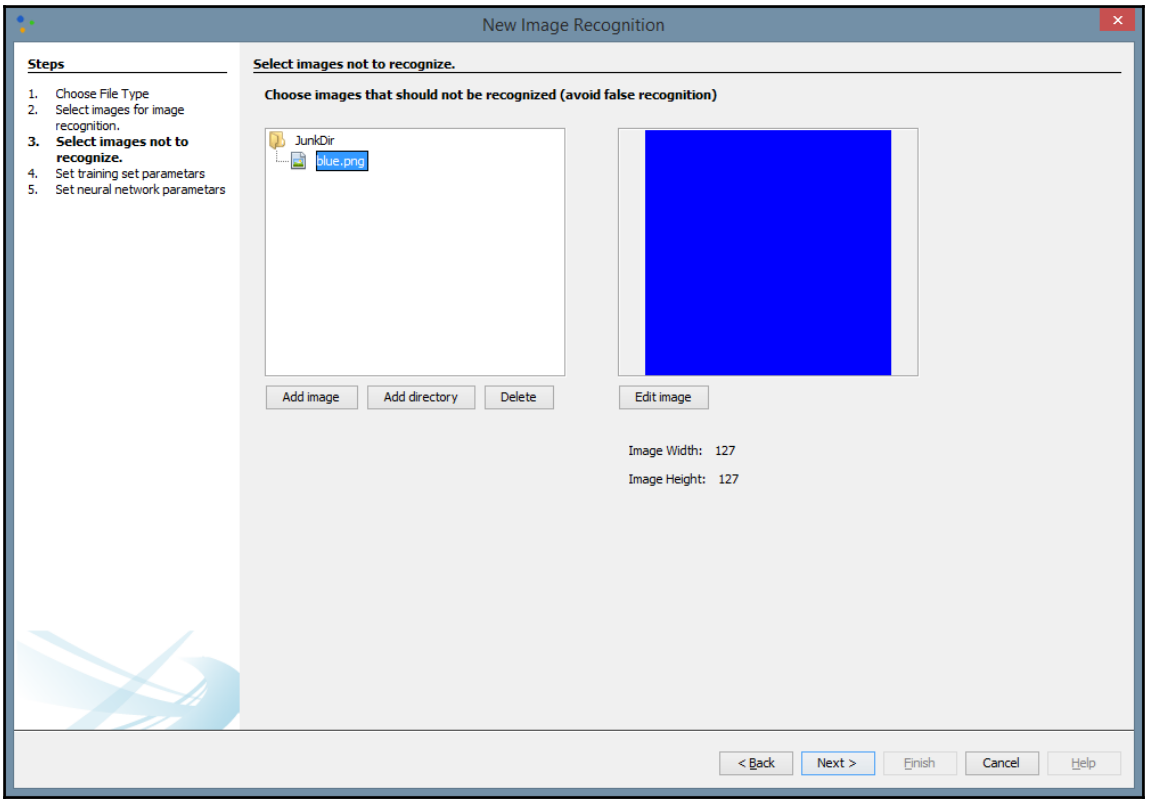
Next, we create a new file in our project. There are many types of project to choose from, but we will choose an **Image Recognition** type:



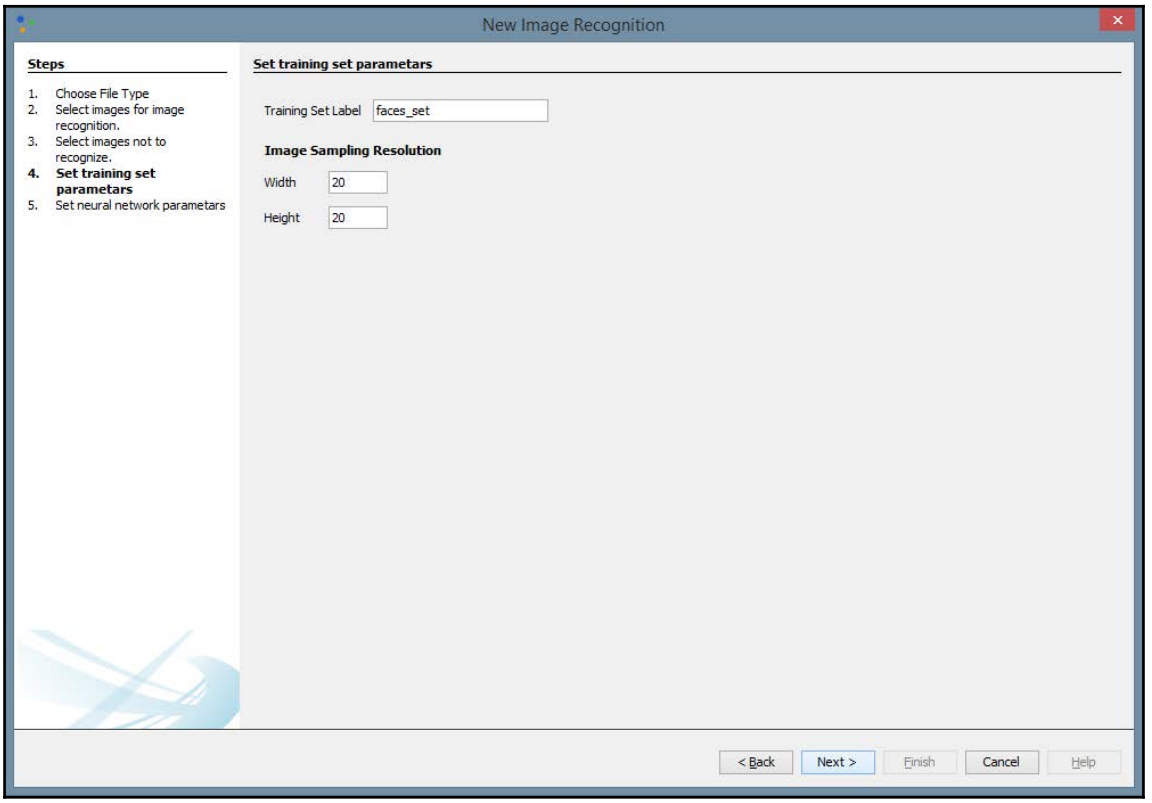
Click **Next** and then click **Add directory**. We have created a directory on our local machine and added several different black and white images of faces to use for training. These can be found by searching Google images or another search engine. The more quality images you have to train with, theoretically, the better your network will be:



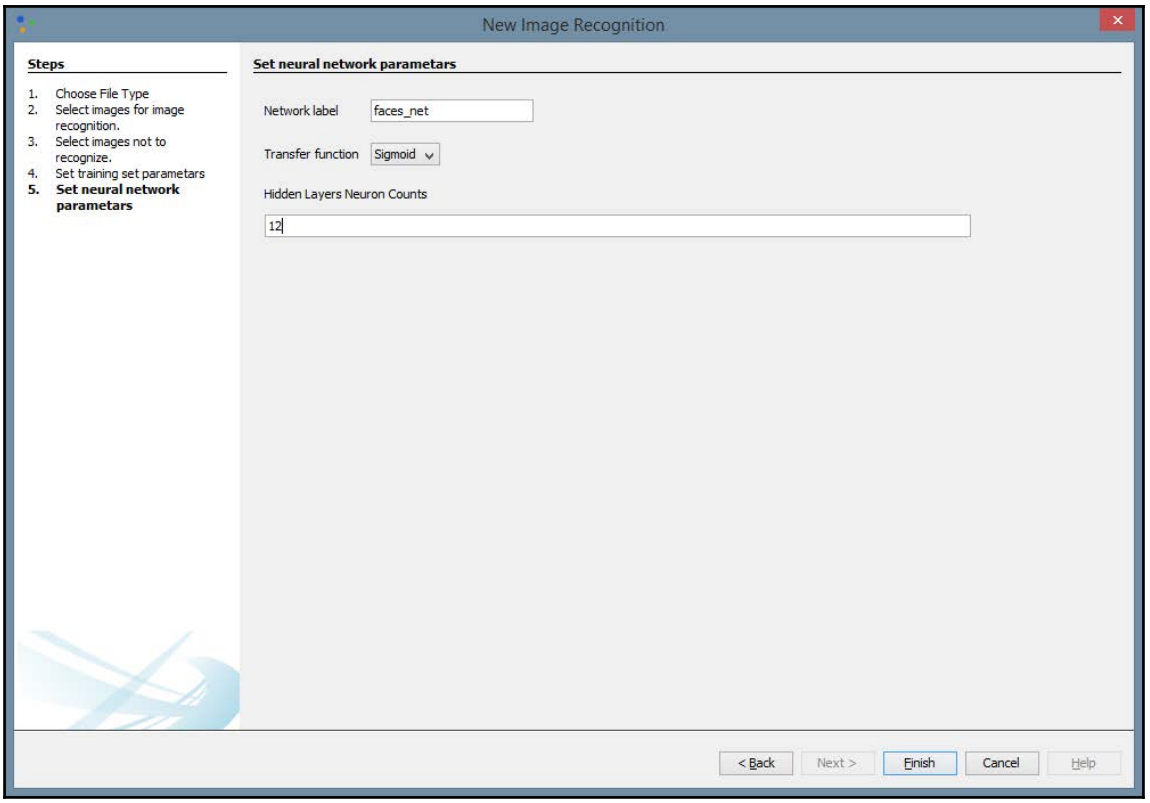
After you click **Next**, you will be directed to select an image to not recognize. You may need to try different images based upon the images you want to recognize. The image you select here will prevent false recognitions. We have chosen a simple blue square from another directory on our local machine, but if you are using other types of image for training, other color blocks may work better:



Next, we need to provide network training parameters. We also need to label our training dataset and set our resolution. A height and width of `20` is a good place to start, but you may want to change these values to improve your results. Some trial and error may be involved. The purpose of providing this information is to allow for image scaling. When we scale images to a smaller size, our network can process them and learn faster:

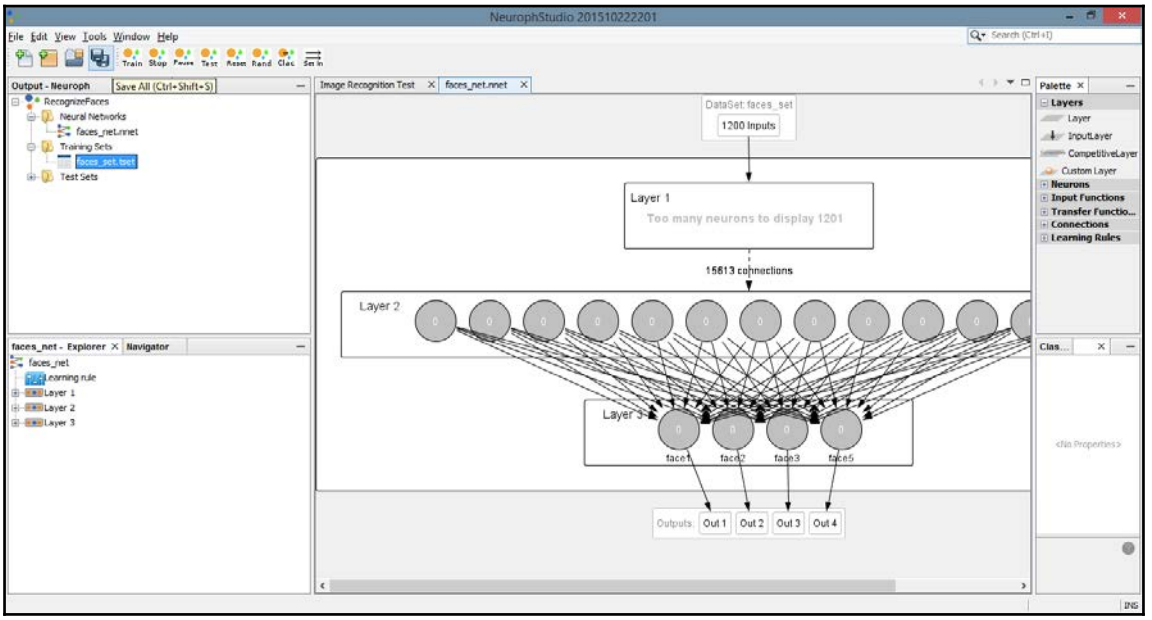


Finally, we can create our network. We assign a label to our network and define our **Transfer function**. The default function, **Sigmoid**, will work for most networks, but if your results are not optimal you may want to try **Tanh**. The default number of **Hidden Layers Neuron Counts** is `12`, and that is a good place to start. Be aware that increasing the number of neurons increases the time it takes to train the network and decreases your ability to generalize the network to other images. As with some of our previous values, some trial and error may be necessary to find the optimal settings for a given network. Select **Finish** when you are done:

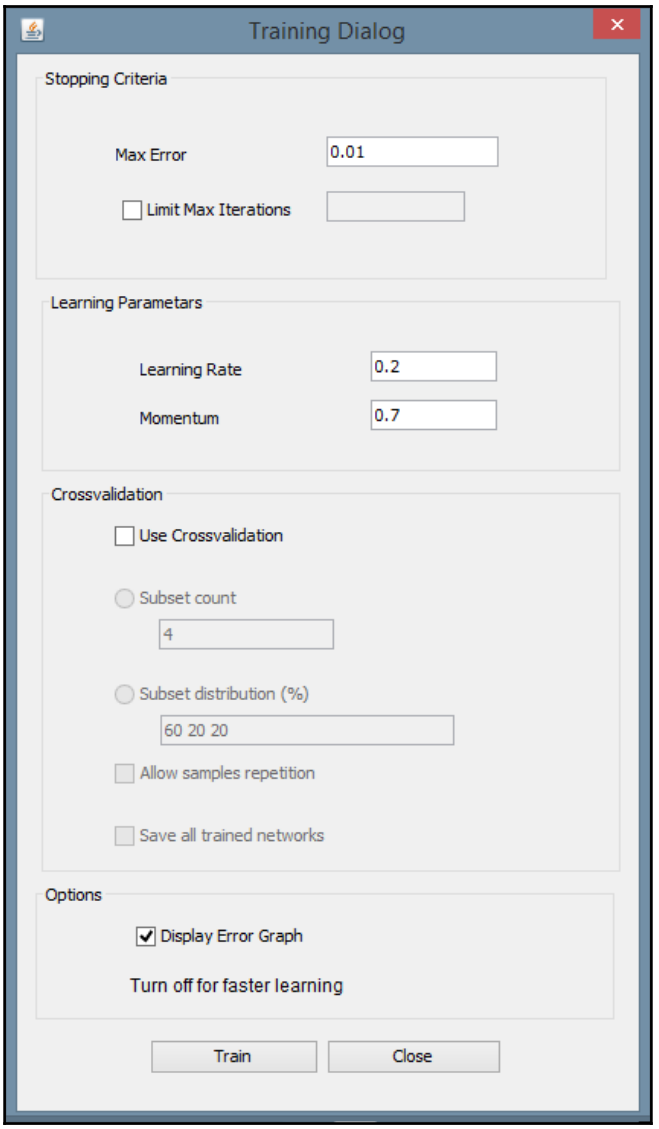


Training the model

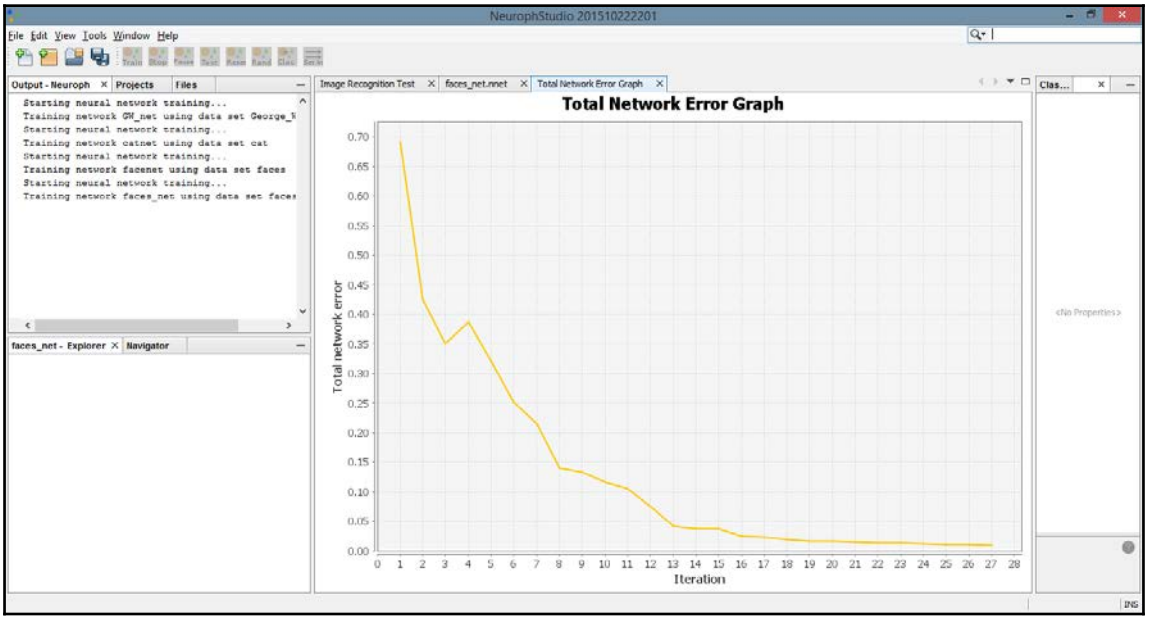
Once we have created our network, we need to train it. Begin by double-clicking on your neural network in the left pane. This is the file with the `.nnet` extension. When you do this, you will open a visual representation of the network in the main window. Then drag the dataset, with the file extension `.tsest`, from the left pane to the top node of the neural network. You will notice the description on the node change to the name of your dataset. Next, click the **Train** button, located in the top-left part of the window:



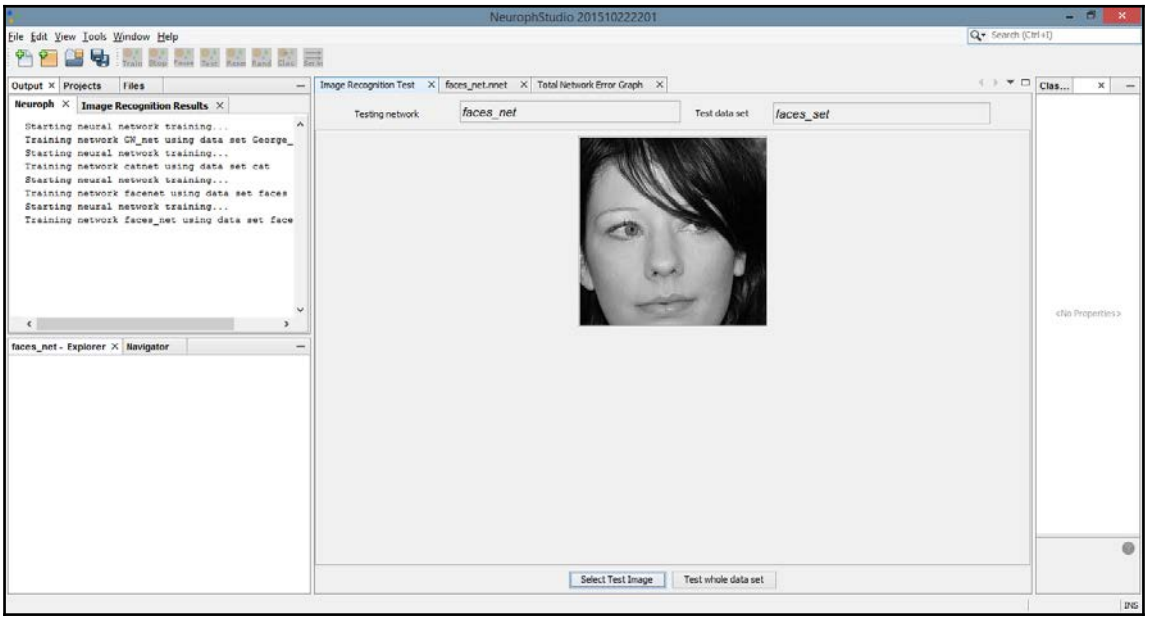
This will open a dialog box with settings for the training. You can leave the default values for **Max Error**, **Learning Rate**, and **Momentum**. Make sure the **Display Error Graph** box is checked. This allows you to see the improvement in the error rate as the training process continues:



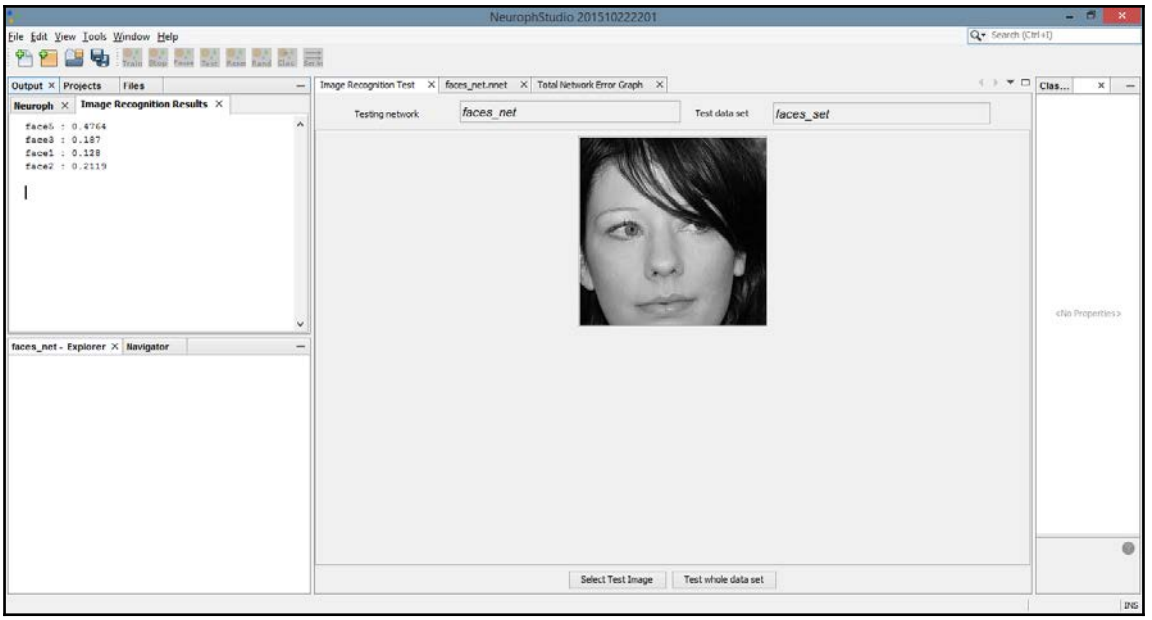
After you click the **Train** button, you should see an error graph similar to the following one:



Select the tab titled **Image Recognition Test**. Then click on the **Select Test Image** button. We have loaded a simple image of a face that was not included in our original dataset:



Locate the **Output** tab. It will be in the bottom or left pane and will display the results of comparing our test image with each image in the training set. The greater the number, the more closely our test image matches the image from our training set. The last image results in a greater output number than the first few comparisons. If we compare these images, they are more similar than the others in the dataset, and thus the network was able to create a more positive recognition of our test image:



We can now save our network for later use. Select **Save** from the **File** menu and then you can use the `.nnet` file in external applications. The following code example shows a simple technique for running test data through your pre-built neural network. The `NeuralNetwork` class is part of the Neuroph core package, and the `load` method allows you to load the trained network into your project. Notice we used our neural network name, `faces_net`. We then retrieve the plugin for our image recognition file. Next, we call the `recognizeImage` method with a new image, which must handle an `IOException`. Our results are stored in a `HashMap` and printed to the console:

```
NeuralNetwork iRNet = NeuralNetwork.load("faces_net.nnet");
ImageRecognitionPlugin iRFile
  = (ImageRecognitionPlugin)iRNet.getPlugin(
    ImageRecognitionPlugin.class);
try {
    HashMap<String, Double> newFaceMap
      = imageRecognition.recognizeImage(
        new File("testFace.jpg"));
    out.println(newFaceMap.toString());
} catch(IOException e) {
   // Handle exceptions
}
```

This process allows us to use a GUI editor application to create our network in a more visual environment, but then embed the trained network into our own applications.

Summary

In this chapter, we demonstrated many techniques for processing speech and images. This capability is becoming important, as electronic devices are increasingly embracing these communication mediums.

TTS was demonstrated using FreeTSS. This technique allows a computer to present results as speech as opposed to text. We learned how we can control the attributes of the voice used, such as its gender and age.

Recognizing speech is useful and helps bridge the human-computer interface gap. We demonstrated how CMUSphinx is used to recognize human speech. As there is often more than one way speech can be interpreted, we learned how the API can return various options. We also demonstrated how individual words are extracted, along with the relative confidence that the right word was identified.

Image processing is a critical aspect of many applications. We started our discussion of image processing by use Tess4J to extract text from an image. This process is sometimes referred to as OCR. We learned, as with many visual and audio data files, that the quality of the results is related to the quality of the image.

We also learned how to use OpenCV to identify faces within an image. Information about specific view of faces, such as frontal or profile views, are contained in XML files. These files were used to outline faces within an image. More than one face can be detected at a time.

It can be helpful to classify images and sometimes external tools are useful for this purpose. We examined Neuroph Studio and created a neural network designed to recognize and classify images. We then tested our network with images of human faces.

In the next chapter, we will learn how to speed up common data science applications using multiple processors.

11
Mathematical and Parallel Techniques for Data Analysis

The concurrent execution of a program can result in significant performance improvements. In this chapter, we will address the various techniques that can be used in data science applications. These can range from low-level mathematical calculations to higher-level API-specific options.

Always keep in mind that performance enhancement starts with ensuring that the correct set of application functionality is implemented. If the application does not do what a user expects, then the enhancements are for nought. The architecture of the application and the algorithms used are also more important than code enhancements. Always use the most efficient algorithm. Code enhancement should then be considered. We are not able to address the higher-level optimization issues in this chapter; instead, we will focus on code enhancements.

Many data science applications and supporting APIs use matrix operations to accomplish their tasks. Often these operations are buried within an API, but there are times when we may need to use these directly. Regardless, it can be beneficial to understand how these operations are supported. To this end, we will explain how matrix multiplication is handled using several different approaches.

Concurrent processing can be implemented using Java threads. A developer can use threads and thread pools to improve an application's response time. Many APIs will use threads when multiple CPUs or GPUs are not available, as is the case with **Aparapi**. We will not illustrate the use of threads here. However, the reader is assumed to have a basic knowledge of threads and thread pools.

The map-reduce algorithm is used extensively for data science applications. We will present a technique for achieving this type of parallel processing using Apache's Hadoop. Hadoop is a framework supporting the manipulation of large datasets, and can greatly decrease the required processing time for large data science projects. We will demonstrate a technique for calculating an average value for a sample set of data.

There are several well-known APIs that support multiple processors, including CUDA and OpenCL. CUDA is supported using **Java bindings for CUDA** (**JCuda**) (`http://jcuda.org/`). We will not demonstrate this technique directly here. However, many of the APIs we will use do support CUDA if it is available, such as DL4J. We will briefly discuss OpenCL and how it is supported in Java. It is worth nothing that the Aparapi API provides higher-level support, which may use either multiple CPUs or GPUs. A demonstration of Aparapi in support of matrix multiplication will be illustrated.

In this chapter, we will examine how multiple CPUs and GPUs can be harnessed to speed up data mining tasks. Many of the APIs we have used already take advantage of multiple processors or at least provide a means to enable GPU usage. We will introduce a number of these options in this chapter.

Concurrent processing is also supported extensively in the cloud. Many of the techniques discussed here are used in the cloud. As a result, we will not explicitly address how to conduct parallel processing in the cloud.

Implementing basic matrix operations

There are several different types of matrix operations, including simple addition, subtraction, scalar multiplication, and various forms of multiplication. To illustrate the matrix operations, we will focus on what is known as **matrix product**. This is a common approach that involves the multiplication of two matrixes to produce a third matrix.

Consider two matrices, *A* and *B*, where matrix *A* has *n* rows and *m* columns. Matrix *B* will have *m* rows and *p* columns. The product of *A* and *B*, written as *AB*, is an *n* row and *p* column matrix. The *m* entries of the rows of *A* are multiplied by the *m* entries of the columns of matrix *B*. This is more explicitly shown here, where:

$$A = \begin{vmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \cdots & \cdots & \cdots & \cdots \\ A_{n1} & A_{n1} & \cdots & A_{nm} \end{vmatrix} B = \begin{vmatrix} B_{11} & B_{12} & \cdots & B_{1p} \\ B_{21} & B_{22} & \cdots & B_{2p} \\ \cdots & \cdots & \cdots & \cdots \\ B_{m1} & B_{m1} & \cdots & B_{mp} \end{vmatrix}$$

Where the product is defined as follows:

$$\left(AB\right)_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}$$

We start with the declaration and initialization of the matrices. The variables `n`, `m`, `p` represent the dimensions of the matrices. The `A` matrix is `n` by `m`, the `B` matrix is `m` by `p`, and the `C` matrix representing the product is `n` by `p`:

```
int n = 4;
int m = 2;
int p = 3;

double A[][] = {
    {0.1950, 0.0311},
    {0.3588, 0.2203},
    {0.1716, 0.5931},
    {0.2105, 0.3242}};
double B[][] = {
    {0.0502, 0.9823, 0.9472},
    {0.5732, 0.2694, 0.916}};
double C[][] = new double[n][p];
```

The following code sequence illustrates the multiplication operation using nested `for` loops:

```
for (int i = 0; i < n; i++) {
    for (int k = 0; k < m; k++) {
        for (int j = 0; j < p; j++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

This following code sequence formats output to display our matrix:

```
out.println("\nResult");
for (int i = 0; i < n; i++) {
    for (int j = 0; j < p; j++) {
        out.printf("%.4f  ", C[i][j]);
    }
    out.println();
}
```

The result appears as follows:

```
Result
0.0276  0.1999  0.2132
0.1443  0.4118  0.5417
0.3486  0.3283  0.7058
0.1964  0.2941  0.4964
```

Later, we will demonstrate several alternative techniques for performing the same operation. Next, we will discuss how to tailor support for multiple processors using DL4J.

Using GPUs with DeepLearning4j

DeepLearning4j works with GPUs such as those provided by NVIDIA. There are options available that enable the use of GPUs, specify how many GPUs should be used, and control the use of GPU memory. In this section, we will show how you can use these options. This type of control is often available with other high-level APIs.

DL4J uses **n-dimensional arrays for Java** (**ND4J**) (`http://nd4j.org/`) to perform numerical computations. This is a library that supports n-dimensional array objects and other numerical computations, such as linear algebra and signal processing. It includes support for GPUs and is also integrated with Hadoop and Spark.

A **vector** is a one-dimensional array of numbers and is used extensively with neural networks. A vector is a type of mathematical structure called a **tensor**. A tensor is essentially a multidimensional array. We can think of a tensor as an array with three or more dimensions, and each dimension is called a **rank**.

There is often a need to map a multidimensional set of numbers to a one-dimensional array. This is done by flattening the array using a defined order. For example, with a two-dimensional array many systems will allocate the members of the array in row-column order. This means the first row is added to the vector, followed by the second vector, and then the third, and so forth. We will use this approach in the *Using the ND4J API* section.

To enable GPU use, the project's POM file needs to be modified. In the properties section of the POM file, the `nd4j.backend` tag needs to added or modified, as shown here:

```
<nd4j.backend>nd4j-cuda-7.5-platform</<nd4j.backend>
```

Models can be trained in parallel using the `ParallelWrapper` class. The training task is automatically distributed among available CPUs/GPUs. The model is used as an argument to the `ParallelWrapper` class' `Builder` constructor, as shown here:

```
ParallelWrapper parallelWrapper =
    new ParallelWrapper.Builder(aModel)
        // Builder methods...
        .build();
```

When executed, a copy of the model is used on each GPU. After the number of iterations is specified by the `averagingFrequency` method, the models are averaged and then the training process continues.

There are various methods that can be used to configure the class, as summarized in the following table:

Method	**Purpose**
`prefetchBuffer`	Specifies the size of a buffer used to pre-fetch data
`workers`	Specifies the number of workers to be used

`averageUpdaters`
`averagingFrequency`
`reportScoreAfterAveraging`
`useLegacyAveraging`
	Various methods to control how averaging is achieved

The number of workers should be greater than the number of GPUs available.

As with most computations, using a lower precision value will speed up the processing. This can be controlled using the `setDTypeForContext` method, as shown next. In this case, half precision is specified:

```
DataTypeUtil.setDTypeForContext(DataBuffer.Type.HALF);
```

This support and more details regarding optimization techniques can be found at `http://deeplearning4j.org/gpu`.

Using map-reduce

Map-reduce is a model for processing large sets of data in a parallel, distributed manner. This model consists of a `map` method for filtering and sorting data, and a `reduce` method for summarizing data. The map-reduce framework is effective because it distributes the processing of a dataset across multiple servers, performing mapping and reduction simultaneously on smaller pieces of the data. Map-reduce provides significant performance improvements when implemented in a multi-threaded manner. In this section, we will demonstrate a technique using Apache's Hadoop implementation. In the *Using Java 8 to perform map-reduce* section, we will discuss techniques for performing map-reduce using Java 8 streams.

Hadoop is a software ecosystem providing support for parallel computing. Map-reduce jobs can be run on Hadoop servers, generally set up as clusters, to significantly improve processing speeds. Hadoop has trackers that run map-reduce operations on nodes within a Hadoop cluster. Each node operates independently and the trackers monitor the progress and integrate the output of each node to generate the final output. The following image can be found at `http://www.developer.com/java/data/big-data-tool-map-reduce.html` and demonstrates the basic map-reduce model with trackers.

Using Apache's Hadoop to perform map-reduce

We are going to show you a very simple example of a map-reduce application here. Before we can use Hadoop, we need to download and extract Hadoop application files. The latest versions can be found at `http://hadoop.apache.org/releases.html`. We are using version 2.7.3 for this demonstration.

You will need to set your `JAVA_HOME` environment variable. Additionally, Hadoop is intolerant of long file paths and spaces within paths, so make sure you extract Hadoop to the simplest directory structure possible.

We will be working with a sample text file containing information about books. Each line of our tab-delimited file has the book title, author, and page count:

```
Moby Dick  Herman Melville  822
Charlotte's Web  E.B. White  189
The Grapes of Wrath  John Steinbeck  212
Jane Eyre  Charlotte Bronte  299
A Tale of Two Cities  Charles Dickens  673
War and Peace  Leo Tolstoy  1032
The Great Gatsby  F. Scott Fitzgerald  275
```

We are going to use a `map` function to extract the title and page count information and then a `reduce` function to calculate the average page count of the books in our dataset. To begin, create a new class, `AveragePageCount`. We will create two static classes within `AveragePageCount`, one to handle the map procedure and one to handle the reduction.

Writing the map method

First, we will create the `TextMapper` class, which will implement the `map` method. This class inherits from the `Mapper` class and has two private instance variables, `pages` and `bookTitle`. `pages` is an `IntWritable` object and `bookTitle` is a `Text` object. `IntWritable` and `Text` are used because these objects will need to be serialized to a byte stream before they can be transmitted to the servers for processing. These objects take up less space and transfer faster than the comparable `int` or `String` objects:

```
public static class TextMapper
        extends Mapper<Object, Text, Text, IntWritable> {

    private final IntWritable pages = new IntWritable();
    private final Text bookTitle = new Text();

}
```

Within our `TextMapper` class we create the `map` method. This method takes three parameters: the `key` object, a `Text` object, `bookInfo`, and the `Context`. The key allows the tracker to map each particular object back to the correct job. The `bookInfo` object contains the text or string data about each book. `Context` holds information about the entire system and allows the method to report on progress and update values within the system.

Within the `map` method, we use the `split` method to break each piece of book information into an array of `String` objects. We set our `bookTitle` variable to position `0` of the array and set `pages` to the value stored in position `2`, after parsing it as an integer. We can then write out our book title and page count information through the context and update our entire system:

```
public void map(Object key, Text bookInfo, Context context)
 throws IOException, InterruptedException {
        String[] book = bookInfo.toString().split("\t");
        bookTitle.set(book[0]);
        pages.set(Integer.parseInt(book[2]));
        context.write(bookTitle, pages);
    }
```

Writing the reduce method

Next, we will write our `AverageReduce` class. This class extends the `Reducer` class and will perform the reduction processes to calculate our average page count. We have created four variables for this class: a `FloatWritable` object to store our average page count, a float `average` to hold our temporary average, a float `count` to count how many books exist in our dataset, and an integer `sum` to add up the page counts:

```
public static class AverageReduce
        extends Reducer<Text, IntWritable, Text, FloatWritable> {

    private final FloatWritable finalAvg = new FloatWritable();
    Float average = 0f;
    Float count = 0f;
    int sum = 0;

}
```

Within our `AverageReduce` class we will create the `reduce` method. This method takes as input a `Text` key, an `Iterable` object holding writeable integers representing the page counts, and the `Context`. We use our iterator to process the page counts and add each to our sum. We then calculate the average and set the value of `finalAvg`. This information is paired with a `Text` object label and written to the `Context`:

```
    public void reduce(Text key, Iterable<IntWritable> pageCnts,
        Context context)
            throws IOException, InterruptedException {

    for (IntWritable cnt : pageCnts) {
        sum += cnt.get();
```

```
        }
        count += 1;
        average = sum / count;
        finalAvg.set(average);
        context.write(new Text("Average Page Count = "), finalAvg);
    }
```

Creating and executing a new Hadoop job

We are now ready to create our `main` method in the same class and execute our map-reduce processes. To do this, we need to create a new `Configuration` object and a new `Job`. We then set up the significant classes to use in our application.

```
    public static void main(String[] args) throws Exception {
        Configuration con = new Configuration();
        Job bookJob = Job.getInstance(con, "Average Page Count");
        ...
    }
```

We set our main class, `AveragePageCount`, in the `setJarByClass` method. We specify our `TextMapper` and `AverageReduce` classes using the `setMapperClass` and `setReducerClass` methods, respectively. We also specify that our output will have a text-based key and a writeable integer using the `setOutputKeyClass` and `setOutputValueClass` methods:

```
        bookJob.setJarByClass(AveragePageCount.class);
        bookJob.setMapperClass(TextMapper.class);
        bookJob.setReducerClass(AverageReduce.class);
        bookJob.setOutputKeyClass(Text.class);
        bookJob.setOutputValueClass(IntWritable.class);
```

Finally, we create new input and output paths using the `addInputPath` and `setOutputPath` methods. These methods both take our `Job` object as the first parameter and a `Path` object representing our input and output file locations as the second parameter. We then call `waitForCompletion`. Our application exits once this call returns true:

```
        FileInputFormat.addInputPath(bookJob, new Path("C:/Hadoop/books.txt"));
        FileOutputFormat.setOutputPath(bookJob, new
            Path("C:/Hadoop/BookOutput"));
        if (bookJob.waitForCompletion(true)) {
            System.exit(0);
        }
```

To execute the application, open a command prompt and navigate to the directory containing our `AveragePageCount.class` file. We then use the following command to execute our sample application:

```
hadoop AveragePageCount
```

While our task is running, we see updated information about our process output to the screen. A sample of our output is shown as follows:

```
...
File System Counters
    FILE: Number of bytes read=1132
    FILE: Number of bytes written=569686
    FILE: Number of read operations=0
    FILE: Number of large read operations=0
    FILE: Number of write operations=0
Map-Reduce Framework
    Map input records=7
    Map output records=7
    Map output bytes=136
    Map output materialized bytes=156
    Input split bytes=90
    Combine input records=0
    Combine output records=0
    Reduce input groups=7
    Reduce shuffle bytes=156
    Reduce input records=7
    Reduce output records=7
    Spilled Records=14
    Shuffled Maps =1
    Failed Shuffles=0
    Merged Map outputs=1
    GC time elapsed (ms)=11
    Total committed heap usage (bytes)=536870912
Shuffle Errors
    BAD_ID=0
    CONNECTION=0
    IO_ERROR=0
    WRONG_LENGTH=0
    WRONG_MAP=0
    WRONG_REDUCE=0
File Input Format Counters
    Bytes Read=249
File Output Format Counters
     Bytes Written=216
```

If we open the `BookOutput` directory created on our local machine, we find four new files. Use a text editor to open `part-r-00000`. This file contains information about the average page count as it was calculated using parallel processes. A sample of this output follows:

```
Average Page Count =  673.0
Average Page Count =   431.0
Average Page Count =   387.0
Average Page Count =   495.75
Average Page Count =   439.0
Average Page Count =   411.66666
Average Page Count =   500.2857
```

Notice how the average changes as each individual process is combined with the other reduction processes. This has the same effect as calculating the average of the first two books first, then adding in the third book, then the fourth, and so on. The advantage here of course is that the averaging is done in a parallel manner. If we had a huge dataset, we should expect to see a noticeable advantage in execution time. The last line of `BookOutput` reflects the correct and final average of all seven page counts.

Various mathematical libraries

There are numerous mathematical libraries available for Java use. In this section, we will provide a quick and high-level overview of several libraries. These libraries do not necessarily automatically support multiple processors. In addition, the intent of this section is to provide some insight into how these libraries can be used. In most cases, they are relatively easy to use.

A list of Java mathematical libraries is found at `https://en.wikipedia.org/wiki/List_of_numerical_libraries#Java` and `https://java-matrix.org/`. We will demonstrate the use of the jblas, Apache Commons Math, and the ND4J libraries.

Using the jblas API

The jblas API (`http://jblas.org/`) is a math library supporting Java. It is based on **Basic Linear Algebra Subprograms** (**BLAS**) (`http://www.netlib.org/blas/`) and **Linear Algebra Package** (**LAPACK**) (`http://www.netlib.org/lapack/`), which are standard libraries for fast arithmetic calculation. The jblas API provides a wrapper around these libraries.

The following is a demonstration of how matrix multiplication is performed. We start with the matrix definitions:

```
DoubleMatrix A = new DoubleMatrix(new double[][]{
    {0.1950, 0.0311},
    {0.3588, 0.2203},
    {0.1716, 0.5931},
    {0.2105, 0.3242}});

DoubleMatrix B = new DoubleMatrix(new double[][]{
    {0.0502, 0.9823, 0.9472},
    {0.5732, 0.2694, 0.916}});
DoubleMatrix C;
```

The actual statement to perform multiplication is quite short, as shown next. The `mmul` method is executed against the `A` matrix, where the `B` array is passed as an argument:

```
C = A.mmul(B);
```

The resulting `C` matrix is then displayed:

```
for(int i=0; i<C.getRows(); i++) {
    out.println(C.getRow(i));
}
```

The output should be as follows:

```
[0.027616, 0.199927, 0.213192]
[0.144288, 0.411798, 0.541650]
[0.348579, 0.328344, 0.705819]
[0.196399, 0.294114, 0.496353]
```

This library is fairly easy to use and supports an extensive set of arithmetic operations.

Using the Apache Commons math API

The Apache Commons math API (`http://commons.apache.org/proper/commons-math/`) supports a large number of mathematical and statistical operations. The following example illustrates how to perform matrix multiplication.

We start with the declaration and initialization of the `A` and `B` matrices:

```
double[][] A = {
    {0.1950, 0.0311},
    {0.3588, 0.2203},
    {0.1716, 0.5931},
```

```
        {0.2105, 0.3242}};

    double[][] B = {
        {0.0502, 0.9823, 0.9472},
        {0.5732, 0.2694, 0.916}};
```

Apache Commons uses the `RealMatrix` class to hold a matrix. In the following code sequence, the corresponding matrices for the `A` and `B` matrices are created using the `Array2DRowRealMatrix` constructor:

```
RealMatrix aRealMatrix = new Array2DRowRealMatrix(A);
RealMatrix bRealMatrix = new Array2DRowRealMatrix(B);
```

The multiplication is straightforward using the multiply method, as shown next:

```
RealMatrix cRealMatrix = aRealMatrix.multiply(bRealMatrix);
```

The next `for` loop will display the following results:

```
for (int i = 0; i < cRealMatrix.getRowDimension(); i++) {
    out.println(cRealMatrix.getRowVector(i));
}
```

The output should be as follows:

```
{0.02761552; 0.19992684; 0.2131916}
{0.14428772; 0.41179806; 0.54165016}
{0.34857924; 0.32834382; 0.70581912}
{0.19639854; 0.29411363; 0.4963528}
```

Using the ND4J API

ND4J (`http://nd4j.org/`) is the library used by DL4J to perform arithmetic operations. The library is also available for direct use. In this section, we will demonstrate how matrix multiplication is performed using the `A` and `B` matrices.

Before we can perform the multiplication, we need to flatten the matrices to vectors. The following declares and initializes these vectors:

```
double[] A = {
    0.1950, 0.0311,
    0.3588, 0.2203,
    0.1716, 0.5931,
    0.2105, 0.3242};

double[] B = {
```

```
        0.0502, 0.9823, 0.9472,
        0.5732, 0.2694, 0.916};
```

The `Nd4j` class' `create` method creates an `INDArray` instance given a vector and dimension information. The first argument of the method is the vector. The second argument specifies the dimensions of the matrix. The last argument specifies the order the rows and columns are laid out. This order is either row-column major as exemplified by `c`, or column-row major order as used by FORTRAN. Row-column order means the first row is allocated to the vector, followed by the second row, and so forth.

In the following code sequence `2INDArray` instances are created using the `A` and `B` vectors. The first is a `4` row, `2` column matrix using row-major order as specified by the third argument, `c`. The second `INDArray` instance represents the `B` matrix. If we wanted to use column-row ordering, we would use an `f` instead.

```
INDArray aINDArray = Nd4j.create(A,new int[]{4,2},'c');
INDArray bINDArray = Nd4j.create(B,new int[]{2,3},'c');
```

The `C` array, represented by `cINDArray`, is then declared and assigned the result of the multiplication. The `mmul` performs the operation:

```
INDArray cINDArray;
cINDArray = aINDArray.mmul(bINDArray);
```

The following sequence displays the results using the `getRow` method:

```
for(int i=0; i<cINDArray.rows(); i++) {
   out.println(cINDArray.getRow(i));
}
```

The output should be as follows:

```
[0.03, 0.20, 0.21]
[0.14, 0.41, 0.54]
[0.35, 0.33, 0.71]
[0.20, 0.29, 0.50]
```

Next, we will provide an overview of the OpenCL API that provide supports for concurrent operations on a number of platforms.

Using OpenCL

Open Computing Language (**OpenCL**) (`https://www.khronos.org/opencl/`) supports programs that execute across heterogeneous platforms, that is, platforms potentially using different vendors and architectures. The platforms can use different processing units, including **Central Processing Unit** (**CPU**), **Graphical Processing Unit** (**GPU**), **Digital Signal Processor** (**DSP**), **Field-Programmable Gate Array** (**FPGA**), and other types of processors.

OpenCL uses a C99-based language to program the devices, providing a standard interface for programming concurrent behavior. OpenCL supports an API that allows code to be written in different languages. For Java, there are several APIs that support the development of OpenCL based languages:

- **Java bindings for OpenCL** (**JOCL**) (`http://www.jocl.org/`) - This is a binding to the original OpenCL C implementation and can be verbose.
- **JavaCl** (`https://code.google.com/archive/p/javacl/`) - Provides an object-oriented interface to JOCL.
- **Java OpenCL** (`http://jogamp.org/jocl/www/`) - Also provides an object-oriented abstraction of JOCL. It is not intended for client use.
- The **Lightweight Java Game Library** (**LWJGL**) (`https://www.lwjgl.org/`) - Also provides support for OpenCL and is oriented toward GUI applications.

In addition, Aparapi provides higher-level access to OpenCL, thus avoiding some of the complexity involved in creating OpenCL applications.

Code that runs on a processor is encapsulated in a kernel. Multiple kernels will execute in parallel on different computing devices. There are different levels of memory supported by OpenCL. A specific device may not support each level. The levels include:

- **Global memory** - Shared by all computing units
- **Read-only memory** - Generally not writable
- **Local memory** - Shared by a group of computing units
- **Per-element private memory** - Often a register

OpenCL applications require a considerable amount of initial code to be useful. This complexity does not permit us to provide a detailed example of its use. However, the Aparapi section does provide some feel for how OpenCL applications are structured.

Using Aparapi

Aparapi (`https://github.com/aparapi/aparapi`) is a Java library that supports concurrent operations. The API supports code running on GPUs or CPUs. GPU operations are executed using OpenCL, while CPU operations use Java threads. The user can specify which computing resource to use. However, if GPU support is not available, Aparapi will revert to Java threads.

The API will convert Java byte codes to OpenCL at runtime. This makes the API largely independent from the graphics card used. The API was initially developed by AMD but has been released as open source. This is reflected in the basic package name, `com.amd.aparari`. Aparapi offers a higher level of abstraction than provided by OpenCL.

Aparapi code is located in a class derived from the `Kernel` class. Its `execute` method will start the operations. This will result in an internal call to a `run` method, which needs to be overridden. It is within the `run` method that concurrent code is placed. The `run` method is executed multiple times on different processors.

Due to OpenCL limitations, we are unable to use inheritance or method overloading. In addition, it does not like `println` in the `run` method, since the code may be running on a GPI. Aparapi only supports one-dimensional arrays. Arrays using two or more dimensions need to be flattened to a one dimension array. The support for double values is dependent on the OpenCL version and GPU configuration.

When a Java thread pool is used, it allocates one thread per CPU core. The kernel containing the Java code is cloned, one copy per thread. This avoids the need to access data across a thread. Each thread has access to information, such as a global ID, to assist in the code execution. The kernel will wait for all of the threads to complete.

Aparapi downloads can be found at `https://github.com/aparapi/aparapi/releases`.

Creating an Aparapi application

The basic framework for an Aparapi application is shown next. It consists of a `Kernel` derived class where the `run` method is overridden. In this example, the `run` method will perform scalar multiplication. This operation involves multiplying each element of a vector by some value.

The `ScalarMultiplicationKernel` extends the `Kernel` class. It possesses two instance variables used to hold the matrices for input and output. The constructor will initialize the matrices. The `run` method will perform the actual computations, and the `displayResult` method will show the results of the multiplication:

```
public class ScalarMultiplicationKernel extends Kernel {
    float[] inputMatrix;
    float outputMatrix [];

    public ScalarMultiplicationKernel(float inputMatrix[]) {
        ...
    }

    @Override
    public void run() {
        ...
    }

    public void displayResult() {
        ...
    }
}
```

The constructor is shown here:

```
public ScalarMultiplicationKernel(float inputMatrix[]) {
    this.inputMatrix = inputMatrix;
    outputMatrix = new float[this.inputMatrix.length];
}
```

In the `run` method, we use a global ID to index into the matrix. This code is executed on each computation unit, for example, a GPU or thread. A unique global ID is provided to each computational unit, allowing the code to access a specific element of the matrix. In this example, each element of the input matrix is multiplied by `2` and then assigned to the corresponding element of the output matrix:

```
public void run() {
    int globalID = this.getGlobalId();
    outputMatrix[globalID] = 2.0f * inputMatrix[globalID];
}
```

The `displayResult` method simply displays the contents of the `outputMatrix` array:

```
public void displayResult() {
    out.println("Result");
    for (float element : outputMatrix) {
        out.printf("%.4f ", element);
    }
    out.println();
}
```

To use this kernel, we need to declare variables for the `inputMatrix` and its `size`. The `size` will be used to control how many kernels to execute:

```
float inputMatrix[] = {3, 4, 5, 6, 7, 8, 9};
int size = inputMatrix.length;
```

The kernel is then created using the input matrix followed by the invocation of the `execute` method. This method starts the process and will eventually invoke the `Kernel` class' `run` method based on the `execute` method's argument. This argument is referred to as the pass ID. While not used in this example, we will use it in the next section. When the process is complete, the resulting output matrix is displayed and the `dispose` method is called to stop the process:

```
ScalarMultiplicationKernel kernel =
        new ScalarMultiplicationKernel(inputMatrix);
kernel.execute(size);
kernel.displayResult();
kernel.dispose();
```

When this application is executed we will get the following output:

```
6.0000 8.0000 10.0000 12.0000 14.0000 16.0000 18.000
```

We can specify the execution mode using the Kernel class' `setExecutionMode` method, as shown here:

```
kernel.setExecutionMode(Kernel.EXECUTION_MODE.GPU);
```

However, it is best to let Aparapi determine the execution mode. The following table summarizes the execution modes available:

Execution mode	**Meaning**
`Kernel.EXECUTION_MODE.NONE`	Does not specify mode
`Kernel.EXECUTION_MODE.CPU`	Use CPU
`Kernel.EXECUTION_MODE.GPU`	Use GPU
`Kernel.EXECUTION_MODE.JTP`	Use Java threads
`Kernel.EXECUTION_MODE.SEQ`	Use single loop (for debugging purposes)

Next, we will demonstrate how we can use Aparapi to perform dot product matrix multiplication.

Using Aparapi for matrix multiplication

We will use the matrices as used in the *Implementing basic matrix operations* section. We start with the declaration of the `MatrixMultiplicationKernel` class, which contains the vector declarations, a constructor, the `run` method, and a `displayResults` method. The vectors for matrices `A` and `B` have been flattened to one-dimensional arrays by allocating the matrices in row-column order:

```
class MatrixMultiplicationKernel extends Kernel {
    float[] vectorA = {
        0.1950f, 0.0311f, 0.3588f,
        0.2203f, 0.1716f, 0.5931f,
        0.2105f, 0.3242f};
    float[] vectorB = {
        0.0502f, 0.9823f, 0.9472f,
        0.5732f, 0.2694f, 0.916f};
    float[] vectorC;
    int n;
    int m;
    int p;

    @Override
    public void run() {
        ...
    }

    public MatrixMultiplicationKernel(int n, int m, int p) {
        ...
```

```
    }

    public void displayResults () {
        ...
    }
}
```

The `MatrixMultiplicationKernel` constructor assigns values for the matrices' dimensions and allocates memory for the result stored in `vectorC`, as shown here:

```
public MatrixMultiplicationKernel(int n, int m, int p) {
    this.n = n;
    this.p = p;
    this.m = m;
    vectorC = new float[n * p];
}
```

The run method uses a global ID and a pass ID to perform the matrix multiplication. The pass ID is specified as the second argument of the `Kernel` class' `execute` method, as we will see shortly. This value allows us to advance the column index for `vectorC`. The vector indexes map to the corresponding row and column positions of the original matrices:

```
public void run() {
    int i = getGlobalId();
    int j = this.getPassId();
    float value = 0;
    for (int k = 0; k < p; k++) {
        value += vectorA[k + i * m] * vectorB[k * p + j];
    }
    vectorC[i * p + j] = value;
}
```

The `displayResults` method is shown as follows:

```
public void displayResults() {
    out.println("Result");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < p; j++) {
            out.printf("%.4f  ", vectorC[i * p + j]);
        }
        out.println();
    }
}
```

The kernel is started in the same way as in the previous section. The `execute` method is passed the number of kernels that should be created and an integer indicating the number of passes to make. The number of passes is used to control the index into the `vectorA` and `vectorB` arrays:

```
MatrixMultiplicationKernel kernel = new MatrixMultiplicationKernel(n, m,
   p);kernel.execute(6, 3);kernel.displayResults();
kernel.dispose();
```

When this example is executed, you will get the following output:

```
Result
0.0276  0.1999  0.2132
0.1443  0.4118  0.5417
0.3486  0.3283  0.7058
0.1964  0.2941  0.4964
```

Next, we will see how Java 8 additions can contribute to solving math-intensive problems in a parallel manner.

Using Java 8 streams

The release of Java 8 came with a number of important enhancements to the language. The two enhancements of interest to us include lambda expressions and streams. A lambda expression is essentially an anonymous function that adds a functional programming dimension to Java. The concept of streams, as introduced in Java 8, does not refer to IO streams. Instead, you can think of it as a sequence of objects that can be generated and manipulated using a fluent style of programming. This style will be demonstrated shortly.

As with most APIs, programmers must be careful to consider the actual execution performance of their code using realistic test cases and environments. If not used properly, streams may not actually provide performance improvements. In particular, parallel streams, if not crafted carefully, can produce incorrect results.

We will start with a quick introduction to lambda expressions and streams. If you are familiar with these concepts you may want to skip over the next section.

Understanding Java 8 lambda expressions and streams

A lambda expression can be expressed in several different forms. The following illustrates a simple lambda expression where the symbol, `->`, is the lambda operator. This will take some value, `e`, and return the value multiplied by two. There is nothing special about the name `e`. Any valid Java variable name can be used:

```
e -> 2 * e
```

It can also be expressed in other forms, such as the following:

```
(int e) -> 2 * e
(double e) -> 2 * e
(int e) -> {return 2 * e;
```

The form used depends on the intended value of `e`. Lambda expressions are frequently used as arguments to a method, as we will see shortly.

A stream can be created using a number of techniques. In the following example, a stream is created from an array. The `IntStream` interface is a type of stream that uses integers. The `Arrays` class' `stream` method converts an array into a stream:

```
IntStream stream = Arrays.stream(numbers);
```

We can then apply various `stream` methods to perform an operation. In the following statement, the `forEach` method will simply display each integer in the stream:

```
stream.forEach(e -> out.printf("%d  ", e));
```

There are a variety of `stream` methods that can be applied to a stream. In the following example, the `mapToDouble` method will take an integer, multiply it by `2`, and then return it as a `double`. The `forEach` method will then display these values:

```
stream
        .mapToDouble(e-> 2 * e)
        .forEach(e -> out.printf("%.4f  ", e));
```

The cascading of method invocations is referred to as **fluent programing**.

Using Java 8 to perform matrix multiplication

Here, we will illustrate how streams can be used to perform matrix multiplication. The definitions of the `A`, `B`, and `C` matrices are the same as declared in the *Implementing basic matrix operations* section. They are duplicated here for your convenience:

```
double A[][] = {
    {0.1950, 0.0311},
    {0.3588, 0.2203},
    {0.1716, 0.5931},
    {0.2105, 0.3242}};
double B[][] = {
    {0.0502, 0.9823, 0.9472},
    {0.5732, 0.2694, 0.916}};
double C[][] = new double[n][p];
```

The following sequence is a stream implementation of matrix multiplication. A detailed explanation of the code follows:

```
C = Arrays.stream(A)
        .parallel()
        .map(AMatrixRow -> IntStream.range(0, B[0].length)
                .mapToDouble(i -> IntStream.range(0, B.length)
                        .mapToDouble(j -> AMatrixRow[j] * B[j][i])
                        .sum()
                ).toArray()).toArray(double[][]::new);
```

The first `map` method, shown as follows, creates a stream of double vectors representing the `4` rows of the `A` matrix. The `range` method will return a list of stream elements ranging from its first argument to the second argument.

```
.map(AMatrixRow -> IntStream.range(0, B[0].length)
```

The variable `i` corresponds to the numbers generated by the second `range` method, which corresponds to the number of rows in the `B` matrix (`2`). The variable `j` corresponds to the numbers generated by the third `range` method, representing the number of columns of the `B` matrix (`3`).

At the heart of the statement is the matrix multiplication, where the `sum` method calculates the sum:

```
.mapToDouble(j -> AMatrixRow[j] * B[j][i])
.sum()
```

The last part of the expression creates the two-dimensional array for the C matrix. The operator, `::new`, is called a method reference and is a shorter way of invoking the new operator to create a new object:

```
).toArray()).toArray(double[][]::new);
```

The `displayResult` method is as follows:

```
public void displayResult() {
    out.println("Result");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < p; j++) {
            out.printf("%.4f  ", C[i][j]);
        }
        out.println();
    }
}
```

The output of this sequence follows:

```
Result
0.0276  0.1999  0.2132
0.1443  0.4118  0.5417
0.3486  0.3283  0.7058
0.1964  0.2941  0.4964
```

Using Java 8 to perform map-reduce

In the next section, we will use Java 8 streams to perform a map-reduce operation similar to the one demonstrated using Hadoop in the *Using map-reduce* section. In this example, we will use a `Stream` of `Book` objects. We will then demonstrate how to use the Java 8 `reduce` and `average` methods to get our total page count and average page count.

Rather than begin with a text file, as we did in the Hadoop example, we have created a `Book` class with title, author, and page-count fields. In the `main` method of the `driver` class, we have created new instances of `Book` and added them to an `ArrayList` called `books`. We have also created a `double` value `average` to hold our average, and initialized our variable `totalPg` to zero:

```
ArrayList<Book> books = new ArrayList<>();
double average;
int totalPg = 0;

books.add(new Book("Moby Dick", "Herman Melville", 822));
books.add(new Book("Charlotte's Web", "E.B. White", 189));
```

```
books.add(new Book("The Grapes of Wrath", "John Steinbeck", 212));
books.add(new Book("Jane Eyre", "Charlotte Bronte", 299));
books.add(new Book("A Tale of Two Cities", "Charles Dickens", 673));
books.add(new Book("War and Peace", "Leo Tolstoy", 1032));
books.add(new Book("The Great Gatsby", "F. Scott Fitzgerald",    275));
```

Next, we perform a map and reduce operation to calculate the total number of pages in our set of books. To accomplish this in a parallel manner, we use the `stream` and `parallel` methods. We then use the `map` method with a lambda expression to accumulate all of the page counts from each `Book` object. Finally, we use the `reduce` method to merge our page counts into one final value, which is to be assigned to `totalPg`:

```
totalPg = books
        .stream()
        .parallel()
        .map((b) -> b.pgCnt)
        .reduce(totalPg, (accumulator, _item) -> {
            out.println(accumulator + " " +_item);
            return accumulator + _item;
                });
```

Notice in the preceding `reduce` method we have chosen to print out information about the reduction operation's cumulative value and individual items. The `accumulator` represents the aggregation of our page counts. The `_item` represents the individual task within the map-reduce process undergoing reduction at any given moment.

In the output that follows, we will first see the `accumulator` value stay at zero as each individual book item is processed. Gradually, the `accumulator` value increases. The final operation is the reduction of the values `1223` and `2279`. The sum of these two numbers is `3502`, or the total page count for all of our books:

```
0 822
0 189
0 299
0 673
0 212
299 673
0 1032
0 275
1032 275
972 1307
189 212
822 401
1223 2279
```

Next, we will add code to calculate the average page count of our set of books. We multiply our `totalPg` value, determined using map-reduce, by `1.0` to prevent truncation when we divide by the integer returned by the `size` method. We then print out `average`.

```
average = 1.0 * totalPg / books.size();
out.printf("Average Page Count: %.4f\n", average);
```

Our output is as follows:

```
Average Page Count: 500.2857
```

We could have used Java 8 streams to calculate the average directly using the `map` method. Add the following code to the `main` method. We use `parallelStream` with our `map` method to simultaneously get the page count for each of our books. We then use `mapToDouble` to ensure our data is of the correct type to calculate our average. Finally, we use the `average` and `getAsDouble` methods to calculate our average page count:

```
average = books
        .parallelStream()
        .map(b -> b.pgCnt)
        .mapToDouble(s -> s)
        .average()
        .getAsDouble();
out.printf("Average Page Count: %.4f\n", average);
```

Then we print out our average. Our output, identical to our previous example, is as follows:

```
Average Page Count: 500.2857
```

These techniques made use of Java 8 capabilities related to the map-reduce framework to solve numeric problems. This type of process can also be applied to other types of data, including text-based data. The true benefit is seen when these processes handle extremely large datasets within a greatly reduced time frame.

Summary

Data science uses math extensively to analyze problems. There are numerous Java math libraries available, many of which support concurrent operations. In this chapter, we introduced a number of libraries and techniques to provide some insight into how they can be used to support and improve the performance of applications.

We started with a discussion of how simple matrix multiplication is performed. A basic Java implementation was presented. In later sections, we duplicated the implementation using other APIs and technologies.

Many higher level APIs, such as DL4J, support a number of useful data analysis techniques. Beneath these APIs often lies concurrent support for multiple CPUs and GPUs. Sometimes this support is configurable, as is the case for DL4J. We briefly discussed how we can configure ND4J to support multiple processors.

The map-reduce algorithm has found extensive use in the data science community. We took advantage of the parallel processing power of this framework to calculate the average of a given set of values, the page counts for a set of books. This technique used Apache's Hadoop to perform the map and reduce functions.

Mathematical techniques are supported by a large number of libraries. Many of these libraries do not directly support parallel operations. However, understanding what is available and how they can be used is important. To that end, we demonstrated how three different Java APIs can be used: jblas, Apache Commons Math, and ND4J.

OpenCL is an API that supports parallel operations on a variety of hardware platforms, processor types, and languages. This support is fairly low level. There are a number of Java bindings for OpenCL, which we reviewed.

Aparapi is a higher level of support for Java that can use CPUs, CUDA, or OpenCL to effect parallel operations. We demonstrated this support using the matrix multiplication example.

We wrapped up our discussion with an introduction to Java 8 streams and lambda expressions. These language elements can support parallel operations to improve an application's performance. In addition, this can often provide a more elegant and more maintainable implementation once the programmer becomes familiar with the techniques. We also demonstrated techniques for performing map-reduce using Java 8.

In the next chapter, we will conclude the book by illustrating how many of the techniques introduced can be used to build a complete application.

12
Bringing It All Together

While we have demonstrated many aspects of using Java to support data science tasks, the need to combine and use these techniques in an integrated manner exists. It is one thing to use the techniques in isolation and another to use them in a cohesive fashion. In this chapter, we will provide you with additional experience with these technologies and insights into how they can be used together.

Specifically, we will create a console-based application that analyzes tweets related to a user-defined topic. Using a console-based application allows us to focus on data-science-specific technologies and avoids having to choose a specific GUI technology that may not be relevant to us. It provides a common base from which a GUI implementation can be created if needed.

The application performs and illustrates the following high-level tasks:

- Data acquisition
- Data cleaning, including:
    - Removing stop words
    - Cleaning the text
- Sentiment analysis
- Basic data statistic collection
- Display of results

More than one type of analysis can be used with many of these steps. We will show the more relevant approaches and allude to other possibilities as appropriate. We will use Java 8's features whenever possible.

Defining the purpose and scope of our application

The application will prompt the user for a set of selection criteria, which include topic and sub-topic areas, and the number of tweets to process. The analysis performed will simply compute and display the number of positive and negative tweets for a topic and sub-topic. We used a generic sentiment analysis model, which will affect the quality of the sentiment analysis. However, other models and more analysis can be added.

We will use a Java 8 stream to structure the processing of tweet data. It is a stream of `TweetHandler` objects, as we will describe shortly.

We use several classes in this application. They are summarized here:

- `TweetHandler`: This class holds the raw tweet text and specific fields needed for the processing including the actual tweet, username, and similar attributes.
- `TwitterStream`: This is used to acquire the application's data. Using a specific class separates the acquisition of the data from its processing. The class possesses a few fields that control how the data is acquired.
- `ApplicationDriver`: This contains the `main` method, user prompts, and the `TweetHandler` stream that controls the analysis.

Each of these classes will be detailed in later sections. However, we will present `ApplicationDriver` next to provide an overview of the analysis process and how the user interacts with the application.

Understanding the application's architecture

Every application has its own unique structure, or architecture. This architecture provides the overarching organization or framework for the application. For this application, we combine the three classes using a Java 8 stream in the `ApplicationDriver` class. This class consists of three methods:

- `ApplicationDriver`: Contains the applications' user input
- `performAnalysis`: Performs the analysis
- `main`: Creates the `ApplicationDriver` instance

The class structure is shown next. The three instance variables are used to control the processing:

```
public class ApplicationDriver {
    private String topic;
    private String subTopic;
    private int numberOfTweets;

    public ApplicationDriver() { ... }
    public void performAnalysis() { ...     }

    public static void main(String[] args) {
        new ApplicationDriver();
    }
}
```

The `ApplicationDriver` constructor follows. A `Scanner` instance is created and the sentiment analysis model is built:

```
public ApplicationDriver() {
    Scanner scanner = new Scanner(System.in);
    TweetHandler swt = new TweetHandler();
    swt.buildSentimentAnalysisModel();
    ...
}
```

The remainder of the method prompts the user for input and then calls the `performAnalysis` method:

```
out.println("Welcome to the Tweet Analysis Application");
out.print("Enter a topic: ");
this.topic = scanner.nextLine();
out.print("Enter a sub-topic: ");
this.subTopic = scanner.nextLine().toLowerCase();
out.print("Enter number of tweets: ");
this.numberOfTweets = scanner.nextInt();
performAnalysis();
```

The `performAnalysis` method uses a Java 8 `Stream` instance obtained from the `TwitterStream` instance. The `TwitterStream` class constructor uses the number of tweets and `topic` as input. This class is discussed in the *Data acquisition using Twitter* section:

```
public void performAnalysis() {
Stream<TweetHandler> stream = new TwitterStream(
    this.numberOfTweets, this.topic).stream();
    ...
}
```

The stream uses a series of `map`, `filter`, and a `forEach` method to perform the processing. The `map` method modifies the stream's elements. The `filter` methods remove elements from the stream. The `forEach` method will terminate the stream and generate the output.

The individual methods of the stream are executed in order. When acquired from a public Twitter stream, the Twitter information arrives as a JSON document, which we process first. This allows us to extract relevant tweet information and set the data to fields of the `TweetHandler` instance. Next, the text of the tweet is converted to lowercase. Only English tweets are processed and only those tweets that contain the sub-topic will be processed. The tweet is then processed. The last step computes the statistics:

```
stream
        .map(s -> s.processJSON())
        .map(s -> s.toLowerCase())
        .filter(s -> s.isEnglish())
        .map(s -> s.removeStopWords())
        .filter(s -> s.containsCharacter(this.subTopic))
        .map(s -> s.performSentimentAnalysis())
        .forEach((TweetHandler s) -> {
            s.computeStats();
            out.println(s);
        });
```

The results of the processing are then displayed:

```
out.println();
out.println("Positive Reviews: "
        + TweetHandler.getNumberOfPositiveReviews());
out.println("Negative Reviews: "
        + TweetHandler.getNumberOfNegativeReviews());
```

We tested our application on a Monday night during a Monday-night football game and used the topic #MNF. The # symbol is called a **hashtag** and is used to categorize tweets. By selecting a popular category of tweets, we ensured that we would have plenty of Twitter data to work with. For simplicity, we chose the football subtopic. We also chose to only analyze 50 tweets for this example. The following is an abbreviated sample of our prompts, input, and output:

```
Building Sentiment Model
Welcome to the Tweet Analysis Application
Enter a topic: #MNF
Enter a sub-topic: football
Enter number of tweets: 50
Creating Twitter Stream
51 messages processed!
Text: rt @ bleacherreport : touchdown , broncos ! c . j . anderson punches
```

```
! lead , 7 - 6 # mnf # denvshou
Date: Mon Oct 24 20:28:20 CDT 2016
Category: neg
...
Text: i cannot emphasize enough how big td drive . @ broncos offense .
needed confidence booster & amp ; just got . # mnf # denvshou
Date: Mon Oct 24 20:28:52 CDT 2016
Category: pos
Text: least touchdown game . # mnf
Date: Mon Oct 24 20:28:52 CDT 2016
Category: neg
Positive Reviews: 13
Negative Reviews: 27
```

We print out the text of each tweet, along with a timestamp and category. Notice that the text of the tweet does not always make sense. This may be due to the abbreviated nature of Twitter data, but it is partially due to the fact this text has been cleaned and stop words have been removed. We should still see our topic, `#MNF`, although it will be lowercase due to our text cleaning. At the end, we print out the total number of tweets classified as positive and negative.

The classification of tweets is done by the `performSentimentAnalysis` method. Notice the process of classification using sentiment analysis is not always precise. The following tweet mentions a touchdown by a Denver Broncos player. This tweet could be construed as positive or negative depending on an individual's personal feelings about that team, but our model classified it as positive:

```
Text: cj anderson td run @ broncos . broncos now lead 7 - 6 . # mnf
Date: Mon Oct 24 20:28:42 CDT 2016
Category: pos
```

Additionally, some tweets may have a neutral tone, such as the one shown next, but still be classified as either positive or negative. The following tweet is a retweet of a popular sports news twitter handle, `@bleacherreport`:

```
Text: rt @ bleacherreport : touchdown , broncos ! c . j . anderson punches
! lead , 7 - 6 # mnf # denvshou
Date: Mon Oct 24 20:28:37 CDT 2016
Category: neg
```

This tweet has been classified as negative but perhaps could be considered neutral. The contents of the tweet simply provide information about a score in a football game. Whether this is a positive or negative event will depend upon which team a person may be rooting for. When we examine the entire set of tweet data analysed, we notice that this same `@bleacherreport` tweet has been retweeted a number of times and classified as negative each time. This could skew our analysis when we consider that we may have a large number of improperly classified tweets. Using incorrect data decreases the accuracy of the results.

One option, depending on the purpose of analysis, may be to exclude tweets by news outlets or other popular Twitter users. Additionally we could exclude tweets with *RT*, an abbreviation denoting that the tweet is a retweet of another user.

There are additional issues to consider when performing this type of analysis, including the sub-topic used. If we were to analyze the popularity of a Star Wars character, then we would need to be careful which names we use. For example, when choosing a character name such as Han Solo, the tweet may use an alias. Aliases for Han Solo include Vykk Draygo, Rysto, Jenos Idanian, Solo Jaxal, Master Marksman, and Jobekk Jonn, to mention a few (`http://starwars.wikia.com/wiki/Category:Han_Solo_aliases`). The actor's name may be used instead of the actual character, which is Harrison Ford in the case of Han Solo. We may also want to consider the actor's nickname, such as Harry for Harrison.

Data acquisition using Twitter

The Twitter API is used in conjunction with HBC's HTTP client to acquire tweets, as previously illustrated in the Handling Twitter section of `Chapter 2`, *Data Acquisition*. This process involves using the public stream API at the default access level to pull a sample of public tweets currently streaming on Twitter. We will refine the data based on user-selected keywords.

To begin, we declare the `TwitterStream` class. It consists of two instance variables, (`numberOfTweets` and `topic`), two constructors, and a `stream` method. The `numberOfTweets` variable contains the number of tweets to select and process, and `topic` allows the user to search for tweets related to a specific topic. We have set our default constructor to pull `100` tweets related to `Star Wars`:

```
public class TwitterStream {
    private int numberOfTweets;
    private String topic;

    public TwitterStream() {
        this(100, "Stars Wars");
```

```
    }

    public TwitterStream(int numberOfTweets, String topic) { ... }

}
```

The heart of our `TwitterStream` class is the `stream` method. We start by performing authentication using the information provided by Twitter when we created our Twitter application. We then create a `BlockingQueue` object to hold our streaming data. In this example, we will set a default capacity of `1000`. We use our `topic` variable in the `trackTerms` method to specify the types of tweets we are searching for. Finally, we specify our `endpoint` and turn off stall warnings:

```
String myKey = "mySecretKey";
String mySecret = "mySecret";
String myToken = "myToKen";
String myAccess = "myAccess";

out.println("Creating Twitter Stream");
BlockingQueue<String> statusQueue = new
LinkedBlockingQueue<>(1000);
StatusesFilterEndpoint endpoint = new StatusesFilterEndpoint();
endpoint.trackTerms(Lists.newArrayList("twitterapi", this.topic));
endpoint.stallWarnings(false);
```

Now we can create an `Authentication` object using `OAuth1`, a variation of the `OAuth` class. This allows us to build our connection client and complete the HTTP connection:

```
Authentication twitterAuth = new OAuth1(myKey, mySecret, myToken,
  myAccess);

BasicClient twitterClient = new ClientBuilder()
        .name("Twitter client")
        .hosts(Constants.STREAM_HOST)
        .endpoint(endpoint)
        .authentication(twitterAuth)
        .processor(new StringDelimitedProcessor(statusQueue))
        .build();

twitterClient.connect();
```

Next, we create two ArrayLists, `list` to hold our `TweetHandler` objects and `twitterList` to hold the JSON data streamed from Twitter. We will discuss the `TweetHandler` object in the next section. We use the `drainTo` method in place of the `poll` method demonstrated in `Chapter 2`, *Data Acquisition*, because it can be more efficient for large amounts of data:

```
List<TweetHandler> list = new ArrayList();
List<String> twitterList = new ArrayList();
```

Next we loop through our retrieved messages. We call the `take` method to remove each string message from the `BlockingQueue` instance. We then create a new `TweetHandler` object using the message and place it in our `list`. After we have handled all of our messages and the for loop completes, we stop the HTTP client, display the number of messages, and return our stream of `TweetHandler` objects:

```
statusQueue.drainTo(twitterList);
for(int i=0; i<numberOfTweets; i++) {
    String message;
    try {
        message = statusQueue.take();
        list.add(new TweetHandler(message));
    } catch (InterruptedException ex) {
        ex.printStackTrace();
    }
}
twitterClient.stop();
out.printf("%d messages processed!\n",
    twitterClient.getStatsTracker().getNumMessages());

return list.stream();
}
```

We are now ready to clean and analyze our data.

Understanding the TweetHandler class

The `TweetHandler` class holds information about a specific tweet. It takes the raw JSON tweet and extracts those parts that are relevant to the application's needs. It also possesses the methods to process the tweet's text such as converting the text to lowercase and removing tweets that are not relevant. The first part of the class is shown next:

```
public class TweetHandler {
    private String jsonText;
    private String text;
    private Date date;
```

```
    private String language;
    private String category;
    private String userName;
    ...
    public TweetHandler processJSON() { ... }
    public TweetHandler toLowerCase(){ ... }
    public TweetHandler removeStopWords(){ ... }
    public boolean isEnglish(){ ... }
    public boolean containsCharacter(String character) { ... }
    public void computeStats(){ ... }
    public void buildSentimentAnalysisModel{ ... }
    public TweetHandler performSentimentAnalysis(){ ... }
}
```

The instance variables show the type of data retrieved from a tweet and processed, as detailed here:

- `jsonText`: The raw JSON text
- `text`: The text of the processed tweet
- `date`: The date of the tweet
- `language`: The language of the tweet
- `category`: The tweet classification, which is positive or negative
- `userName`: The name of the Twitter user

There are several other instance variables used by the class. The following are used to create and use a sentiment analysis model. The classifier static variable refers to the model:

```
private static String[] labels = {"neg", "pos"};
private static int nGramSize = 8;
private static DynamicLMClassifier<NGramProcessLM>
    classifier = DynamicLMClassifier.createNGramProcess(
        labels, nGramSize);
```

The default constructor is used to provide an instance to build the sentiment model. The single argument constructor creates a `TweetHandler` object using the raw JSON text:

```
    public TweetHandler() {
        this.jsonText = "";
    }

    public TweetHandler(String jsonText) {
        this.jsonText = jsonText;
    }
```

The remainder of the methods are discussed in the following sections.

Extracting data for a sentiment analysis model

In `Chapter 9`, *Text Analysis*, we used DL4J to perform sentiment analysis. We will use LingPipe in this example as an alternative to our previous approach. Because we want to classify Twitter data, we chose a dataset with pre-classified tweets, available at `http://thinknook.com/wp-content/uploads/2012/09/Sentiment-Analysis-Dataset.zip`. We must complete a one-time process of extracting this data into a format we can use with our model before we continue with our application development.

This dataset exists in a large `.csv` file with one tweet and classification per line. The tweets are classified as either `0` (negative) or `1` (positive). The following is an example of one line of this data file:

```
95,0,Sentiment140, - Longest night ever.. ugh!
http://tumblr.com/xwp1yxhi6
```

The first element represents a unique ID number which is part of the original data set and which we will use for the filename. The second element is the classification, the third is a data set label (effectively ignored for the purposes of this project), and the last element is the actual tweet text. Before we can use this data with our LingPipe model, we must write each tweet into an individual file. To do this, we created three string variables. The `filename` variable will be assigned either `pos` or `neg` depending on each tweet's classification and will be used in the write operation. We also use the `file` variable to hold the name of the individual tweet file and the `text` variable to hold the individual tweet text. Next, we use the `readAllLines` method with the `Paths` class's `get` method to store our data in a `List` object. We need to specify the charset, `StandardCharsets.ISO_8859_1`, as well:

```
try {
    String filename;
    String file;
    String text;
    List<String> lines = Files.readAllLines(
Paths.get("\\path-to-file\\SentimentAnalysisDataset.csv"),
StandardCharsets.ISO_8859_1);
    ...

} catch (IOException ex) {
    // Handle exceptions
}
```

Now we can loop through our list and use the `split` method to store our `.csv` data in a string array. We convert the element at position `1` to an integer and determine whether it is a `1`. Tweets classified with a `1` are considered positive tweets and we set `filename` to `pos`. All other tweets set the `filename` to `neg`. We extract the output filename from the element at position `0` and the text from element `3`. We ignore the label in position `2` for the purposes of this project. Finally, we write out our data:

```
for(String s : lines) {
    String[] oneLine = s.split(",");
    if(Integer.parseInt(oneLine[1])==1) {
        filename = "pos";
    } else {
        filename = "neg";
    }
    file = oneLine[0]+".txt";
    text = oneLine[3];
    Files.write(Paths.get(
        path-to-file\\txt_sentoken"+filename+""+file),
        text.getBytes());
}
```

Notice that we created the `neg` and `pos` directories within the `txt_sentoken` directory. This location is important when we read the files to build our model.

Building the sentiment model

Now we are ready to build our model. We loop through the `labels` array, which contains `pos` and `neg`, and for each label we create a new `Classification` object. We then create a new file using this label and use the `listFiles` method to create an array of filenames. Next, we will traverse these filenames using a `for` loop:

```
public void buildSentimentAnalysisModel() {
    out.println("Building Sentiment Model");

    File trainingDir = new File("\\path to file\\txt_sentoken");
    for (int i = 0; i < labels.length; i++) {
        Classification classification =
            new Classification(labels[i]);
        File file = new File(trainingDir, labels[i]);
        File[] trainingFiles = file.listFiles();
        ...
    }
}
```

Within the `for` loop, we extract the tweet data and store it in our string, `review`. We then create a new `Classified` object using `review` and `classification`. Finally we can call the `handle` method to classify this particular text:

```
for (int j = 0; j < trainingFiles.length; j++) {
    try {
        String review = Files.readFromFile(trainingFiles[j],
            "ISO-8859-1");
        Classified<CharSequence> classified = new
            Classified<>(review, classification);
        classifier.handle(classified);
    } catch (IOException ex) {
        // Handle exceptions
    }
 }
```

For the dataset discussed in the previous section, this process may take a substantial amount of time. However, we consider this time trade-off to be worth the quality of analysis made possible by this training data.

Processing the JSON input

The Twitter data is retrieved using JSON format. We will use Twitter4J (`http://twitter4j.org`) to extract the relevant parts of the tweet and store in the corresponding field of the `TweetHandler` class.

The `TweetHandler` class's `processJSON` method does the actual data extraction. An instance of the `JSONObject` is created based on the JSON text. The class possesses several methods to extract specific types of data from an object. We use the `getString` method to get the fields we need.

The start of the `processJSON` method is shown next, where we start by obtaining the `JSONObject` instance, which we will use to extract the relevant parts of the tweet:

```
public TweetHandler processJSON() {
    try {
        JSONObject jsonObject = new JSONObject(this.jsonText);
        ...
    } catch (JSONException ex) {
        // Handle exceptions
    }
    return this;
}
```

First, we extract the tweet's text as shown here:

```
this.text = jsonObject.getString("text");
```

Next, we extract the tweet's date. We use the `SimpleDateFormat` class to convert the date string to a `Date` object. Its constructor is passed a string that specifies the format of the date string. We used the string `"EEE MMM d HH:mm:ss Z yyyy"`, whose parts are detailed next. The order of the string elements corresponds to the order found in the JSON entity:

- `EEE`: Day of the week specified using three characters
- `MMM`: Month, using three characters
- `d`: Day of the month
- `HH:mm:ss`: Hours, minutes, and seconds
- `Z`: Time zone
- `yyyy`: Year

The code follows:

```
SimpleDateFormat sdf = new SimpleDateFormat(
    "EEE MMM d HH:mm:ss Z yyyy");
try {
    this.date = sdf.parse(jsonObject.getString("created_at"));
} catch (ParseException ex) {
    // Handle exceptions
}
```

The remaining fields are extracted as shown next. We had to extract an intermediate JSON object to extract the `name` field:

```
this.language = jsonObject.getString("lang");
JSONObject user = jsonObject.getJSONObject("user");
this.userName = user.getString("name");
```

Having acquired and extracted the text, we are now ready to perform the important task of cleaning the data.

Cleaning data to improve our results

Data cleaning is a critical step in most data science problems. Data that is not properly cleaned may have errors such as misspellings, inconsistent representation of elements such as dates, and extraneous words.

There are numerous data cleaning options that we can apply to Twitter data. For this application, we perform simple cleaning. In addition, we will filter out certain tweets.

The conversion of the text to lowercase letters is easily achieved as shown here:

```
public TweetHandler toLowerCase() {
    this.text = this.text.toLowerCase().trim();
    return this;
}
```

Part of the process is to remove certain tweets that are not needed. For example, the following code illustrates how to detect whether the tweet is in English and whether it contains a sub-topic of interest to the user. The `boolean` return value is used by the `filter` method in the Java 8 stream, which performs the actual removal:

```
public boolean isEnglish() {
    return this.language.equalsIgnoreCase("en");
}

public boolean containsCharacter(String character) {
    return this.text.contains(character);
}
```

Numerous other cleaning operations can be easily added to the process such as removing leading and trailing white space, replacing tabs, and validating dates and email addresses.

Removing stop words

Stop words are those words that do not contribute to the understanding or processing of data. Typical stop words include the 0, and, a, and or. When they do not contribute to the data process, they can be removed to simplify processing and make it more efficient.

There are several techniques for removing stop words, as discussed in `Chapter 9`, *Text Analysis*. For this application, we will use LingPipe (`http://alias-i.com/lingpipe/`) to remove stop words. We use the `EnglishStopTokenizerFactory` class to obtain a model for our stop words based on an `IndoEuropeanTokenizerFactory` instance:

```
public TweetHandler removeStopWords() {
    TokenizerFactory tokenizerFactory
            = IndoEuropeanTokenizerFactory.INSTANCE;
    tokenizerFactory =
        new EnglishStopTokenizerFactory(tokenizerFactory);
    ...
    return this;
}
```

A series of tokens that do not contain stop words are extracted, and a `StringBuilder` instance is used to create a string to replace the original text:

```
Tokenizer tokens = tokenizerFactory.tokenizer(
        this.text.toCharArray(), 0, this.text.length());
StringBuilder buffer = new StringBuilder();
for (String word : tokens) {
    buffer.append(word + " ");
}
this.text = buffer.toString();
```

The LingPipe model we used may not be the best suited for all tweets. In addition, it has been suggested that removing stop words from tweets may not be productive (`http://oro.open.ac.uk/40666/`). Options to select various stop words and whether stop words should even be removed can be added to the stream process.

Performing sentiment analysis

We can now perform sentiment analysis using the model built in the Building the sentiment model section of this chapter. We create a new `Classification` object by passing our cleaned text to the `classify` method. We then use the `bestCategory` method to classify our text as either positive or negative. Finally, we set `category` to the result and return the `TweetHandler` object:

```
public TweetHandler performSentimentAnalysis() {
    Classification classification =
        classifier.classify(this.text);
    String bestCategory = classification.bestCategory();
    this.category = bestCategory;
    return this;
}
```

We are now ready to analyze the results of our application.

Analysing the results

The analysis performed in this application is fairly simple. Once the tweets have been classified as either positive or negative, a total is computed. We used two static variables for this purpose:

```
    private static int numberOfPositiveReviews = 0;
    private static int numberOfNegativeReviews = 0;
```

The `computeStats` method is called from the Java 8 stream and increments the appropriate variable:

```
public void computeStats() {
    if(this.category.equalsIgnoreCase("pos")) {
        numberOfPositiveReviews++;
    } else {
        numberOfNegativeReviews++;
    }
}
```

Two `static` methods provide access to the number of reviews:

```
public static int getNumberOfPositiveReviews() {
    return numberOfPositiveReviews;
}

public static int getNumberOfNegativeReviews() {
    return numberOfNegativeReviews;
}
```

In addition, a simple `toString` method is provided to display basic tweet information:

```
public String toString() {
    return "\nText: " + this.text
            + "\nDate: " + this.date
            + "\nCategory: " + this.category;
}
```

More sophisticated analysis can be added as required. The intent of this application was to demonstrate a technique for combining the various data processing tasks.

Other optional enhancements

There are numerous improvements that can be made to the application. Many of these are user preferences and others relate to improving the results of the application. A GUI interface would be useful in many situations. Among the user options, we may want add support for:

- Displaying individual tweets
- Allowing null sub-topics
- Processing other tweet fields

- Providing list of topics or sub-topics the user can choose from
- Generating additional statistics and supporting charts

With regard to process result improvements, the following should be considered:

- Correct user entries for misspelling
- Remove spacing around punctuation
- Use alternate stop word removal techniques
- Use alternate sentiment analysis techniques

The details of many of these enhancements are dependent on the GUI interface used and the purpose and scope of the application.

Summary

The intent of this chapter was to illustrate how various data science tasks can be integrated into an application. We chose an application that processes tweets because it is a popular social medium and allows us to apply many of the techniques discussed in earlier chapters.

A simple console-based interface was used to avoid cluttering the discussion with specific but possibly irrelevant GUI details. The application prompted the user for a Twitter topic, a sub-topic, and the number of tweets to process. The analysis consisted of determining the sentiments of the tweets, with simple statistics regarding the positive or negative nature of the tweets.

The first step in the process was to build a sentiment model. We used LingPipe classes to build a model and perform the analysis. A Java 8 stream was used and supported a fluent style of programming where the individual processing steps could be easily added and removed.

Once the stream was created, the JSON raw text was processed and used to initialize a `TweetHandler` class. Instances of this class were subsequently modified, including converting the text to lowercase, removing non-English tweets, removing stop words, and selecting only those tweets that contain the sub-topic. Sentiment analysis was then performed, followed by the computation of the statistics.

Data science is a broad topic that utilizes a wide range of statistical and computer science topics. In this book, we provided a brief introduction to many of these topics and how they are supported by Java.

Module 2

Mastering Java for Data Science

Building Data Science Applications in Java

1
Data Science Using Java

This book is about building data science applications using the Java language. In this book, we will cover all the aspects of implementing projects from data preparation to model deployment.

The readers of this book are assumed to have some previous exposure to Java and data science, and the book will help to take this knowledge to the next level. This means learning how to effectively tackle a specific data science problem and get the most out of the available data.

This is an introductory chapter where we will prepare the foundation for all the other chapters. Here we will cover the following topics:

- What is machine learning and data science?
- **Cross Industry Standard Process for Data Mining** (**CRIPS-DM**), a methodology for doing data science projects
- Machine learning libraries in Java for medium and large-scale data science applications

By the end of this chapter, you will know how to approach a data science project and what Java libraries to use to do that.

Data science

Data science is the discipline of extracting actionable knowledge from data of various forms. The name **data science** emerged quite recently--it was invented by DJ Patil and Jeff Hammerbacher and popularized in the article *Data Scientist: The Sexiest Job of the 21st Century* in 2012. But the discipline itself had existed before for quite a while and previously was known by other names such as **data mining** or **predictive analytics**. Data science, like its predecessors, is built on statistics and machine learning algorithms for knowledge extraction and model building.

The **science** part of the term **data science** is no coincidence--if we look up **science**, its definition can be summarized to *systematic organization of knowledge in terms testable explanations and predictions*. This is exactly what data scientists do, by extracting patterns from available data, they can make predictions about future unseen data, and they make sure the predictions are validated beforehand.

Nowadays, data science is used across many fields, including (but not limited to):

- **Banking**: Risk management (for example, credit scoring), fraud detection, trading
- **Insurance**: Claims management (for example, accelerating claim approval), risk and losses estimation, also fraud detection
- **Health care**: Predicting diseases (such as strokes, diabetes, cancer) and relapses
- **Retail and e-commerce**: Market basket analysis (identifying product that go well together), recommendation engines, product categorization, and personalized searches

This book covers the following practical use cases:

- Predicting whether an URL is likely to appear on the first page of a search engine
- Predicting how fast an operation will be completed given the hardware specifications
- Ranking text documents for a search engine
- Checking whether there is a cat or a dog on a picture
- Recommending friends in a social network
- Processing large-scale textual data on a cluster of computers

In all these cases, we will use data science to learn from data and use the learned knowledge to solve a particular business problem.

We will also use a running example throughout the book, building a search engine. We will use it to illustrate many data science concepts such as, supervised machine learning, dimensionality reduction, text mining, and learning to rank models.

Machine learning

Machine learning is a part of computer science, and it is at the core of data science. The data itself, especially in big volumes, is hardly useful, but inside it hides highly valuable patterns. With the help of machine learning, we can recognize these hidden patterns, extract them, and then apply the learned information to the new unseen items.

For example, given the image of an animal, a machine learning algorithm can say whether the picture is a dog or a cat; or, given the history of a bank client, it will say how likely the client is to default, that is, to fail to pay the debt.

Often, machine learning models are seen as black boxes that take in a data point and output a prediction for it. In this book, we will look at what is inside these black boxes and see how and when it is best to use them.

The typical problems that machine learning solves can be categorized in the following groups:

- **Supervised learning**: For each data point, we have a *label*--extra information that describes the outcome that we want to learn. In the cats versus dogs case, the data point is an image of the animal; the label describes whether it's a dog or a cat.
- **Unsupervised learning**: We only have raw data points and no label information is available. For example, we have a collection of e-mails and we would like to group them based on how similar they are. There is no explicit label associated with the e-mails, which makes this problem unsupervised.
- **Semi-supervised learning**: Labels are given only for a part of the data.
- **Reinforcement learning**: Instead of labels, we have a *reward*; something the model gets by interacting with the *environment* it runs in. Based on the reward, it can adapt and maximize it. For example, a model that learns how to play chess gets a positive reward each time it eats a figure of the opponent, and gets a negative reward each time it loses a figure; and the reward is proportional to the value of the figure.

Supervised learning

As we discussed previously, for supervised learning we have some information attached to each data point, the label, and we can train a model to use it and to learn from it. For example, if we want to build a model that tells us whether there is a dog or a cat on a picture, then the picture is the data point and the information whether it is a dog or a cat is the label. Another example is predicting the price of a house--the description of a house is the data point, and the price is the label.

We can group the algorithms of supervised learning into classification and regression algorithms based on the nature of this information.

In **classification** problems, the labels come from some fixed finite set of classes, such as {cat, dog}, {default, not default}, or {office, food, entertainment, home}. Depending on the number of classes, the classification problem can be **binary** (only two possible classes) or **multi-class** (several classes).

Examples of classification algorithms are Naive Bayes, logistic regression, perceptron, **Support Vector Machine** (**SVM**), and many others. We will discuss classification algorithms in more detail in the first part of `Chapter 4`, *Supervised Learning - Classification and Regression*.

In **regression** problems, the labels are real numbers. For example, a person can have a salary in the range from $0 per year to several billions per year. Hence, predicting the salary is a regression problem.

Examples of regression algorithms are linear regression, LASSO, **Support Vector Regression** (**SVR**), and others. These algorithms will be described in more detail in the second part of `Chapter 4`, *Supervised Learning - Classification and Regression.*

Some of the supervised learning methods are universal and can be applied to both classification and regression problems. For example, decision trees, random forest, and other tree-based methods can tackle both types. We will discuss one such algorithm, gradient boosting machines in `Chapter 7`, *Extreme Gradient Boosting.*

Neural networks can also deal with both classification and regression problems, and we will talk about them in `Chapter 8`, *Deep Learning with DeepLearning4J*.

Unsupervised learning

Unsupervised learning covers the cases where we have no labels available, but still want to find some patterns hidden in the data. There are several types of unsupervised learning, and we will look into cluster analysis, or clustering and unsupervised dimensionality reduction.

Clustering

Typically, when people talk about unsupervised learning, they talk about **cluster analysis** or **clustering**. A cluster analysis algorithm takes a set of data points and tries to categorize them into groups such that similar items belong to the same group, and different items do not. There are many ways where it can be used, for example, in customer segmentation or text categorization.

Customer segmentation is an example of clustering. Given some description of customers, we try to put them into groups such that the customers in one group have similar profiles and behave in a similar way. This information can be used to understand what do the people in these groups want, and this can be used to target them with better advertisements and other promotional messages.

Another example is **text categorization**. Given a collection of texts, we would like to find common topics among these texts and arrange the texts according to these topics. For example, given a set of complaints in an e-commerce store, we may want to put ones that talk about similar things together, and this should help the users of the system navigate through the complaints easier.

Examples of cluster analysis algorithms are hierarchical clustering, k-means, **density-based spatial clustering of applications with noise** (**DBSCAN**), and many others. We will talk about clustering in detail in the first part of `Chapter 5`, *Unsupervised Learning - Clustering and Dimensionality Reduction*.

Dimensionality reduction

Another group of unsupervised learning algorithms is **dimensionality reduction** algorithms. This group of algorithms *compresses* the dataset, keeping only the most useful information. If our dataset has too much information, it can be hard for a machine learning algorithm to use all of it at the same time. It may just take too long for the algorithm to process all the data and we would like to compress the data, so processing it takes less time.

There are multiple algorithms that can reduce the dimensionality of the data, including **Principal Component Analysis** (**PCA**), Locally linear embedding, and t-SNE. All these algorithms are examples of unsupervised dimensionality reduction techniques.

Not all dimensionality reduction algorithms are unsupervised; some of them can use labels to reduce the dimensionality better. For example, many feature selection algorithms rely on labels to see what features are useful and what are not.

We will talk more about this in `Chapter 5`, *Unsupervised Learning - Clustering and Dimensionality Reduction.*

Natural Language Processing

Processing natural language texts is very complex, they are not very well structured and require a lot of cleaning and normalizing. Yet the amount of textual information around us is tremendous: a lot of text data is generated every minute, and it is very hard to retrieve useful information from them. Using data science and machine learning is very helpful for text problems as well; they allow us to find the right text, process it, and extract the valuable bits of information.

There are multiple ways we can use the text information. One example is information retrieval, or, simply, text search--given a user query and a collection of documents, we want to find what are the most relevant documents in the corpus with respect to the query, and present them to the user. Other applications include sentiment analysis--predicting whether a product review is positive, neutral or negative, or grouping the reviews according to how they talk about the products.

We will talk more about information retrieval, **Natural Language Processing** (**NLP**) and working with texts in `Chapter 6`, *Working with Text - Natural Language Processing and Information Retrieval*. Additionally, we will see how to process large amounts of text data in `Chapter 9`, *Scaling Data Science*.

The methods we can use for machine learning and data science are very important. What is equally important is the the way we create them and then put them to use in production systems. Data science process models help us make it more organized and systematic, which is why we will talk about them next.

Data science process models

Applying data science is much more than just selecting a suitable machine learning algorithm and using it on the data. It is always good to keep in mind that machine learning is only a small part of the project; there are other parts such as understanding the problem, collecting the data, testing the solution and deploying to the production.

When working on any project, not just data science ones, it is beneficial to break it down into smaller manageable pieces and complete them one-by-one. For data science, there are best practices that describe how to do it the best way, and they are called **process models**. There are multiple models, including CRISP-DM and OSEMN.

In this chapter, CRISP-DM is explained as **Obtain, Scrub, Explore, Model, and iNterpret** (**OSEMN**), which is more suitable for data analysis tasks and addresses many important steps to a lesser extent.

CRISP-DM

Cross Industry Standard Process for Data Mining (**CRISP-DM**) is a process methodology for developing data mining applications. It was created before the term *data science* became popular, it's reliable and time-tested by several generations of analytics. These practices are still useful nowadays and describe the high-level steps of any analytical project quite well.

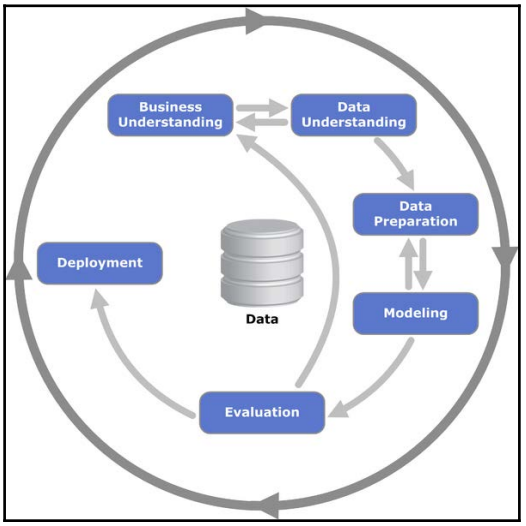


Image source: `https://en.wikipedia.org/wiki/File:CRISP-DM_Process_Diagram.png`

The CRISP-DM methodology breaks down a project into the following steps:

- Business understanding
- Data understanding
- Data preparation
- Modeling
- Evaluation
- Deployment

The methodology itself defines much more than just these steps, but typically knowing what the steps are and what happens at each step is enough for a successful data science project. Let's look at each of these steps separately.

The first step is **Business Understanding**. This step aims at learning what kinds of problems the business has and what they want to achieve by solving these problems. To be successful, a data science application must be useful for the business. The result of this step is the formulation of a problem which we want to solve and what is the desired outcome of the project.

The second step is **Data Understanding**. In this step, we try to find out what data can be used to solve the problem. We also need to find out if we already have the data; if not, we need to think how we can we get it. Depending on what data we find (or do not find), we may want to alter the original goal.

When the data is collected, we need to explore it. The process of reviewing the data is often called **Exploratory Data Analysis** and it is an integral part of any data science project. it helps to understand the processes that created the data, and can already suggest approaches for tackling the problem. The result of this step is the knowledge about which data sources are needed to solve the problem. We will talk more about this step in `Chapter 3`, *Exploratory Data Analysis*.

The third step of CRISP-DM is **Data Preparation**. For a dataset to be useful, it needs to be cleaned and transformed to a tabular form. The tabular form means that each row corresponds to exactly one observation. If our data is not in this shape, it cannot be used by most of the machine learning algorithms. Thus, we need to prepare the data such that it eventually can be converted to a matrix form and fed to a model.

Also, there could be different datasets that contain the needed information, and they may not be homogenous. What this means is that we need to convert these datasets to some common format, which can be read by the model.

This step also includes **Feature Engineering**--the process of creating features that are most informative for the problem and describe the data in the best way.

Many data scientists say that they spend most of their time on this step when building Data Science applications. We will talk about this step in `Chapter 2`, *Data Processing Toolbox* and throughout the book.

The fourth step is **Modeling**. In this step, the data is already in the right shape and we feed it to different Machine Learning algorithms. This step also includes parameter tuning, feature selection, and selecting the best model.

Evaluation of the quality of the models from the machine learning point of view happens during this step. The most important thing to check is the ability to generalize, and this is typically done via cross validation. In this step, we also may want to go back to the previous step and do extra cleaning and feature engineering. The outcome is a model that is potentially useful for solving the problem defined in Step 1.

The fifth step is **Evaluation**. It includes evaluating the model from the business perspective--not from the machine learning perspective. This means that we need to perform a critical review of the results so far and plan the next steps. Does the model achieve what we want? Additionally, some of the findings may lead to reconsidering the initial question. After this step, we can go to the deployment step or re-iterate the process.

The, final, sixth step is **Model Deployment**. During this step, the produced model is added to the production, so the result is the model integrated to the live system. We will cover this step in `Chapter 10`, *Deploying Data Science Models*.

Often, evaluation is hard because it is not always possible to say whether the model achieves the desired result or not. In these cases, the evaluation and deployment steps can be combined into one, the model is deployed and applied only to a part of users, and then the data for evaluating the model is collected. We will also briefly cover the ways of doing them, such as A/B testing and multi-armed bandits, in the last chapter of the book.

A running example

There will be many practical use cases throughout the book, sometimes a couple in each chapter. But we will also have a running example, building a search engine. This problem is interesting for a number of reasons:

- It is fun
- Business in almost any domain can benefit from a search engine

- Many businesses already have text data; often it is not used effectively, and its use can be improved
- Processing text requires a lot of effort, and it is useful to learn to do this effectively

We will try to keep it simple, yet, with this example, we will touch on all the technical parts of the data science process throughout the book:

- **Data Understanding**: Which data can be useful for the problem? How can we obtain this data?
- **Data Preparation**: Once the data is obtained, how can we process it? If it is HTML, how do we extract text from it? How do we extract individual sentences and words from the text?
- **Modeling**: Ranking documents by their relevance with respect to a query is a data science problem and we will discuss how it can be approached.
- **Evaluation**: The search engine can be tested to see if it is useful for solving the business problem or not.
- **Deployment**: Finally, the engine can be deployed as a REST service or integrated directly to the live system.

We will obtain and prepare the data in `Chapter 2`, *Data Processing Toolbox*, understand the data in `Chapter 3`, *Exploratory Data Analysis*, build simple models and evaluate them in `Chapter 4`, *Supervised Machine Learning - Classification and Regression*, look at how to process text in `Chapter 6`, *Working with Text - Natural Language Processing and Information Retrieval*, see how to apply it to millions of webpages in `Chapter 9`, *Scaling Data Science*, and, finally, learn how we can deploy it in `Chapter 10`, *Deploying Data Science Models*.

Data science in Java

In this book, we will use Java for doing data science projects. Java might not seem a good choice for data science at first glance, unlike Python or R, it has fewer data science and machine learning libraries, it is more verbose and lacks interactivity. On the other hand, it has a lot of upsides as follows:

- Java is a statically typed language, which makes it easier to maintain the code base and harder to make silly mistakes--the compiler can detect some of them.
- The standard library for data processing is very rich, and there are even richer external libraries.

- Java code is typically faster than the code in scripting languages that are usually used for data science (such as R or Python).
- Maven, the de-facto standard for dependency management in the Java world, makes it very easy to add new libraries to the project and avoid version conflicts.
- Most of big data frameworks for scalable data processing are written in either Java or JVM languages, such as Apache Hadoop, Apache Spark, or Apache Flink.
- Very often production systems are written in Java and building models in other languages adds unnecessary levels of complexity. Creating the models in Java makes it easier to integrate them to the product.

Next, we will look at the data science libraries available in Java.

Data science libraries

While there are not as many data science libraries in Java compared to R, there are quite a few. Additionally, it is often possible to use machine learning and data mining libraries written in other JVM languages, such as Scala, Groovy, or Clojure. Because these languages share the runtime environment, it makes it very easy to import libraries written in Scala and use them directly in Java code.

We can divide the libraries into the following categories:

- Data processing libraries
- Math and stats libraries
- Machine learning and data mining libraries
- Text processing libraries

Now we will see each of them in detail.

Data processing libraries

The standard Java library is very rich and offers a lot of tools for data processing, such as collections, I/O tools, data streams, and means of parallel task execution.

There are very powerful extensions to the standard library such as:

- Google Guava (`https://github.com/google/guava`) and Apache Common Collections (`https://commons.apache.org/collections/`) for richer collections

- Apache Commons IO (`https://commons.apache.org/io/`) for simplified I/O
- AOL Cyclops-React (`https://github.com/aol/cyclops-react`) for richer functional-way parallel streaming

We will cover both the standard API for data processing and its extensions in `Chapter 2`, *Data Processing Toolbox*. In this book, we will use Maven for including external libraries such as Google Guava or Apache Commons IO. It is a dependency management tool and allows to specify the external dependencies with a few lines of XML code. For example, to add Google Guava, it is enough to declare the following dependency in `pom.xml`:

```
<dependency>
 <groupId>com.google.guava</groupId>
 <artifactId>guava</artifactId>
 <version>19.0</version>
</dependency>
```

When we do it, Maven will go to the Maven Central repository and download the dependency of the specified version. The best way to find the dependency snippets for `pom.xml` (such as the previous one) is to use the search at `https://mvnrepository.com` or your favorite search engine.

Java gives an easy way to access databases through **Java Database Connectivity** (**JDBC**)--a unified database access protocol. JDBC makes it possible to connect virtually any relational database that supports SQL, such as MySQL, MS SQL, Oracle, PostgreSQL, and many others. This allows moving the data manipulation from Java to the database side.

When it is not possible to use a database for handling tabular data, then we can use DataFrame libraries for doing it directly in Java. The DataFrame is a data structure that originally comes from R and it allows to easily manipulate textual data in the program, without resorting to external database.

For example, with DataFrames it is possible to filter rows based on some condition, apply the same operation to each element of a column, group by some condition or join with another DataFrame. Additionally, some data frame libraries make it easy to convert tabular data to a matrix form so that the data can be used by machine learning algorithms.

There are a few data frame libraries available in Java. Some of them are as follows:

- Joinery (`https://cardillo.github.io/joinery/`)
- Tablesaw (`https://github.com/lwhite1/tablesaw`)
- Saddle (`https://saddle.github.io/`) a data frame library for Scala
- Apache Spark DataFrames (`http://spark.apache.org/`)

We will also cover databases and data frames in `Chapter 2`, *Data Processing Toolbox* and we will use DataFrames throughout the book.

There are more complex data processing libraries such as Spring Batch (`http://projects.spring.io/spring-batch/`). They allow creating complex data pipelines (called ETLs from Extract-Transform-Load) and manage their execution.

Additionally, there are libraries for distributed data processing such as:

- Apache Hadoop (`http://hadoop.apache.org/`)
- Apache Spark (`http://spark.apache.org/`)
- Apache Flink (`https://flink.apache.org/`)

We will talk about distributed data processing in `Chapter 9`, *Scaling Data Science*.

Math and stats libraries

The math support in the standard Java library is quite limited, and only includes methods such as `log` for computing the logarithm, `exp` for computing the exponent and other basic methods.

There are external libraries with richer support of mathematics. For example:

- Apache Commons Math (`http://commons.apache.org/math/`) for statistics, optimization, and linear algebra
- Apache Mahout (`http://mahout.apache.org/`) for linear algebra, also includes a module for distributed linear algebra and machine learning
- JBlas (`http://jblas.org/)` optimized and very fast linear algebra package that uses the BLAS library

Also, many machine learning libraries come with some extra math functionality, often linear algebra, stats, and optimization.

Machine learning and data mining libraries

There are quite a few machine learning and data mining libraries available for Java and other JVM languages. Some of them are as follows:

- Weka (`http://www.cs.waikato.ac.nz/ml/weka/`) is probably the most famous data mining library in Java, contains a lot of algorithms and has many extensions.

- JavaML (`http://java-ml.sourceforge.net/`) is quite an old and reliable ML library, but unfortunately not updated anymore
- Smile (`http://haifengl.github.io/smile/`) is a promising ML library that is under active development at the moment and a lot of new methods are being added there.
- JSAT (`https://github.com/EdwardRaff/JSAT`) contains quite an impressive list of machine learning algorithms.
- H2O (`http://www.h2o.ai/`) is a framework for distributed ML written in Java, but is available for multiple languages, including Scala, R, and Python.
- Apache Mahout (`http://mahout.apache.org/`) is used for in-core (one machine) and distributed machine learning. The Mahout Samsara framework allows writing the code in a framework-independent way and then executes it on Spark, Flink, or H2O.

There are several libraries that specialize solely on neural networks:

- Encog (`http://www.heatonresearch.com/encog/`)
- DeepLearning4j (`http://deeplearning4j.org/`)

We will cover some of these libraries throughout the book.

Text processing

It is possible to do simple text processing using only the standard Java library with classes such as `StringTokenizer`, the `java.text` package, or the regular expressions.

In addition to that, there is a big variety of text processing frameworks available for Java as follows:

- Apache Lucene (`https://lucene.apache.org/`) is a library that is used for information retrieval
- Stanford CoreNLP (`http://stanfordnlp.github.io/CoreNLP/`)
- Apache OpenNLP (`https://opennlp.apache.org/`)
- LingPipe (`http://alias-i.com/lingpipe/`)
- GATE (`https://gate.ac.uk/`)
- MALLET (`http://mallet.cs.umass.edu/`)
- Smile (`http://haifengl.github.io/smile/`) also has some algorithms for NLP

Most NLP libraries have very similar functionality and coverage of algorithms, which is why selecting which one to use is usually a matter of habit or taste. They all typically have tokenization, parsing, part-of-speech tagging, named entity recognition, and other algorithms for text processing. Some of them (such as StanfordNLP) support multiple languages, and some support only English.

We will cover some of these libraries in `Chapter 6`, *Working with Text - Natural Language Processing and Information Retrival*.

Summary

In this chapter, we briefly discussed data science and what role machine learning plays in it. Then we talked about doing a data science project, and what methodologies are useful for it. We discussed one of them, CRISP-DM, the steps it defines, how these steps are related and the outcome of each step.

Finally, we spoke about why doing a data science project in Java is a good idea, it is statically compiled, it's fast, and often the existing production systems already run in Java. We also mentioned libraries and frameworks one can use to successfully accomplish a data science project using the Java language.

With this foundation, we will now go to the most important (and most time-consuming) step in a data science project--Data Preparation.

2
Data Processing Toolbox

In the previous chapter, we discussed the best practices for approaching data science problems. We looked at CRISP-DM, which is the methodology for dealing with data mining projects, and one of the first steps there is data preprocessing. In this chapter, we will take a closer look at how to do this in Java.

Specifically, we will cover the following topics:

- Standard Java library
- Extensions to the standard library
- Reading data from different sources such as text, HTML, JSON, and databases
- DataFrames for manipulating tabular data

In the end, we will put everything together to prepare the data for the search engine.

By the end of this chapter, you will be able to process data such that it can be used for machine learning and further analysis.

Standard Java library

The standard Java library is very rich and offers a lot of tools for data manipulation, including:

- Collections for organizing data in memory
- I/O for reading and writing data
- Streaming APIs for making data transformations easy

In this chapter, we will look at all these tools in detail.

Collections

Data is the most important part of data science. When dealing with data, it needs to be efficiently stored and processed, and for this we use data structures. A data structure describes a way to store data efficiently to solve a specific problem, and the Java Collection API is the standard Java API for data structures. This API offers a wide variety of implementations that are useful in practical data science applications.

We will not describe the collection API in full detail, but concentrate on the most useful and important ones--list, set, and map interfaces.

Lists are collections where each element can be accessed by its index. The g0-to implementation of the `List` interface is `ArrayList`, which should be used in 99% of cases and it can be used as follows:

```
List<String> list = new ArrayList<>();
list.add("alpha");
list.add("beta");
list.add("beta");
list.add("gamma");
System.out.println(list);
```

There are other implementations of the `List` interface, `LinkedList` or `CopyOnWriteArrayList`, but they are rarely needed.

Set is another interface in the Collections API, and it describes a collection which allows no duplicates. The go-to implementation is `HashSet`, if the order in which we insert elements does not matter, or `LinkedHashSet`, if the order matters. We can use it as follows:

```
Set<String> set = new HashSet<>();
set.add("alpha");
set.add("beta");
set.add("beta");
set.add("gamma");
System.out.println(set);
```

`List` and `Set` both implement the `Iterable` interface, which makes it possible to use the `for-each` loop with them:

```
for (String el : set) {
    System.out.println(el);
}
```

The `Map` interface allows mapping keys to values, and is sometimes called as dictionary or associative array in other languages. The g0-to implementation is `HashMap`:

```
Map<String, String> map = new HashMap<>();
map.put("alpha", "α");
map.put("beta", "β");
map.put("gamma", "γ");
System.out.println(map);
```

If you need to keep the insertion order, you can use `LinkedHashMap`; if you know that the `map` interface will be accessed from multiple threads, use `ConcurrentHashMap`.

The `Collections` class provides several helper methods for dealing with collections such as sorting, or extracting the `max` or `min` elements:

```
String min = Collections.min(list);
String max = Collections.max(list);
System.out.println("min: " + min + ", max: " + max);
Collections.sort(list);
Collections.shuffle(list);
```

There are other collections such as `Queue`, `Deque`, `Stack`, thread-safe collections, and some others. They are less frequently used and not very important for data science.

Input/Output

Data scientists often work with files and other data sources. I/O is needed for reading from the data sources and writing the results back. The Java I/O API provides two main types of abstraction for this:

- `InputStream`, `OutputStream` for binary data
- `Reader`, `Writer` for text data

Typical data science applications deal with text rather than raw binary data--the data is often stored in TXT, CSV, JSON, and other similar text formats. This is why we will concentrate on the second part.

Reading input data

Being able to read data is the most important skill for a data scientist, and this data is usually in text format, be it TXT, CSV, or any other format. In Java I/O API, the subclasses of the `Reader` classes deal with reading text files.

Suppose we have a `text.txt` file with some sentences (which may or may not make sense):

- My dog also likes eating sausage
- The motor accepts beside a surplus
- Every capable slash succeeds with a worldwide blame
- The continued task coughs around the guilty kiss

If you need to read the whole file as a list of strings, the usual Java I/O way of doing this is using `BufferedReader`:

```
List<String> lines = new ArrayList<>();

try (InputStream is = new FileInputStream("data/text.txt")) {
    try (InputStreamReader isReader = new InputStreamReader(is,
                StandardCharsets.UTF_8)) {
        try (BufferedReader reader = new BufferedReader(isReader)) {
            while (true) {
                String line = reader.readLine();
                if (line == null) {
                    break;
                }
                lines.add(line);
            }

            isReader.close();
        }
    }
}
```



It is important to provide character encoding--this way, the `Reader` knows how to translate the sequence of bytes into a proper `String` object. Apart from UTF-8, there are UTF-16, ISO-8859 (which is ASCII-based text encoding for English), and many others.

There is a shortcut to get `BufferedReader` for a file directly:

```
Path path = Paths.get("data/text.txt");
try (BufferedReader reader = Files.newBufferedReader(path,
        StandardCharsets.UTF_8)) {
    // read line-by-line
}
```

Even with this shortcut, you can see that this is quite verbose for such a simple task as reading a list of lines from a file. You can wrap this in a helper function, or instead use the Java NIO API, which gives some helper methods to make this task easier:

```
Path path = Paths.get("data/text.txt");
List<String> lines = Files.readAllLines(path, StandardCharsets.UTF_8);
System.out.println(lines);
```



The Java NIO shortcuts work only for files. Later, we will talk about shortcuts that work for any InputStream objects, not just files.

Writing ouput data

After the data is read and processed, we often want to put it back on disk. For text, this is usually done using the `Writer` objects.

Suppose we read the sentences from `text.txt` and we need to convert each line to uppercase and write them back to a new file `output.txt`; the most convenient way of writing the text data is via the `PrintWriter` class:

```
try (PrintWriter writer = new PrintWriter("output.txt", "UTF-8")) {
    for (String line : lines) {
        String upperCase = line.toUpperCase(Locale.US);
        writer.println(upperCase);
    }
}
```

In Java NIO API, it would look like this:

```
Path output = Paths.get("output.txt");
try (BufferedWriter writer = Files.newBufferedWriter(output,
           StandardCharsets.UTF_8)) {
    for (String line : lines) {
        String upperCase = line.toUpperCase(Locale.US);
        writer.write(upperCase);
        writer.newLine();
    }
}
```

Both ways are correct and you should select the one that you prefer. However, it is important to remember to always include the encoding; otherwise, it may use some default values which are platform-dependent and sometimes arbitrary.

Streaming API

Java 8 was a big step forward in the history of the Java language. Among other features, there were two important things--Streams and Lambda expressions.

In Java, a stream is a sequence of objects, and the Streams API provides functional-style operations to transform these sequences, such as map, filter, and reduce. The sources for streams can be anything that contain elements, for example, arrays, collections, or files.

For example, let's create a simple `Word` class, which contains a token and its part of speech:

```
public class Word {
    private final String token;
    private final String pos;
    // constructor and getters are omitted
}
```

For brevity, we will always omit constructors and getters for such data classes, but indicate that with a comment.

Now, let's consider a sentence *My dog also likes eating sausage*. Using this class, we can represent it as follows:

```
Word[] array = { new Word("My", "RPR"), new Word("dog", "NN"),
     new Word("also", "RB"), new Word("likes", "VB"),
     new Word("eating", "VB"), new Word("sausage", "NN"),
     new Word(".", ".") };
```

Here, we use the Penn Treebank POS notation, where `NN` represents a noun or `VB` represents a verb.

Now, we can convert this array to a stream using the `Arrays.stream` utility method:

```
Stream<Word> stream = Arrays.stream(array);
```

Streams can be created from collections using the `stream` method:

```
List<Word> list = Arrays.asList(array);
Stream<Word> stream = list.stream();
```

The operations on streams are chained together and form nice and readable data processing pipelines. The most common operations on streams are the map and filter operations:

- Map applies the same transformer function to each element
- Filter, given a predicate function, filters out elements that do not satisfy it

At the end of the pipeline, you collect the results using a collector. The `Collectors` class provides several implementations such as `toList`, `toSet`, `toMap`, and others.

Suppose we want to keep only tokens which are nouns. With the Streams API, we can do it as follows:

```
List<String> nouns = list.stream()
        .filter(w -> "NN".equals(w.getPos()))
        .map(Word::getToken)
        .collect(Collectors.toList());
System.out.println(nouns);
```

Alternatively, we may want to check how many unique POS tags there are in the stream. For this, we can use the `toSet` collector:

```
Set<String> pos = list.stream()
        .map(Word::getPos)
        .collect(Collectors.toSet());
System.out.println(pos);
```

When dealing with texts, we may sometimes want to join a sequence of strings together:

```
String rawSentence = list.stream()
        .map(Word::getToken)
        .collect(Collectors.joining(" "));
System.out.println(rawSentence);
```

Alternatively, we can group words by their `POS` tag:

```
Map<String, List<Word>> groupByPos = list.stream()
        .collect(Collectors.groupingBy(Word::getPos));
System.out.println(groupByPos.get("VB"));
System.out.println(groupByPos.get("NN"));
```

Also, there is a useful `toMap` collector that can index a collection using some fields. For example, if we want to get a map from tokens to the `Word` objects, it can be achieved using the following code:

```
Map<String, Word> tokenToWord = list.stream()
        .collect(Collectors.toMap(Word::getToken, Function.identity()));
System.out.println(tokenToWord.get("sausage"));
```

Apart from object streams, the Streams API provides primitive streams--streams of ints, doubles, and other primitives. These streams have useful methods for statistical calculations such as `sum`, `max`, `min`, or `average`. A usual stream can be converted to a primitive stream using functions such as `mapToInt` or `mapToDouble`.

For example, this is how we can find the maximum length across all words in our sentence:

```
int maxTokenLength = list.stream()
        .mapToInt(w -> w.getToken().length())
        .max().getAsInt();
System.out.println(maxTokenLength);
```

Stream operations are easy to parallelize; they are applied to each item separately, and therefore multiple threads can do that without interfering with one another. So, it is possible to make these operations a lot faster by splitting the work across multiple processors and execute all the tasks in parallel.

Java leverages that and provides an easy and expressive way to create parallel code; for collections, you just need to call the `parallelStream` method:

```
int[] firstLengths = list.parallelStream()
        .filter(w -> w.getToken().length() % 2 == 0)
        .map(Word::getToken)
        .mapToInt(String::length)
        .sequential()
        .sorted()
        .limit(2)
        .toArray();
System.out.println(Arrays.toString(firstLengths));
```

In this example, the filtering and mapping is done in parallel, but then the stream is converted to a sequential stream, sorted, and the top two elements are extracted to an array. While the example is not very meaningful, it shows how much it is possible to do with streams.

Finally, the standard Java I/O library offers some convenience methods. For example, it is possible to represent a text file as a stream of lines using the `Files.lines` method:

```
Path path = Paths.get("text.txt");
try (Stream<String> lines = Files.lines(path, StandardCharsets.UTF_8)) {
    double average = lines
        .flatMap(line -> Arrays.stream(line.split(" ")))
        .map(String::toLowerCase)
        .mapToInt(String::length)
        .average().getAsDouble();
    System.out.println("average token length: " + average);
}
```

Streams are an expressive and powerful way to process data and mastering this API is very helpful for doing data science in Java. Later on, we will often use the Stream API, so you will see more examples of how to use it.

Extensions to the standard library

The standard Java library is quite powerful, but some things take a long time to write using it or they are simply missing. There are a number of extensions to the standard library, and the most prominent libraries are Apache Commons (a collection of libraries) and Google Guava. They make it easier to use the standard API or extend it, for example, by adding new collections or implementations.

To begin with, we will briefly go over the most relevant parts of these libraries, and later on we will see how they are used in practice.

Apache Commons

Apache Commons is a collection of open source libraries for Java, with the goal of creating reusable Java components. There are quite a few of them, including Apache Commons Lang, Apache Commons IO, Apache Commons Collections, and many others.

Commons Lang

Apache Commons Lang is a set of utility classes that extend the `java.util` package and they make the life of a Java developer a lot easier by providing many little methods that solve common problems and save a lot of time.

To include external libraries in Java, we usually use Maven, which makes it very easy to manage dependencies. With Maven, the Apache Commons Lang library can be included using this `dependency` snippet:

```
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-lang3</artifactId>
  <version>3.4</version>
</dependency>
```

The library contains a lot of methods useful for general-purpose Java programming, such as making it easier to implement the `equals` and `hashCode` methods for objects, serialization helpers and others. In general, they are not very specific to data science, but there are a few helper functions that are quite useful. For example,

- `RandomUtils` and `RandomStringUtils` for generating data
- `StringEscapeUtils` and `LookupTranslator` for escaping and un-escaping strings

- `EqualsBuilder` and `HashCodeBuilder` for the fast implementation of `equals` and `hashCode` methods
- `StringUtils` and `WordUtils` for useful string manipulation methods
- the `Pair` class

For more information, you can read the documentation at `https://commons.apache.org/lang`.



The best way to see what is available is to download the package and see the code available there. Every Java developer will find a lot of useful things.

Commons IO

Like Apache Commons Lang extends the `java.util` standard package, Apache Commons IO extends `java.io`; it is a Java library of utilities to assist with I/O in Java, which, as we previously learned, can be quite verbose.

To include the library in your project, add the `dependency` snippet to the `pom.xml` file:

```
<dependency>
  <groupId>commons-io</groupId>
  <artifactId>commons-io</artifactId>
  <version>2.5</version>
</dependency>
```

We already learned about `Files.lines` from Java NIO. While it is handy, we do not always work with files, and sometimes need to get lines from some other `InputStream`, for example, a web page or a web socket.

For this purpose, Commons IO provides a very helpful utility class `IOUtils`. Using it, reading the entire input stream into string or a list of strings is quite easy:

```
try (InputStream is = new FileInputStream("data/text.txt")) {
    String content = IOUtils.toString(is, StandardCharsets.UTF_8);
    System.out.println(content);
}

try (InputStream is = new FileInputStream("data/text.txt")) {
    List<String> lines = IOUtils.readLines(is, StandardCharsets.UTF_8);
    System.out.println(lines);
}
```

Although we use `FileInputStream` objects in this example, it can be any other stream. The first method, `IOUtils.toString`, is particularly useful, and we will use it later for crawling web pages and processing the responses from web services.

There are a lot more useful methods for I/O in this library, and to get a good overview, you can consult the documentation available at `https://commons.apache.org/io`.

Commons Collections

The Java Collections API is very powerful and it defines a good set of abstractions for data structures in Java. The Commons Collections use these abstractions and extend the standard Collections API with new implementations as well as new collections. To include the library, use this snippet:

```
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-collections4</artifactId>
  <version>4.1</version>
</dependency>
```

Some useful collections from this library are:

- `Bag`: This is an interface for sets that can hold the same element multiple times
- `BidiMap`: This stands for Bi-directional map. It can map from keys to values and from values to keys

It has some overlap with collections from Google Guava, explained in the next session, but has some additional collections that aren't implemented there. For example,

- `LRUMap`: This is used for implementing caches
- `PatriciaTrie`: This is used for fast string prefix lookups

Other commons modules

Commons Lang, IO, and Collections are a few commons libraries out of many. There are other commons modules that are useful for data science:

- Commons compress is used for reading compressed files such as, `bzip2` (for reading Wikipedia dumps), `gzip`, `7z`, and others

- Commons CSV is used for reading and writing CSV files (we will use it later)
- Commons math is used for statistical calculation and linear algebra (we will also use it later)

You can refer to `https://commons.apache.org/` for the whole list.

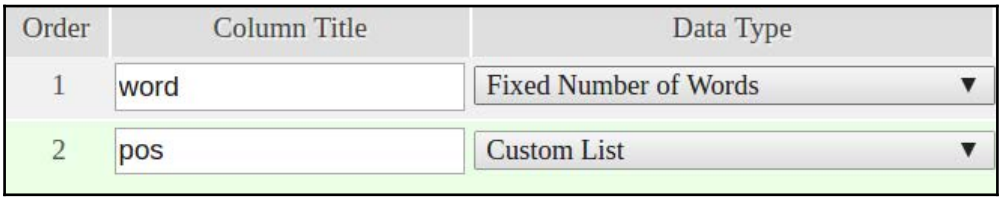
Google Guava

Google Guava is very similar to Apache Commons; it is a set of utilities that extend the standard Java API and make life easier. But unlike Apache Commons, Google Guava is one library that covers many areas at once, including collections and I/O.

To include it in a project, use `dependency`:

```
<dependency>
  <groupId>com.google.guava</groupId>
  <artifactId>guava</artifactId>
  <version>19.0</version>
</dependency>
```

We will start with the Guava I/O module. To give an illustration, we will use some generated data. We already used the `word` class, which contains a token and its part-of-speech tag, and here we will generate more words. To do that, we can use a data generation tool such as `http://www.generatedata.com/`. Let's define the following schema as shown in the following screenshot:



After that it is possible to save the generated data to the CSV format, set the delimiter to tab (`t`), and save it to `words.txt`. We already generated a file for you; you can find it in the `chapter2` repository.

Guava defines a few abstractions for working with I/O. One of them is `CharSource`, an abstraction for any source of character-based data, which, in some sense, is quite similar to the standard `Reader` class. Additionally, similarly to Commons IO, there is a utility class for working with files. It is called `Files` (not to be confused with `java.nio.file.Files`), and contains helper functions that make file I/O easier. Using this class, it is possible to read all lines of a text file as follows:

```
File file = new File("data/words.txt");
CharSource wordsSource = Files.asCharSource(file, StandardCharsets.UTF_8);
List<String> lines = wordsSource.readLines();
```

Google Guava Collections follows the same idea as Commons Collections; it builds on the Standard Collections API and provides new implementations and abstractions. There are a few utility classes such as `Lists`, for working with lists, `Sets` for working with sets, and so on.

One of the methods from `Lists` is `transform`, it is like `map` on streams and it is applied to every element from the list. The elements of the resulting list are evaluated lazily; the computation of the function is triggered only when the element is needed. Let's use it for transforming the lines from the text file to a list of `Word` objects:

```
List<Word> words = Lists.transform(lines, line -> {
    String[] split = line.split("t");
    return new Word(split[0].toLowerCase(), split[1]);
});
```

The main difference between this and map from the Streams API is that transform immediately returns a list, so there is no need to first create a stream, call the `map` function, and finally collect the results to list.

Similarly to Commons Collections, there are new collections that are not available in the Java API. The most useful collections for data science are `Multiset`, `Multimap`, and `Table`.

Multisets are sets where the same element can be stored multiple times, and they are usually used for counting things. This class is especially useful for text processing, when we want to calculate how many times each term appears.

Let's take the words that we read and calculate how many times each `pos` tag appeared:

```
Multiset<String> pos = HashMultiset.create();
for (Word word : words) {
    pos.add(word.getPos());
}
```

If we want to output the results sorted by counts, there is a special utility function for that:

```
Multiset<String> sortedPos = Multisets.copyHighestCountFirst(pos);
System.out.println(sortedPos);
```

Multimap is a map that for each key can have multiple values. There are several types of multimaps. The two most common maps are as follows:

- `ListMultimap`: This associates a key with a list of values, similar to `Map<Key, List<Value>>`
- `SetMultimap`: This associates a key to a set of values, similar to `Map<Key, Set<Value>>`

This can be quite useful for implementing `group by` logic. Let's look at the average length per `POS` tag:

```
ArrayListMultimap<String, String> wordsByPos = ArrayListMultimap.create();
for (Word word : words) {
    wordsByPos.put(word.getPos(), word.getToken());
}
```

It is possible to view a multimap as a map of collections:

```
Map<String, Collection<String>> wordsByPosMap = wordsByPos.asMap();
wordsByPosMap.entrySet().forEach(System.out::println);
```

Finally, the `Table` collection can be seen as a two-dimensional extension of the `map` interface; now, instead of one key, each entry is indexed by two keys, `row` keys and `column` keys. In addition to that, it is also possible to get the entire column using the `column` key or a row using the `row` key.

For example, we can count how many times each (word, POS) pair appeared in the dataset:

```
Table<String, String, Integer> table = HashBasedTable.create();
for (Word word : words) {
    Integer cnt = table.get(word.getPos(), word.getToken());
    if (cnt == null) {
        cnt = 0;
    }
    table.put(word.getPos(), word.getToken(), cnt + 1);
}
```

Once the data is put to the table, we can access the rows and columns individually:

```
Map<String, Integer> nouns = table.row("NN");
System.out.println(nouns);
```

```
String word = "eu";
Map<String, Integer> posTags = table.column(word);
System.out.println(posTags);
```

Like in Commons Lang, Guava also contains utility classes for working with primitives such as `Ints` for `int` primitives, `Doubles` for `double` primitives, and so on. For example, it can be used to convert a collection of primitive wrappers to a primitive array:

```
Collection<Integer> values = nouns.values();
int[] nounCounts = Ints.toArray(values);
int totalNounCount = Arrays.stream(nounCounts).sum();
System.out.println(totalNounCount);
```

Finally, Guava provides a nice abstraction for sorting data--`Ordering`, which extends the standard `Comparator` interface. It provides a clean fluent interface for creating comparators:

```
Ordering<Word> byTokenLength =
        Ordering.natural().<Word> onResultOf(w ->
w.getToken().length()).reverse();
List<Word> sortedByLength = byTokenLength.immutableSortedCopy(words);
System.out.println(sortedByLength);
```

Since `Ordering` implements the `Comparator` interface, it can be used wherever a comparator is expected. For example, for `Collections.sort`:

```
List<Word> sortedCopy = new ArrayList<>(words);
Collections.sort(sortedCopy, byTokenLength);
```

In addition to that, it provides other methods such as extracting the top-k or bottom-k elements:

```
List<Word> first10 = byTokenLength.leastOf(words, 10);
System.out.println(first10);
List<Word> last10 = byTokenLength.greatestOf(words, 10);
System.out.println(last10);
```

It is the same as first sorting and then taking the first or last k elements, but more efficient.

There are other useful classes:

- Customizable hash implementations such as Murmur hash and others
- `Stopwatch` for measuring time

For more insights, you can refer to `https://github.com/google/guava`and `https://github.com/google/guava/wiki`.

You may have noticed that Guava and Apache Commons have a lot in common. Selecting which one to use is a matter of taste--both libraries are very well tested and actively used in many production systems. However, Guava is more actively developed and new features appear more often, so if you want to use only one of them, then Guava may be a better choice.

AOL Cyclops React

As we already learned, Java Streams API is a very powerful way of dealing with data in a functional way. The Cyclops React library extends this API by adding new operations on streams and allows for more control of the flow execution. To include the library, add this to the `pom.xml` file:

```
<dependency>
  <groupId>com.aol.simplereact</groupId>
  <artifactId>cyclops-react</artifactId>
  <version>1.0.0-RC4</version>
</dependency>
```

Some of the methods it adds are `zipWithIndex` and cast and convenience collectors such as `toList`, `toSet`, and `toMap`. What is more, it gives more control for parallel execution, for example, it is possible to provide a custom executor, which will be used for processing data or intercepting exceptions declaratively.

Also, with this library, it is easy to create a parallel stream from the iterator--it is hard to do it with the standard library.

For example, let's take `words.txt`, extract all POS tags from it, and then create a map that associates each tag with a unique index. For reading data, we will use `LineIterator` from Commons IO, which otherwise would be hard to parallelize using only standard Java APIs. Additionally, we create a custom executor, which will be used for executing the stream operations in parallel:

```
LineIterator it = FileUtils.lineIterator(new File("data/words.txt"),
"UTF-8");
ExecutorService executor = Executors.newCachedThreadPool();
LazyFutureStream<String> stream =
        LazyReact.parallelBuilder().withExecutor(executor).from(it);

Map<String, Integer> map = stream
        .map(line -> line.split("t"))
        .map(arr -> arr[1].toLowerCase())
        .distinct()
        .zipWithIndex()
```

```
            .toMap(Tuple2::v1, t -> t.v2.intValue());

    System.out.println(map);
    executor.shutdown();
    it.close();
```

It is a very simple example and does not come close to describing all the functionality available in this library. For more information, refer to their documentation, which can be found at `https://github.com/aol/cyclops-react`. We will also use it in other examples in later chapters.

Accessing data

By now we already have spent a lot of time describing how to read and write data. But there is much more to that: data often comes in different formats such as CSV, HTML, or JSON or it can be stored in a database. Knowing how to access and process this data is important for Data Science and now we will describe in detail how to do it for the most common data formats and sources.

Text data and CSV

We already have spoken about reading text data in great detail, and it can be done, for example, using the `Files` helper class from the NIO API or `IOUtils` from Commons IO.

CSV (Comma Separated Values) is a common way to organize tabular data in plain text files. While it is possible to parse CSV files by hand, there are some corner cases, which make it a bit cumbersome. Luckily, there are nice libraries for that purpose, and one of them is Apache Commons CSV:

```
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-csv</artifactId>
  <version>1.4</version>
</dependency>
```

To illustrate how to use this library, let's generate some random data once again. This time we can also use `http://www.generatedata.com/` and define the following schema:



Now we can create a special class for holding this data:

```
public static class Person {
    private final String name;
    private final String email;
    private final String country;
    private final int salary;
    private final int experience;
    // constructor and getters are omitted
}
```

Then, to read a CSV file you can do the following:

```
List<Person> result = new ArrayList<>();

Path csvFile = Paths.get("data/csv-example-generatedata_com.csv");
try (BufferedReader reader = Files.newBufferedReader(csvFile,
StandardCharsets.UTF_8)) {
    CSVFormat csv = CSVFormat.RFC4180.withHeader();
    try (CSVParser parser = csv.parse(reader)) {
        Iterator<CSVRecord> it = parser.iterator();
        it.forEachRemaining(rec -> {
            String name = rec.get("name");
            String email = rec.get("email");
            String country = rec.get("country");
            int salary = Integer.parseInt(rec.get("salary").substring(1));
            int experience = Integer.parseInt(rec.get("experience"));
            Person person = new Person(name, email, country, salary,
experience);
            result.add(person);
        });
    }
}
```

The preceding code creates an iterator of `CSVRecord` objects, and from each such object we extract the values and pass them to a data object. Creating an iterator is useful when the CSV file is very large and may not fit entirely into available memory.

If the file is not too large, it is also possible to read the entire CSV at once and put the results into a list:

```
List<CSVRecord> records = parse.getRecords();
```

Finally, tab-separated files can be seen as a special case of CSV and also can be read using this library. To do it, you just need to use the `TDF` format for parsing:

```
CSVFormat tsv = CSVFormat.TDF.withHeader();
```

Web and HTML

There is a lot of data on the Internet nowadays, and being able to access this data and transform it into something machine-readable is a very important skill for a Data Scientist.

There are multiple ways of accessing data from the Internet. Luckily for us, the standard Java API provides a special class for doing that, `URL`. With `URL`, it is possible to open an `InputStream`, which will contain the response body. For web pages it typically is its HTML content. With `IOUtils` from Commons IO, doing this is simple:

```
try (InputStream is = new URL(url).openStream()) {
    return IOUtils.toString(is, StandardCharsets.UTF_8);
}
```

This piece of code is quite useful, so putting it into some helper method, for example, `UrlUtils.request`, will be helpful.

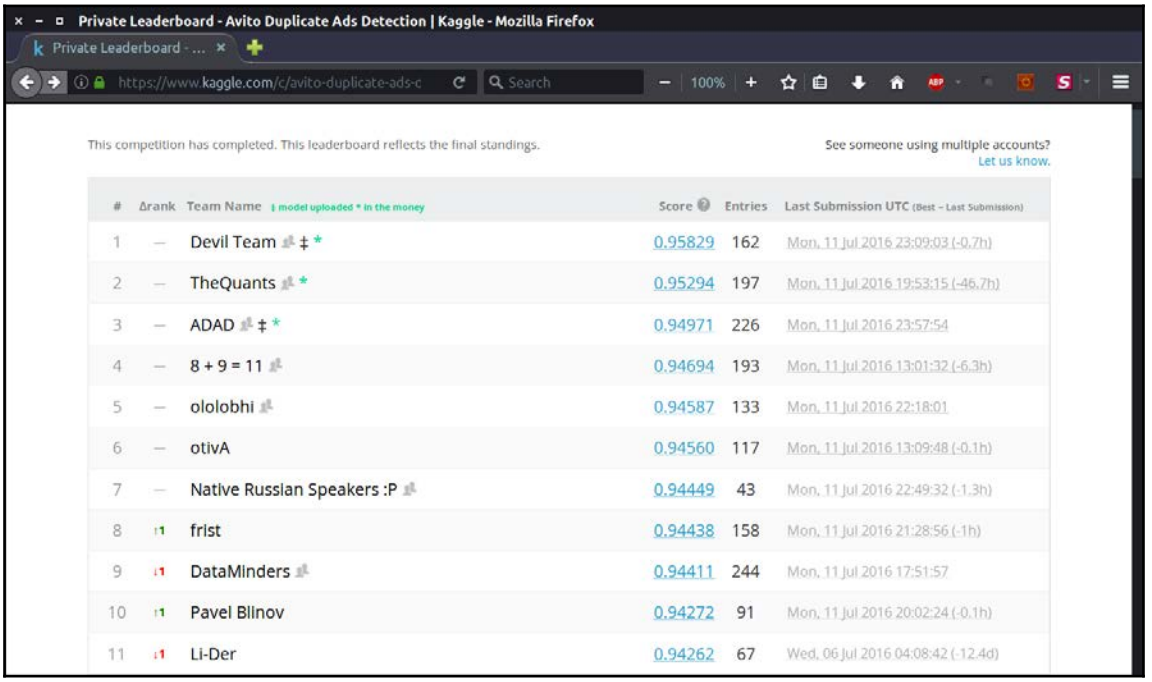


Here we assume that the content of a web page is always UTF-8. It may work for many cases, especially for pages in English, but it occasionally may fail. For more complex crawlers, you can use encoding detection from Apache Tika (`https://tika.apache.org/`).

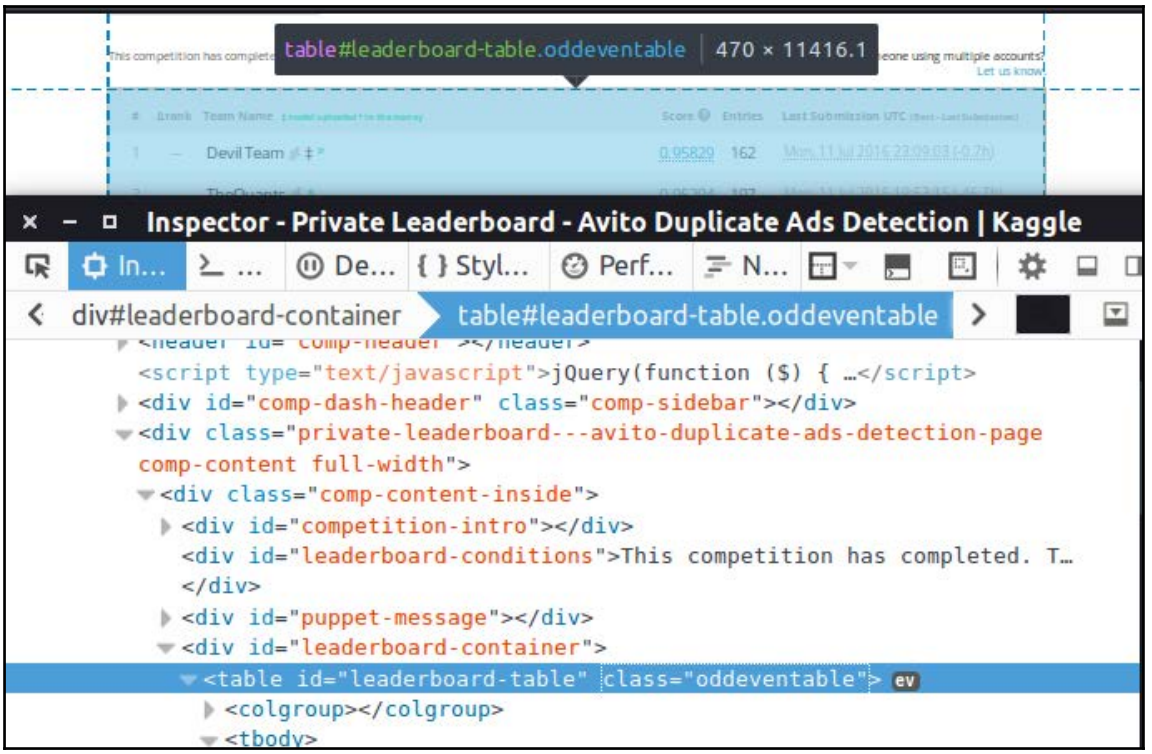
The preceding method returns raw HTML data, which is not useful in itself; most of the time we are interested in the text content rather than markup. There are libraries for processing HTML and one of them is Jsoup:

```
<dependency>
  <groupId>org.jsoup</groupId>
  <artifactId>jsoup</artifactId>
  <version>1.9.2</version>
</dependency>
```

Let's consider an example. Kaggle.com is a website for hosting data science competitions, and for each competition there is a leader board that shows the performance of each participant. Suppose you are interested in extracting this information from `https://www.kaggle.com/c/avito-duplicate-ads-detection/leaderboard` as shown in the following screenshot:



This information is contained in a table, and to extract the data from this table we need to find an anchor that uniquely points to this table. To do that, you can have a look at the page using an inspector (pressing *F12* in Mozilla Firefox or Google Chrome will open the **Inspector** window):



Using the **Inspector**, we can notice that the ID of the table is `leaderboard-table`, and to get this table in Jsoup, we can use the following CSS selector, `#leaderboard-table`. Since we are actually interested in the rows of the table, we will use `#leaderboard-table tr`.

The information about the team name is contained in the third column of the table in the list. Thus, to extract it, we need to get the third `<td>` tag and then look inside its `<a>` tag. Likewise, to extract the score, we get the content of the fourth `<td>` tag.

The code for doing this is as follows:

```
Map<String, Double> result = new HashMap<>();

String rawHtml =
UrlUtils.request("https://www.kaggle.com/c/avito-duplicate-ads-detection/le
```

```
aderboard");
Document document = Jsoup.parse(rawHtml);
Elements tableRows = document.select("#leaderboard-table tr");
for (Element tr : tableRows) {
    Elements columns = tr.select("td");
    if (columns.isEmpty()) {
        continue;
    }

    String team = columns.get(2).select("a.team-link").text();
    double score = Double.parseDouble(columns.get(3).text());
    result.put(team, score);
}

Comparator<Map.Entry<String, Double>> byValue =
Map.Entry.comparingByValue();
result.entrySet().stream()
        .sorted(byValue.reversed())
        .forEach(System.out::println);
```

Here we reuse the `UrlUtils.request` function to get the HTML that we defined previously, and then process it with Jsoup.

Jsoup makes use of CSS selectors for accessing items inside the parsed HTML document. To learn more about them, you can read the related documentation, which is accessible at `https://jsoup.org/cookbook/extracting-data/selector-syntax`.

JSON

JSON is becoming more and more popular as a way of communicating between web services, steadily displacing XML and other formats. Knowing how to process it lets you extract data from a huge variety of data sources available on the Internet.

There are quite a few JSON libraries available for Java. Jackson is one of them and there is its simplified version, called `jackson-jr`, which works for most simple cases when all we need is to quickly extract data from JSON. To add it, use the following:

```
<dependency>
  <groupId>com.fasterxml.jackson.jr</groupId>
  <artifactId>jackson-jr-all</artifactId>
  <version>2.8.1</version>
</dependency>
```

To illustrate it, let's consider a simple API that returns JSON. We can use `http://www.jsontest.com/`, which provides a number of dummy web services. One of them is an MD5 service at `http://md5.jsontest.com`; given a string it returns its MD5 hash.

Here is an example of its output:

```
{
  "original": "mastering java for data science",
  "md5": "f4c8637d7274f13b58940ff29f669e8a"
}
```

Let us use it:

```
String text = "mastering java for data science";
String json = UrlUtils.request("http://md5.jsontest.com/?text=" +
text.replace(' ', '+'));

Map<String, Object> map = JSON.std.mapFrom(json);
System.out.println(map.get("original"));
System.out.println(map.get("md5"));
```

In this example, the JSON response of a web service is quite simple. However, there are more complex cases with lists and nested objects. For example, `www.github.com` provides a number of APIs, one of which is `https://api.github.com/users/alexeygrigorev/repos`. For a given user it returns all their repositories. It has a list of objects and each object has a nested object.

In languages with dynamic typing, such as Python, it is quite simple--the language does not force you to have a specific type, which, for this particular case, is good. In Java, however, the static type system requires defining a type; every time you need to extract something, you need to do casting.

For example, if we want to get the element ID of the first object, we would need to do something like this:

```
String username = "alexeygrigorev";
String json = UrlUtils.request("https://api.github.com/users/" + username +
"/repos");

List<Map<String, ?>> list = (List<Map<String, ?>>) JSON.std.anyFrom(json);
String name = (String) list.get(0).get("name");
System.out.println(name);
```

As you can see, we need to do a lot of type casting and the code quickly becomes pretty cluttered. A solution to that may be using a query language similar to Xpath, called JsonPath. An implementation available for Java is accessible at `https://github.com/jayway/JsonPath`. To use it, add the following:

```
<dependency>
  <groupId>com.jayway.jsonpath</groupId>
  <artifactId>json-path</artifactId>
  <version>2.2.0</version>
</dependency>
```

If we want to retrieve all repositories, which are written in Java and have at least one start, then the following query will do it:

```
ReadContext ctx = JsonPath.parse(json);
String query = "$..[?(@.language=='Java' && @.stargazers_count >
0)]full_name";
List<String> javaProjects = ctx.read(query);
```

It definitely will save some time for simple data manipulations such as filtering, but, unfortunately, for more complex things, you may still need to do manual conversions with a lot of casting.



For more complex queries (for example, sending a POST request), it is better to use a special library such as Apache HttpComponents (`https://hc.apache.org/`).

Databases

In organizations, data is usually kept in relational databases. Java defines **Java Database Connectivity** (**JDBC**) as an abstraction for accessing any database that supports SQL.

In our example, we will use MySQL, which can be downloaded from `https://www.mysql.com/`, but in principle it can be any other database, such as PostgreSQL, Oracle, MS SQL and many others. To connect to a MySQL server, we can use a JDBC MySQL driver:

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.39</version>
</dependency>
```

If you would like to use a different database, then you can use your favorite search engine and find the suitable JDBC driver. The interaction code will remain the same, and if you use the standard SQL, the query code should not change as well.

For example, we will use the same data we generated for the CSV example. First, we will load it to the database, and then do a simple select.

Let's define the following schema:

```
CREATE SCHEMA &grave;people&grave; DEFAULT CHARACTER SET utf8 ;
CREATE TABLE &grave;people&grave;.&grave;people&grave; (
  &grave;person_id&grave; INT UNSIGNED NOT NULL AUTO_INCREMENT,
  &grave;name&grave; VARCHAR(45) NULL,
  &grave;email&grave; VARCHAR(100) NULL,
  &grave;country&grave; VARCHAR(45) NULL,
  &grave;salary&grave; INT NULL,
  &grave;experience&grave; INT NULL,
  PRIMARY KEY (&grave;person_id&grave;));
```

Now, to connect to the database, we typically use the `DataSource` abstraction. The MySQL driver provides an implementation: `MysqlDataSource`:

```
MysqlDataSource datasource = new MysqlDataSource();
datasource.setServerName("localhost");
datasource.setDatabaseName("people");
datasource.setUser("root");
datasource.setPassword("");
```

Now using the `DataSource` object, we can load the data. There are two ways of doing it; the simple way, when we insert each object individually, and batch mode, where we first prepare a batch, and then insert all the objects of a batch. The batch mode option typically works faster.

Let's first look at the usual mode:

```
try (Connection connection = datasource.getConnection()) {
    String sql = "INSERT INTO people (name, email, country, salary,
experience) VALUES (?, ?, ?, ?, ?);";
    try (PreparedStatement statement = connection.prepareStatement(sql)) {
        for (Person person : people) {
            statement.setString(1, person.getName());
            statement.setString(2, person.getEmail());
            statement.setString(3, person.getCountry());
            statement.setInt(4, person.getSalary());
            statement.setInt(5, person.getExperience());
            statement.execute();
        }
```

```
    }
}
```



Note that the enumeration of indexes starts from 1 in JDBC, not from 0.

Batch is very similar. To prepare the batches, we first use a `Lists.partition` function from Guava, and chuck all the data into batches of 50 objects. Then each object of a chunk is added to a batch with the `addBatch` function:

```
List<List<Person>> chunks = Lists.partition(people, 50);

try (Connection connection = datasource.getConnection()) {
    String sql = "INSERT INTO people (name, email, country, salary,
experience) VALUES (?, ?, ?, ?, ?);";
    try (PreparedStatement statement = connection.prepareStatement(sql)) {
        for (List<Person> chunk : chunks) {
            for (Person person : chunk) {
                statement.setString(1, person.getName());
                statement.setString(2, person.getEmail());
                statement.setString(3, person.getCountry());
                statement.setInt(4, person.getSalary());
                statement.setInt(5, person.getExperience());
                statement.addBatch();
            }
            statement.executeBatch();
        }
    }
}
```

Batch mode is faster than the usual way of processing the data, but requires more memory. If you need to process a lot of data and care about speed, then batch mode is a better choice, but it makes the code a bit more complex. For that reason, it may be better to use the simpler approach.

Now, when the data is loaded, we can query the database. For instance, let us select all the people from a country:

```
String country = "Greenland";
try (Connection connection = datasource.getConnection()) {
    String sql = "SELECT name, email, salary, experience FROM people WHERE
country = ?;";
    try (PreparedStatement statement = connection.prepareStatement(sql)) {
        List<Person> result = new ArrayList<>();
```

```
            statement.setString(1, country);
            try (ResultSet rs = statement.executeQuery()) {
                while (rs.next()) {
                    String name = rs.getString(1);
                    String email = rs.getString(2);
                    int salary = rs.getInt(3);
                    int experience = rs.getInt(4);
                    Person person = new Person(name, email, country, salary,
    experience);
                    result.add(person);
                }
            }
        }
    }
```

This way we can execute any SQL query we want and put the results into Java objects for further processing.



You may have noticed that there is a lot of boilerplate code in JDBC. The boilerplate can be reduced with the Spring JDBC Template library (see `http://www.springframework.org`).

DataFrames

DataFrames are a convenient way of representing tabular data in memory. Originally DataFrames came from the R programming language, but they are now common in other languages; for example, in Python the pandas library offers a DataFrame data structure similar to R's.

The usual way of storing data in Java is lists, maps, and other collections of objects. We can think of these collections as tables, but we can assess the data only by row. However, for data science manipulation columns are equally important, and this is where DataFrames are helpful.

For example, they allow you to apply the same function over all values of the same column or look at the distribution of the values.

In Java there are not so many mature implementations, but there are some that have all the needed functionality. In our examples we will use `joinery`:

```
<dependency>
  <groupId>joinery</groupId>
  <artifactId>joinery-dataframe</artifactId>
```

```
  <version>1.7</version>
</dependency>
```

Unfortunately, `joinery` is not available on Maven Central; thus, to include it to a Maven project, you need to point to `bintray`, another Maven repository. To do this, add this `repository` to the repositories section in the `pom` file:

```
<repository>
  <id>bintray</id>
  <url>http://jcenter.bintray.com</url>
</repository>
```

Joinery depends on `Apache POI`, so you need to add it as well:

```
<dependency>
  <groupId>org.apache.poi</groupId>
  <artifactId>poi</artifactId>
  <version>3.14</version>
</dependency>
```

With Joinery it is very easy to read the data:

```
DataFrame<Object> df = DataFrame.readCsv("data/csv-example-
generatedata_com.csv");
```

Once the data is read, we can access not only every row, but also every column of the DataFrame. Given a column name, Joinery returns a `List` of values that are stored in the column, and we can use it to apply various transformations to it.

For example, suppose we want to associate every country that we have with a unique index. We can do it like this:

```
List<Object> country = df.col("country");
Map<String, Long> map = LazyReact.sequentialBuilder()
        .from(country)
        .cast(String.class)
        .distinct()
        .zipWithIndex()
        .toMap(Tuple2::v1, Tuple2::v2);

List<Object> indexes =
country.stream().map(map::get).collect(Collectors.toList());
```

After that, we can drop the old column with `country` and include the new index column instead:

```
df = df.drop("country");
df.add("country_index", indexes);
System.out.println(df);
```

Joinery can do a lot more--group by, joins, pivoting, and creating design matrices for machine learning models. We will use it again in the future in almost all the chapters. Meanwhile, you can read more about joinery at `https://cardillo.github.io/joinery/`.

Search engine - preparing data

In the first chapter, we introduced the running example, building a search engine. A search engine is a program that, given a query from the user, returns results ordered by relevance with respect to the query. In this chapter, we will perform the first steps--obtaining and processing data.

Suppose we are working on a web portal where users generate a lot of content, but they have trouble finding what other people have created. To overcome this problem, we propose to build a search engine, and product management has identified the typical queries that the users will put in.

For example, "Chinese food", "homemade pizza", and "how to learn programming" are typical queries from this list.

Now we need to collect the data. Luckily for us, there are already search engines on the Internet that can take in a query and return a list of URLs they consider relevant. We can use them for obtaining the data. You probably already know such engines--Google or Bing, to name just two.

Thus, we can apply what we learned in this chapter and parse the data from Google, Bing, or any other search engine using JSoup. Alternatively, it is possible to use services such as `https://flow-app.com/` to do this extraction for you, but it requires registration.

In the end, what we need is a query and a list of the most relevant URLs. Extraction of the relevant URLs is left as an exercise, but we already prepared some results that you can use if you wish: for each query there are 30 most relevant pages from the first three pages of the search results. Additionally, you can find useful code for crawling in the code bundle.

Now, when we have the URLs, we are interested in retrieving them and saving their HTML code. For this purpose, we need a crawler that is a bit smarter than what we already have in our `UrlUtils.request`.

One particular thing that we must add is timeouts: some pages take a lot of time to load, because they are either big or the server is experiencing some troubles and takes a while to respond. In these cases, it makes sense to give up when a page cannot be downloaded in, for example, 30 seconds.

In Java, this can be done with `Executors`. First, let's create a `Crawler` class, and declare the `executor` field:

```
int numProc = Runtime.getRuntime().availableProcessors();
executor = Executors.newFixedThreadPool(numProc);
```

Then, we can use this executor as follows:

```
try {
    Future<String> future = executor.submit(() -> UrlUtils.request(url));
    String result = future.get(30, TimeUnit.SECONDS);
    return Optional.of(result);
} catch (TimeoutException e) {
    LOGGER.warn("timeout exception: could not crawl {} in {} sec", url,
timeout);
    return Optional.empty();
}
```

This code will drop pages that are taking too long to retrieve.

We need to store the crawled HTML pages somewhere. There are a few options: a bunch of HTML files on the filesystem, a relational store such as MySQL, or a key-value store. The key-value storage looks like the best choice because we have a key, the URL, and the value, the HTML. For that we can use MapDB, a pure Java key-value storage that implements the `Map` interface. In essence, it's a `Map` that is supported by a file on disk.

Since it is pure Java, all you need to do to use it is to include its dependency:

```
<dependency>
  <groupId>org.mapdb</groupId>
  <artifactId>mapdb</artifactId>
  <version>3.0.1</version>
</dependency>
```

And now we can use it:

```
DB db = DBMaker.fileDB("urls.db").closeOnJvmShutdown().make();
HTreeMap<?, ?> htreeMap = db.hashMap("urls").createOrOpen();
Map<String, String> urls = (Map<String, String>) htreeMap;
```

Since it implements the `Map` interface, it can be treated as a usual `Map` and we can put the HTML there. So, let's read the relevant URLs, download their HTML, and save it to the map:

```
Path path = Paths.get("data/search-results.txt");
List<String> lines = FileUtils.readLines(path.toFile(),
StandardCharsets.UTF_8);

lines.parallelStream()
    .map(line -> line.split("t"))
    .map(split -> "http://" + split[2])
    .distinct()
    .forEach(url -> {
        try {
            Optional<String> html = crawler.crawl(url);
            if (html.isPresent()) {
                LOGGER.debug("successfully crawled {}", url);
                urls.put(url, html.get());
            }
        } catch (Exception e) {
            LOGGER.error("got exception when processing url {}", url, e);
        }
    });
```

Here we do that in a `parallelStream` to make the execution faster. The timeouts will ensure that it finishes in a reasonable amount of time.

For starters, let's extract something very simple from the pages, as follows:

- Length of the URL
- Length of the title
- Whether or not the query is contained in the title
- Length of entire text in body
- Number of `<h1>-<h6>` tags
- Number of links

To hold this information, we can create a special class, `RankedPage`.

```
public class RankedPage {
    private String url;
    private int position;
    private int page;
    private int titleLength;
    private int bodyContentLength;
    private boolean queryInTitle;
    private int numberOfHeaders;
    private int numberOfLinks;
    // setters, getters are omitted
}
```

Now, let us crawl the for each page create an object of this class. We use `flatMap` for this because for some URLs there is no HTML data.

```
Stream<RankedPage> pages = lines.parallelStream().flatMap(line -> {
    String[] split = line.split("t");
    String query = split[0];
    int position = Integer.parseInt(split[1]);
    int searchPageNumber = 1 + (position - 1) / 10; // converts position to
a page number
    String url = "http://" + split[2];
    if (!urls.containsKey(url)) { // no crawl available
        return Stream.empty();
    }

    RankedPage page = new RankedPage(url, position, searchPageNumber);
    String html = urls.get(url);
    Document document = Jsoup.parse(html);
    String title = document.title();
    int titleLength = title.length();
    page.setTitleLength(titleLength);

    boolean queryInTitle =
title.toLowerCase().contains(query.toLowerCase());
    page.setQueryInTitle(queryInTitle);

    if (document.body() == null) { // no body for the document
        return Stream.empty();
    }
    int bodyContentLength = document.body().text().length();
    page.setBodyContentLength(bodyContentLength);

    int numberOfLinks = document.body().select("a").size();
    page.setNumberOfLinks(numberOfLinks);
```

```
        int numberOfHeaders =
    document.body().select("h1,h2,h3,h4,h5,h6").size();
        page.setNumberOfHeaders(numberOfHeaders);

        return Stream.of(page);
    });
```

In this piece of code, for each page we look up its HTML. If it is not crawled--we skip the page; then we parse the HTML and retrieve the preceding basic features.

This is only a small fraction of the possible page features that we can compute. Later on, we will build on this and add more features.

In this example, we get a stream of pages. We can do anything we want with this stream, for example, save it to JSON or convert it to a DataFrame. The code bundle that comes with this book has some examples and shows how to do these types of conversion. For example, conversion from a list of Java objects to a Joinery `DataFrame` is available in the `BeanToJoinery` utility class.

Summary

There are a few steps for approaching any data science problem, and the data preparation step is one of the first. The standard Java API has a tremendous number of tools that make this task possible, and there are a lot of libraries that make it a lot easier.

In this chapter, we discussed many of them, including extensions to the Java API such as Google Guava; we talked about ways to read the data from different sources such as text, HTML, and databases; and finally we covered the DataFrame, a useful structure for manipulating tabular data.

In the next chapter, we will take a closer look at the data that we extracted in this chapter and perform Exploratory Data Analysis.

3
Exploratory Data Analysis

the previous chapter, we covered data processing, which is an important step for transforming data into a form usable for analysis. In this chapter, we take the next logical step after cleaning and look at data. This step is called **Exploratory Data Analysis** (**EDA**), and it consists of summarizing data and creating visualizations.

In this chapter, we will cover the following topics:

- Summary statistics with Apache Commons Math and Joinery
- Interactive shells for EDA in Java and JVM

By the end of this chapter, you will know how to calculate summary statistics and create simple graphs with Joinery.

Exploratory data analysis in Java

Exploratory Data Analysis is about taking a dataset and extracting the most important information from it, in such a way that it is possible to get an idea of what the data looks like. This includes two main parts:

The summarization step is very helpful for understanding data. For numerical variables, in this step we calculate the most important sample statistics:

- The extremes (the minimal and the maximal values)
- The mean value, or the sample average
- The standard deviation, which describes the spread of the data

Often we consider other statistics, such as the median and the quartiles (25% and 75%).

As we have already seen in the previous chapter, Java offers a great set of tools for data preparation. The same set of tools can be used for EDA, and especially for creating summaries.

Search engine datasets

In this chapter, we will use our running example--building a search engine. In `Chapter 2`, *Data Processing Toolbox*, we extracted some data from HTML pages returned by a search engine. This dataset included some numerical features, such as the length of the title and the length of the content.

For the purposes of storing these features, we created the following class:

```
public class RankedPage {
    private String url;
    private int position;
    private int page;
    private int titleLength;
    private int bodyContentLength;
    private boolean queryInTitle;
    private int numberOfHeaders;
    private int numberOfLinks;
    // setters, getters are omitted
}
```

It is interesting to see if this information can be useful for the search engine. For example, given a URL, we may want to know whether it is likely to appear on the first page of the engine output or not. Looking at the data by means of EDA will help us know if it is possible.

Additionally, real-world data is rarely clean. We will use EDA to try to spot some strange or problematic observations.

Let's get started. We saved the data in the JSON format, and now we can read it back using streams and Jackson:

```
Path path = Paths.get("./data/ranked-pages.json");
try (Stream<String> lines = Files.lines(path)) {
    return lines.map(line -> parseJson(line)).collect(Collectors.toList());
}
```

This is the body of a function that returns a list of `RankedPage` objects. We read them from the `ranked-page.json` file. And then we use the `parseJson` function to convert the JSON to the Java class:

```
JSON.std.beanFrom(RankedPage.class, line);
```

After reading the data, we can analyze it. Usually, the first step in the analysis is looking at the summary statistics, and we can use Apache Commons Math for that.

Apache Commons Math

Once we read the data, we can calculate the statistics. As we already mentioned earlier, we are typically interested in summaries such as min, max, mean, standard deviation, and so on. We can use the Apache Commons Math library for that. Let's include it in `pom.xml`:

```
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-math3</artifactId>
  <version>3.6.1</version>
</dependency>
```

There is a `SummaryStatistics` class for calculating the summaries. Let's use it to calculate some statistics about the distribution of body content length of the pages we crawled:

```
SummaryStatistics statistics = new SummaryStatistics();
data.stream().mapToDouble(RankedPage::getBodyContentLength)
    .forEach(statistics::addValue);
System.out.println(statistics.getSummary());
```

Here, we create `SummaryStatistics` objects and add all body content lengths. After that, we can call a `getSummary` method to get all summary stats at once. This will print the following:

```
StatisticalSummaryValues:
n: 4067
min: 0.0
max: 8675779.0
mean: 14332.239242685007
std dev: 144877.54551111493
variance: 2.0989503193325176E10
sum: 5.8289217E7
```

The `DescriptiveStatistics` method is another useful class from this library. It allows getting more values, such as median and percentiles, and percentiles; give a better idea of what the data looks like:

```
double[] dataArray = data.stream()
        .mapToDouble(RankedPage::getBodyContentLength)
        .toArray();
DescriptiveStatistics desc = new DescriptiveStatistics(dataArray);
System.out.printf("min: %9.1f%n", desc.getMin());
System.out.printf("p05: %9.1f%n", desc.getPercentile(5));
System.out.printf("p25: %9.1f%n", desc.getPercentile(25));
System.out.printf("p50: %9.1f%n", desc.getPercentile(50));
System.out.printf("p75: %9.1f%n", desc.getPercentile(75));
System.out.printf("p95: %9.1f%n", desc.getPercentile(95));
System.out.printf("max: %9.1f%n", desc.getMax());
```

This will produce the following output:

```
min: 0.0
p05: 527.6
p25: 3381.0
p50: 6612.0
p75: 11996.0
p95: 31668.4
max: 8675779.0
```

From the output, we can note that the minimum length is zero, which is strange; most likely, there was a data processing issue. Also, the maximal value is very high, which suggests that there are outliers. Later on, it will make sense to exclude these values from our analysis.

Probably the pages with zero content length are crawling errors. Let's see the proportion of these pages:

```
double proportion = data.stream()
    .mapToInt(p -> p.getBodyContentLength() == 0 ? 1 : 0)
    .average().getAsDouble();
System.out.printf("proportion of zero content length: %.5f%n", proportion);
```

We see that not so many pages have zero length, so it is quite safe to just drop them.

Later on, in the next chapter, we will try to predict whether a URL comes from the first search page result or not. If some features have different values for each page, then a machine learning model will be able to see this difference and use it for more accurate predictions. Let's see whether the value for content length is similar across the different pages.

For this purpose, we can group the URLs by page and calculate the mean content length. As we already know, Java streams can be used to do this:

```
Map<Integer, List<RankedPage>> byPage = data.stream()
    .filter(p -> p.getBodyContentLength() != 0)
    .collect(Collectors.groupingBy(RankedPage::getPage));
```

Note that we added a filter for empty pages, so they do not appear in the groups. Now, we can use the groups to calculate the average:

```
List<DescriptiveStatistics> stats = byPage.entrySet().stream()
    .sorted(Map.Entry.comparingByKey())
    .map(e -> calculate(e.getValue(), RankedPage::getBodyContentLength))
    .collect(Collectors.toList());
```

Here, `calculate` is a function that takes in a collection, computes the provided function on every element (using `getBodyContentLength` in this case), and creates a `DescriptiveStatistics` object from it:

```
private static DescriptiveStatistics calculate(List<RankedPage> data,
            ToDoubleFunction<RankedPage> getter) {
    double[] dataArray = data.stream().mapToDouble(getter).toArray();
    return new DescriptiveStatistics(dataArray);
}
```

Now, in the list, you will have the descriptive statistics for each group (page, in this case). Then, we can display them in any way we want. Consider the following example:

```
Map<String, Function<DescriptiveStatistics, Double>> functions = new
LinkedHashMap<>();
functions.put("min", d -> d.getMin());
functions.put("p05", d -> d.getPercentile(5));
functions.put("p25", d -> d.getPercentile(25));
functions.put("p50", d -> d.getPercentile(50));
functions.put("p75", d -> d.getPercentile(75));
functions.put("p95", d -> d.getPercentile(95));
functions.put("max", d -> d.getMax());
System.out.print("page");

for (Integer page : byPage.keySet()) {
    System.out.printf("%9d ", page);
}
System.out.println();

for (Entry<String, Function<DescriptiveStatistics, Double>> pair :
functions.entrySet()) {
    System.out.print(pair.getKey());
    Function<DescriptiveStatistics, Double> function = pair.getValue();
```

```
        System.out.print(" ");
        for (DescriptiveStatistics ds : stats) {
            System.out.printf("%9.1f ", function.apply(ds));
        }
        System.out.println();
    }
```

This produces the following output:

```
page 0 1 2
min 5.0 1.0 5.0
p05 1046.8 900.6 713.8
p25 3706.0 3556.0 3363.0
p50 7457.0 6882.0 6383.0
p75 13117.0 12067.0 11309.8
p95 42420.6 30557.2 27397.0
max 390583.0 8675779.0 1998233.0
```

The output suggests that the distribution of content length is different across URLs coming from different pages of search engine results. Thus, this can potentially be useful when predicting the search page number for a given URL.

Joinery

You might notice that the code we just wrote is quite verbose. Of course, it is possible to put it inside a helper function and call it when needed, but there is another more concise way of computing these statistics--with joinery and its DataFrames.

In Joinery, the `DataFrame` object has a method called `describe`, which creates a new DataFrame with summary statistics:

```
List<RankedPage> pages = Data.readRankedPages();
DataFrame<Object> df = BeanToJoinery.convert(pages, RankedPage.class);
df = df.retain("bodyContentLength", "titleLength", "numberOfHeaders");
DataFrame<Object> describe = df.describe();
System.out.println(describe.toString());
```

In the preceding code, `Data.readRankedPages` is a helper method that reads JSON data and converts it to a list of Java objects, and `BeanToJoinery.convert` is a helper class that converts a list of Java objects to a `DataFrame`.

Then, we keep only three columns and drop everything else. The following is the output:

```
body   contentLength   numberOfHeaders  titleLength
count  4067.00000000   4067.00000000    4067.00000000
mean   14332.23924269  25.25325793      46.17334645
std    144877.5455111  32.13788062      27.72939822
var    20989503193.32  1032.84337047    768.91952552
max    8675779.000000  742.00000000     584.00000000
min    0.00000000      0.00000000       0.00000000
```

We can also look at the means across different groups, for example, across different pages. For that, we can use the `groupBy` method:

```
DataFrame<Object> meanPerPage = df.groupBy("page").mean()
    .drop("position")
    .sortBy("page")
    .transpose();
System.out.println(meanPerPage);
```

Apart from applying mean after `groupBy`, we also remove one column position because we already know that position will be different for different pages. Additionally, we apply the transpose operation at the end; this is a trick to make the output fit into a screen when there are many columns. This produces the following output:

```
page 0 1 2
bodyContentLength 12577 18703 11286
numberOfHeaders 30 23 21
numberOfLinks 276 219 202
queryInTitle 0 0 0
titleLength 46 46 45
```

We can see that the averages are quite different for some variables. For other variables, such as `queryInTitle`, there does not seem to be any difference. However, remember that this is a Boolean variable, so the mean is between 0 and 1. For some reason, Joinery decided not to show the decimal part here.

Now, we know how to compute some simple summary statistics in Java, but to do that, we first need to write some code, compile it, and then run it. This is not the most convenient procedure, and there are better ways to do it interactively, that is, avoiding compilation and getting the results right away. Next, we will see how to do it in Java.

Interactive Exploratory Data Analysis in Java

Java is a statically typed programming language and code written in Java needs compiling. While Java is good for developing complex data science applications, it makes it harder to interactively explore the data; every time, we need to recompile the source code and re-run the analysis script to see the results. This means that, if we need to read some data, we will have to do it over and over again. If the dataset is large, the program takes more time to start.

So it is hard to interact with data and this makes EDA more difficult in Java than in other languages. In particular, **Read-Evaluate-Print Loop** (**REPL**), an interactive shell, is quite an important feature for doing EDA.

Unfortunately, Java 8 does not have REPL, but there are several alternatives:

- Other interactive JVM languages such as JavaScript, Groovy, or Scala
- Java 9 with jshell
- Completely alternative platforms such as Python or R

In this chapter, we will look at the first two options--JVM languages and Java 9's REPL.

JVM languages

As you probably know, the Java platform is not only the Java programming language, but also the **Java Virtual Machine** (JVM) can run code from other JVM languages. There are a number of languages that run on JVM and have REPL, for example, JavaScript, Scala, Groovy, JRuby, and Jython. There are many more. All these languages can access any code written in Java and they have interactive consoles.

For example, Groovy is very similar to Java, and prior to Java 8 almost any code written in Java could be run in Groovy. However, for Java 8, this is no longer the case. Groovy does not support the new Java syntax for lambda expressions and functional interfaces, so we will not be able to run most of the code from this book there.

Scala is another popular functional JVM language, but its syntax is very different from Java. It is a very powerful and expressive language for data processing, it has a nice interactive shell, and there are many libraries for doing data analysis and data science.

Additionally, there are a couple of JavaScript implementations available for JVM. One of them is Nashorn, which comes with Java 8 out-of-the-box; there is no need to include it as a separate dependency. Joinery also comes in with a built-in interactive console, which utilizes JavaScript, and later in this chapter, we will see how to use it.While all these languages are nice, they are beyond the scope of this book. You can learn more about them from these books:

- *Groovy in Action, Dierk Konig, Manning*
- *Scala Data Analysis Cookbook, Arun Manivannan, Packt Publishing*

Interactive Java

It would not be fair to say that Java is a 100% not-interactive language; there are some extensions that provide a REPL environment directly for Java.

One such environment is a scripting language (BeanShell) that looks exactly like Java. However, it is quite old and the new Java 8 syntax is not supported, so it is not very useful for doing interactive data analysis.

What is more interesting is Java 9, which comes with an integrated REPL called JShell and supports autocompletion on tab, Java Streams, and the Java 8 syntax for lambda expressions. At the time of writing, Java 9 is only available as an Early Access Release. You can download it from `https://jdk9.java.net/download/`.

Starting the shell is easy:

```
$ jshell
```

But typically you would like to access some libraries, and for that they need to be on the classpath. Usually, we use Maven for managing the dependencies, so we can run the following to copy all the `jar` libraries specified in the `pom` file to a directory of our choice:

```
mvn dependency:copy-dependencies -DoutputDirectory=lib
mvn compile
```

After doing it, we can run the shell like this:

```
jshell -cp lib/*:target/classes
```

If you are on Windows, replace the colon with a semicolon:

```
jshell -cp lib/*;target/classes
```

However, our experiments showed that JShell is unfortunately quite raw yet and sometimes crashes. At the time of writing, the plan is to release it at the end of March 2017. For now, we will not cover JShell in more detail, but all the code from the first half of the chapter should work in that console with no additional configuration. And what is more, we should be able to see the output immediately.

By now, we have used Joinery a couple of times already, and it has also some support for performing simple EDA. Next, we will look at how to do the analysis with the Joinery shell.

Joinery shell

Joinery has already proved useful multiple times for doing data processing and simple EDA. It has an interactive shell for doing EDA and you get an answer instantly.

If the data is already in the CSV format, then the Joinery shell can be called from the system console:

```
$ java joinery.DataFrame shell
```

You can see the examples at `https://github.com/cardillo/joinery`. So if your data is already in CSV, you are good to go, just follow the instructions from there.

Here, in this book, we will look at a more complex example when the DataFrame is not in CSV. In our case the data is in JSON, and the Joinery shell does not support that format, so we need to do some pre-processing first.

What we can do is to create a DataFrame object inside the Java code and then create the interactive shell and pass the DataFrame there. Let's see how it can be done.

But before we can do this, we need to add a few dependencies to make it possible. First, the Joinery shell uses JavaScript, but it does not use the Nashorn engin shipped along with the JVM; instead it uses the Mozilla's engine called **Rhino**. Thus, we need to include it to our `pom`:

```
<dependency>
  <groupId>rhino</groupId>
  <artifactId>js</artifactId>
  <version>1.7R2</version>
</dependency>
```

Second, it relies on a special library for autocompletion, `jline`. Let's add it as well:

```
<dependency>
  <groupId>jline</groupId>
  <artifactId>jline</artifactId>
  <version>2.14.2</version>
</dependency>
```

Using Maven gives you a lot of flexibility; it is simpler and does not require you to download all the libraries manually and build Joinery from the source code to be able to execute the shell. So, we let Maven take care of it.

Now we are ready to use it:

```
List<RankedPage> pages = Data.readRankedPages();
DataFrame<Object> dataFrame = BeanToJoinery.convert(pages,
'RankedPage.class);
Shell.repl(Arrays.asList(dataFrame));
```

Let's save this code to a `chapter03.JoineryShell` class. After that, we can run it with the following Maven command:

```
mvn exec:java -Dexec.mainClass="chapter03.JoineryShell"
```

This will bring us to the Joinery shell:

```
# DataFrames for Java -- null, 1.7-8e3c8cf
# Java HotSpot(TM) 64-Bit Server VM, Oracle Corporation, 1.8.0_91
# Rhino 1.7 release 2 2009 03 22
>
```

All the DataFrames that we pass to the Shell object in Java are available in the frames variable in the shell. So, to get `DataFrame`, we can do this:

```
> var df = frames[0]
```

To see the content of `DataFrame`, just write its name:

```
> df
```

And you will see a couple of first rows of `DataFrame`. Note that the autocompletion works as expected:

```
> df.<tab>
```

You will see a list of options.

We can use this shell to call the same methods on the DataFrames as we would use in the usual Java application. For example, you can compute the mean as follows:

```
> df.mean().transpose()
```

We will see the following output:

```
bodyContentLength 14332.23924269
numberOfHeaders 25.25325793
numberOfLinks 231.16867470
page 1.03221047
position 18.76518318
queryInTitle 0.59822965
titleLength 46.17334645
```

Alternatively. we can perform the same `groupBy` example:

```
> df.drop('position').groupBy('page').mean().sortBy('page').transpose()
```

This will produce the following output:

```
page 0 1 2
bodyContentLength 12577 18703 11286
numberOfHeaders 30 23 21
numberOfLinks 276 219 202
queryInTitle 0 0 0
titleLength 46 46 45
```

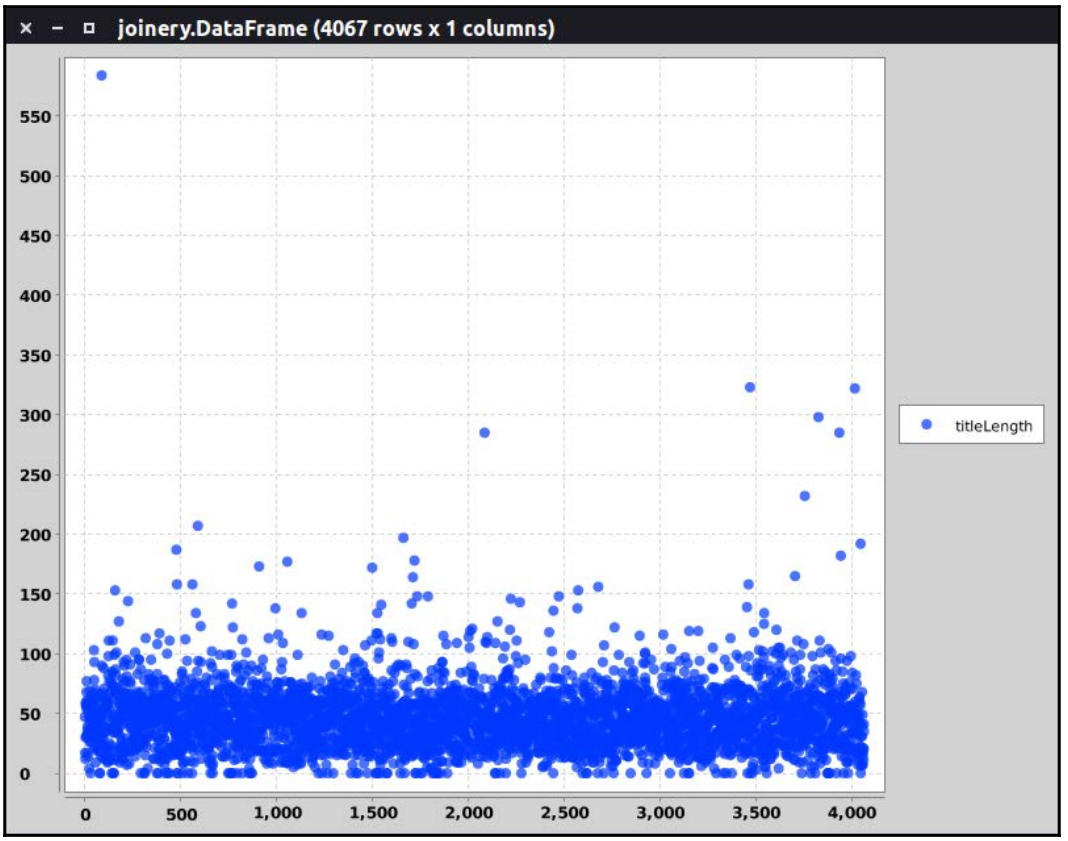
Finally, it is also possible to create some simple plots with Joinery. For that, we will need to use an additional library. For plotting, Joinery uses `xchart`. Let's include it:

```
<dependency>
  <groupId>com.xeiam.xchart</groupId>
  <artifactId>xchart</artifactId>
  <version>2.5.1</version>
</dependency>
```

And run the console again. Now we can use the `plot` function:

```
> df.retain('titleLength').plot(PlotType.SCATTER)
```

And we will see this:



Here, we see an outlier whose title length is more than 550 characters. Let's remove everything that is above 200 and have a look at the picture again. Also, remember that there are some zero-length content pages, so we can filter them out as well.

To keep only those rows that satisfy some condition, we use the `select` method. It takes a function, which is applied to every row; if a function returns `true`, the row is kept.

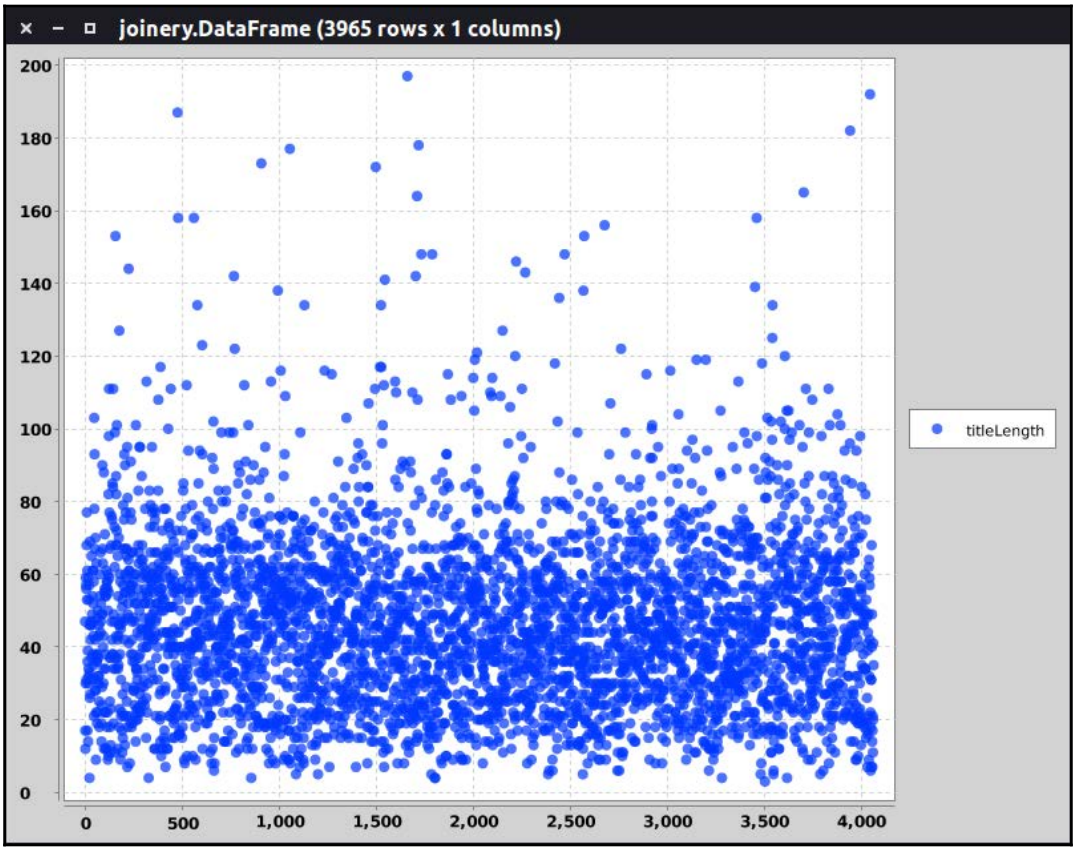
We can use it like this:

```
> df.retain('titleLength')
    .select(function(list) { return list.get(0) <= 200; })
    .select(function(list) { return list.get(0) > 0;})
    .plot(PlotType.SCATTER)
```



The line-breaks in the preceding code are added for better readability, but they will not work in the console, so do not use them.

Now, we get a clearer picture:



Unfortunately, the plotting capabilities of joinery are quite limited, and it takes a lot of effort to produce graphs using `xchart`.

As we already know, in Joinery, it is easy to calculate statistics across different groups; we just need to use the `groupBy` method. However, it is not possible to easily use this method for plotting data so that it is easy to compare the distributions for each group.

There are other tools that you can also use for EDA:

- Weka, which is written in Java, is a library used for performing data mining. It has a GUI interface for performing EDA.
- Smile, another Java library, has a Scala shell and a smile-plot library for creating visualizations.

Unfortunately, Java is typically not the ideal choice for performing EDA and there are other better suited dynamic languages for that. For example, R and Python are ideal for this task, but covering them is beyond the scope of this book. You can learn more about them from the following book:

- Mastering Data Analysis with R, Gergely Daroczi
- Python Machine Learning, Sebastian Raschka

Summary

In this chapter, we talked about Exploratory Data Analysis, or EDA for short. We discussed how to do EDA in Java, which included creating summaries and simple visualizations.

Throughout the chapter, we used our search engine example and analyzed the data we collected previously. Our analysis showed that the distribution of some variables looks different for URLs coming from different pages of the search engine results. This suggests that it is possible to use these differences to build a model that will predict whether a URL comes from the first page or not.

In the next chapter, we will look at how to do it and discuss of supervised machine learning algorithms, such as classification and regression.

4
Supervised Learning - Classification and Regression

In previous chapters, we looked at how to pre-process data in Java and how to do Exploratory Data Analysis. Now, as we covered the foundation, we are ready to start creating machine learning models.

First, we start with supervised learning. In the supervised settings, we have some information attached to each observation, called labels, and we want to learn from it, and predict it for observations without labels.

There are two types of labels: the first are discrete and finite, such as true/false or buy/sell, and the second are continuous, such as salary or temperature. These types correspond to two types of supervised learning: classification and regression. We will talk about them in this chapter.

This chapter covers the following points:

- Classification problems
- Regression problems
- Evaluation metrics for each type
- An overview of the available implementations in Java

By the end of this chapter, you will know how to use Smile, LIBLINEAR, and other Java libraries for building supervised machine learning models.

Classification

In machine learning, the classification problems deal with discrete targets with a finite set of possible values. What this means is that there is a set of possible outcomes, and given some features we want to predict the outcome.

The **binary classification** is the most common type of classification problem, as the `target` variable can have only two possible values, such as `True`/`False`, `Relevant`/`Not Relevant`, `Duplicate`/`Not Duplicate`, `Cat`/`Dog`, and so on.

Sometimes the target variable can have more than two outcomes, for example, colors, category of an item, model of a car, and so on, and we call this **multi-class classification**. Typically, each observation can only have one label, but in some settings an observation can be assigned several values. Multi-class classification can be converted to a set of binary classification problems, which is why we will mostly concentrate on binary classification.

Binary classification models

As we have already discussed, the binary classification model deals with the case when there are only two possible outcomes that we want to predict. Typically, in these settings, we have items of the positive class (the presence of some effect) and items of the negative class (the absence of some effect).

For example, the positive label can be relevant, duplicate, fail to pay the debts, and so on. The instances of the positive class are typically assigned the target value of 1. Also, we have negative instances, such as not relevant, not duplicate, pays the debts, and they are assigned the target value of `0`.

This separation into positive and negative classes is somewhat artificial, and in some cases does not really make sense. For example, if we have images of cats and dogs, even though there are just two classes, it would be a stretch to say that `Cat` is a positive class and `Dog` is negative. But it is not important for the model, so we can still assign the labels in such a way that `Cat` is `1`, and `Dog` is `0`.

Once we have trained a model, we are not typically interested in a hard prediction such as *the positive effect is there*, or *this is a cat*. What is more interesting is the degree of the positive or negative effect, and this is typically achieved by predicting probabilities. For example, if we want to build a model to predict whether a client will fail to pay the debts, then saying *this client has 30% of defaulting* is more useful than *this client will not default*.

There are many models solve the binary classification problem and it is not possible to cover all of them. We will briefly cover the ones that are most often used in practice. They include the following:

- Logistic regression
- Support Vector Machines
- Decision trees
- Neural networks

We assume that you are already familiar with these methods, and at least have some idea as to how they work. Deep familiarity is not required, but for more information, you can check the following books:

- *An Introduction to Statistical Learning*, *G. James*, *D. Witten*, *T. Hastie*, and *R. Tibshirani*, *Springer*
- *Python Machine Learning*, *S. Raschka*, *Packt Publishing*

When it comes to libraries, we will cover Smile, JSAT, LIBSVM, LIBLINEAR, and Encog. Let's start with Smile.

Smile

Statistical Machine Intelligence and Learning Engine (**Smile**) is a library with a large set of classification and other machine learning algorithms. For us, the most interesting ones are logistic regression, SVM, and random forest, but you can see the full list of available algorithms on their official GitHub page at `https://github.com/haifengl/smile`.

The library is available on Maven Central and the latest version at the time of writing is 1.1.0. To include it to your project, add the following dependency:

```
<dependency>
  <groupId>com.github.haifengl</groupId>
  <artifactId>smile-core</artifactId>
  <version>1.1.0</version>
</dependency>
```

It is being actively developed; new features and bug fixes are added quite often, but not released as frequently. We recommend using the latest available version of Smile, and to get it, you will need to build it from the sources. To do this:

- Install `sbt`, which is a tool used for building scala projects. You can follow the instruction at `http://www.scala-sbt.org/release/docs/Manual-Installation.html`
- Use git to clone the project from `https://github.com/haifengl/smile`
- To build and publish the library to a local Maven repository, run the following command:

```
sbt core/publishM2
```

The Smile library consists of several submodules, such as `smile-core`, `smile-nlp`, `smile-plot`, and so on. For the purposes of this chapter, we only need the core package, and the preceding command will build only that. At the moment of writing, the current version available on GitHub is 1.2.0. So, after building it, add the following dependency to your pom:

```
<dependency>
  <groupId>com.github.haifengl</groupId>
  <artifactId>smile-core</artifactId>
  <version>1.2.0</version>
</dependency>
```

The models from Smile expect the data to be in a form of two-dimensional arrays of doubles, and the label information as a one dimensional array of integers. For binary models, the values should be `0` or `1`. Some models in Smile can handle multi-class classification problems, so it is possible to have more labels, not just `0` and `1`, but also `2`, `3`, and so on.

In Smile, the models are built using the `builder` pattern; you create a special class, set some parameters and at the end it returns the object it builds. This `builder` class is typically called `Trainer`, and all models should have a `Trainer` object for them.

For example, consider training a RandomForest model:

```
double[] X = ... // training data
int[] y = ... // 0 and 1 labels
RandomForest model = new RandomForest.Trainer()
    .setNumTrees(100)
    .setNodeSize(4)
    .setSamplingRates(0.7)
    .setSplitRule(SplitRule.ENTROPY)
    .setNumRandomFeatures(3)
```

```
        .train(X, y);
```

The `RandomForest.Trainer` class takes in a set of parameters and the training data, and in the end produces the trained `RandomForest` model. The implementation of RandomForest from Smile has the following parameters:

- `numTrees`: This is the number of trees to train in the model
- `nodeSize`: This is the minimum number of items in the leaf nodes
- `samplingRate`: This is the ratio of training data used to grow each tree
- `splitRule`: This is the impurity measure used for selecting the best split
- `numRandomFeatures`: This is the number of features the model randomly chooses for selecting the best split

Similarly, a logistic regression is trained as follows:

```
LogisticRegression lr = new LogisticRegression.Trainer()
        .setRegularizationFactor(lambda)
        .train(X, y);
```

Once we have a model, we can use it for predicting the label of previously unseen items. For that we use the `predict` method:

```
double[] row = // data
int prediction = model.predict(row);
```

This code outputs the most probable class for the given item. However, we are often more interested not in the label itself, but in the probability of having the label. If a model implements the `SoftClassifier` interface, then it is possible to get these probabilities like this:

```
double[] probs = new double[2];
model.predict(row, probs);
```

After running this code, the `probs` array will contain the probabilities.

JSAT

Java Statistical Analysis Tool (**JSAT**) is another Java library which contains a lot of implementations of commonly-used machine learning algorithms. You can check the full list of implemented models at `https://github.com/EdwardRaff/JSAT/wiki/Algorithms`.

To include JSAT to a Java project, add the following snippet to `pom`:

```
<dependency>
  <groupId>com.edwardraff</groupId>
  <artifactId>JSAT</artifactId>
  <version>0.0.5</version>
</dependency>
```

Unlike Smile models, which require just an array of doubles with the feature information, JSAT requires a special wrapper class for data. If we have an array, it is converted to the JSAT representation like this:

```
double[][] X = ... // data
int[] y = ... // labels

// change to more classes for more classes for multi-classification
CategoricalData binary = new CategoricalData(2);

List<DataPointPair<Integer>> data = new ArrayList<>(X.length);
for (int i = 0; i < X.length; i++) {
    int target = y[i];
    DataPoint row = new DataPoint(new DenseVector(X[i]));
    data.add(new DataPointPair<Integer>(row, target));
}

ClassificationDataSet dataset = new ClassificationDataSet(data, binary);
```

Once we have prepared the dataset, we can train a model. Let's consider the Random Forest classifier again:

```
RandomForest model = new RandomForest();
model.setFeatureSamples(4);
model.setMaxForestSize(150);
model.trainC(dataset);
```

First, we set some parameters for the model, and then, in the end, we call the `trainC` method (which means train a classifier).

In the JSAT implementation, Random Forest has fewer options for tuning than Smile, only the number of features to select and the number of trees to grow.

Also, JSAT contains several implementations of Logistic Regression. The usual Logistic Regression model does not have any parameters, and it is trained like this:

```
LogisticRegression model = new LogisticRegression();
model.trainC(dataset);
```

If we want to have a regularized model, then we need to use the `LogisticRegressionDCD` class. **Dual Coordinate Descent** (**DCD**) is the optimization method used to train the logistic regression). We train it like this:

```
LogisticRegressionDCD model = new LogisticRegressionDCD();
model.setMaxIterations(maxIterations);
model.setC(C);
model.trainC(fold.toJsatDataset());
```

In this code, `C` is the regularization parameter, and the smaller values of `C` correspond to stronger regularization effect.

Finally, for outputting probabilities, we can do the following:

```
double[] row = // data
DenseVector vector = new DenseVector(row);
DataPoint point = new DataPoint(vector);
CategoricalResults out = model.classify(point);
double probability = out.getProb(1);
```

The `CategoricalResults` class contains a lot of information, including probabilities for each class and the most likely label.

LIBSVM and LIBLINEAR

Next, we consider two similar libraries, LIBSVM and LIBLINEAR.

- LIBSVM (`https://www.csie.ntu.edu.tw/~cjlin/libsvm/`) is a library with implementation of Support Vector Machine models, which include support vector classifiers
- LIBLINEAR (`https://www.csie.ntu.edu.tw/~cjlin/liblinear/`) is a library for fast linear classification algorithms such as Liner SVM and Logistic Regression

Both these libraries come from the same research group and have very similar interfaces. We will start with LIBSVM.

LIBSVM is a library that implements a number of different SVM algorithms. It is implemented in C++ and has an officially supported Java version. It is available on Maven Central:

```
<dependency>
  <groupId>tw.edu.ntu.csie</groupId>
  <artifactId>libsvm</artifactId>
  <version>3.17</version>
</dependency>
```



Note that the Java version of LIBSVM is updated not as often as the C++ version. Nevertheless, the preceding version is stable and should not contain bugs, but it might be slower than its C++ version.

To use SVM models from LIBSVM, you first need to specify the parameters. For this, you create a `svm_parameter` class. Inside, you can specify many parameters, including:

- The kernel type (`RBF`, `POLY`, or `LINEAR`)
- The regularization parameter `C`
- `probability` which you can set to `1` to be able to get probabilities
- `svm_type` should be set to `C_SVC`; this tells that the model should be a classifier

Recall that SVM models can have different kernels, and depending on which kernel we use, we have different models with different parameters. Here, we will consider the most commonly used kernels; linear (or no kernel), polynomial and **Radial Basis Function** (**RBF**, also known as Gaussian kernel).
First, let's start with the linear kernel. First, we create an `svm_paramter` object, where we set the kernel type to `LINEAR` and also ask it to output probabilities:

```
svm_parameter param = new svm_parameter();
param.svm_type = svm_parameter.C_SVC;
param.kernel_type = svm_parameter.LINEAR;
param.probability = 1;
param.C = C;

// default parameters
param.cache_size = 100;
param.eps = 1e-3;
param.p = 0.1;
param.shrinking = 1;
```

Next, we have a polynomial kernel. Recall that the polynomial kernel is specified by the following formula:

$$(\mathbf{gamma} \cdot u^T v + \mathbf{coef0})^{\mathbf{degree}}$$

It has three additional parameters, that is, `gamma`, `coef0`, and `degree`, which control the kernel, and also `C`--the regularization parameter. We can configure the `svm_parameter` class for `POLY` SVM like this:

```
svm_parameter param = new svm_parameter();
```

```
param.svm_type = svm_parameter.C_SVC;
param.kernel_type = svm_parameter.POLY;
param.C = C;
param.degree = degree;
param.gamma = 1;
param.coef0 = 1;
param.probability = 1;
// plus defaults from the above
```

Finally, the Gaussian kernel (or RBF) has the following formula:

$$\exp(-\text{gamma} \cdot \| u\text{-}v \|^2)$$

So there is one parameter, `gamma`, which controls the width of the Gaussians. We can specify the model with the `RBF` kernel like this:

```
svm_parameter param = new svm_parameter();
param.svm_type = svm_parameter.C_SVC;
param.kernel_type = svm_parameter.RBF;
param.C = C;
param.gamma = gamma;
param.probability = 1;
// plus defaults from the above
```

Once we have created the configuration object, we need to convert the data in the right format. The library expects the data to be represented in the sparse format. For a single data row, the conversion to the required format is as follows:

```
double[] dataRow = // single row vector
svm_node[] svmRow = new svm_node[dataRow.length];

for (int j = 0; j < dataRow.length; j++) {
    svm_node node = new svm_node();
    node.index = j;
    node.value = dataRow[j];
    svmRow[j] = node;
}
```

Since we typically have a matrix, not just a single row, we apply the preceding code to each row of this matrix:

```
double[][] X = ... // data
int n = X.length;
svm_node[][] nodes = new svm_node[n][];

for (int i = 0; i < n; i++) {
```

```
        nodes[i] = wrapAsSvmNode(X[i]);
    }
```

Here, `wrapAsSvmNode` is a function, that wraps a vector into an array of `svm_node` objects.

Now, we can put the data and the labels together into the `svm_problem` object:

```
double[] y = ... // labels
svm_problem prob = new svm_problem();
prob.l = n;
prob.x = nodes;
prob.y = y;
```

Finally, we can use the parameters and the problem specification to train an SVM model:

```
svm_model model = svm.svm_train(prob, param);
```

Once the model is trained, we can use it to classify unseen data. Getting probabilities is done this way:

```
double[][] X = // test data
int n = X.length;
double[] results = new double[n];
double[] probs = new double[2];

for (int i = 0; i < n; i++) {
    svm_node[] row = wrapAsSvmNode(X[i]);
    svm.svm_predict_probability(model, row, probs);
    results[i] = probs[1];
}
```

Since we used `param.probability = 1`, we can use the `svm.svm_predict_probability` method to predict probabilities. Like in Smile, the method takes an array of doubles and writes the output there. After this operation, it will contain the probabilities in this array.

Finally, while training, LIBSVM outputs a lot of things on the console. If we are not interested in this output, we can disable it with the following code snippet:

```
svm.svm_set_print_string_function(s -> {});
```



Just add this in the beginning of your code and you will not see the debugging information anymore.

The next library is LIBLINEAR, which provides very fast and high-performing linear classifiers, such as SVM with linear kernel and logistic regression. It can easily scale to tens and hundreds of millions of data points. Its interface is quite similar to LIBSVM, where we need to specify the parameters and the data, and then train a model.

Unlike LIBSVM, there is no official Java version of LIBLINEAR, but there is an unofficial Java port available at `http://liblinear.bwaldvogel.de/`. To use it, include the following:

```
<dependency>
 <groupId>de.bwaldvogel</groupId>
 <artifactId>liblinear</artifactId>
 <version>1.95</version>
</dependency>
```

The interface is very similar to LIBSVM. First, we define the parameters:

```
SolverType solverType = SolverType.L1R_LR;
double C = 0.001;
double eps = 0.0001;
Parameter param = new Parameter(solverType, C, eps);
```

In this example, we specify three parameters:

- `solverType`: This defines the model that will be used
- `C`: This is the amount of regularization, the smaller the C, the stronger the regularization
- `epsilon`: This is the level of tolerance for stopping the training process; a reasonable default is `0.0001`

For the classification problem, the following are the solvers that we can use:

- **Logistic regression**: `L1R_LR` or `L2R_LR`
- **SVM**: `L1R_L2LOSS_SVC` or `L2R_L2LOSS_SVC`

Here, we have two models; logistic regression and SVM, and two regularization types, L1 and L2. How can we decide which model to choose and which regularization to use? According to the official FAQ (which can be found here: `https://www.csie.ntu.edu.tw/~cjlin/liblinear/FAQ.html`), we should:

- Prefer SVM to logistic regression as it trains faster and usually gives higher accuracy
- Try L2 regularization first unless you need a sparse solution, in this case use L1

Next, we need to prepare our data. As previously, we need to wrap it to some special format. First, let's see how to wrap a single data row:

```
double[] row = // data
int m = row.length;
Feature[] result = new Feature[m];

for (int i = 0; i < m; i++) {
    result[i] = new FeatureNode(i + 1, row[i]);
}
```



Note that we add `1` to the index. The `0` is the bias term, so the actual features should start from `1`.

We can put this code into a `wrapRow` function and then wrap the entire dataset as follows:

```
double[][] X = // data
int n = X.length;
Feature[][] matrix = new Feature[n][];
for (int i = 0; i < n; i++) {
    matrix[i] = wrapRow(X[i]);
}
```

Now, we can create the `Problem` class with the data and labels:

```
double[] y = // labels

Problem problem = new Problem();
problem.x = wrapMatrix(X);
problem.y = y;
problem.n = X[0].length + 1;
problem.l = X.length;
```

Note that here we also need to provide the dimensionality of the data, and it's the number of features plus one. We need to add one because it includes the bias term.

Now we are ready to train the model:

```
Model model = LibLinear.train(fold, param);
```

When the model is trained, we can use it to classify unseen data. In the following example, we will output probabilities:

```
double[] dataRow = // data
Feature[] row = wrapRow(dataRow);
Linear.predictProbability(model, row, probs);
```

```
double result = probs[1];
```

The preceding code works fine for the logistic regression model, but it will not work for SVM, SVM cannot output probabilities, so the preceding code will throw an error for solvers such as `L1R_L2LOSS_SVC`. What we can do instead is to get the raw output:

```
double[] values = new double[1];
Feature[] row = wrapRow(dataRow);
Linear.predictValues(model, row, values);
double result = values[0];
```

In this case, the results will not contain probability, but some real value. When this value is greater than zero, the model predicts that the class is positive.

If we would like to map this value to the `[0, 1]` range, we can use the `sigmoid` function for that:

```
public static double[] sigmoid(double[] scores) {
    double[] result = new double[scores.length];

    for (int i = 0; i < result.length; i++) {
        result[i] = 1 / (1 + Math.exp(-scores[i]));
    }

    return result;
}
```

Finally, like LIBSVM, LIBLINEAR also outputs a lot of things to standard output. If you do not wish to see it, you can mute it with the following code:

```
PrintStream devNull = new PrintStream(new NullOutputStream());
Linear.setDebugOutput(devNull);
```

Here, we use `NullOutputStream` from Apache IO, which does nothing, so the screen stays clean.

Want to know when to use LIBSVM and when to use LIBLINEAR? For large datasets, it is not often possible to use any kernel methods. In this case, you should prefer LIBLINEAR. Additionally, LIBLINEAR is especially good for text processing purposes such as document classification. We will cover these cases in more detail in `Chapter 6`, *Working with Texts - Natural Language Processing and Information Retrieval*.

Encog

So far, we have covered many models, that is, logistic regression, SVM, and RandomForest, and we have looked at multiple libraries that implement them. But we have not yet covered neural networks. In Java, there is a special library that deals exclusively with neural networks--Encog. It is available on Maven Central and can be added with the following snippet:

```
<dependency>
  <groupId>org.encog</groupId>
  <artifactId>encog-core</artifactId>
  <version>3.3.0</version>
</dependency>
```

After including the library, the first step is to specify the architecture of a neural network. We can do it like this:

```
BasicNetwork network = new BasicNetwork();
network.addLayer(new BasicLayer(new ActivationSigmoid(), true,
noInputNeurons));
network.addLayer(new BasicLayer(new ActivationSigmoid(), true, 30));
network.addLayer(new BasicLayer(new ActivationSigmoid(), true, 1));
network.getStructure().finalizeStructure();
network.reset();
```

Here, we create a network with one input layer, one inner layer with 30 neurons, and one output layer with 1 neuron. In each layer we use sigmoid as the activation function and add the bias input (the `true` parameter). Finally, the `reset` method randomly initializes the weights in the network.

For both input and output, Encog expects two-dimensional double arrays. In the case of binary classification, we typically have a one dimensional array, so we need to convert it:

```
double[][] X = // data
double[] y = // labels
double[][] y2d = new double[y.length][];

for (int i = 0; i < y.length; i++) {
    y2d[i] = new double[] { y[i] };
}
```

Once the data is converted, we wrap it into a special wrapper class:

```
MLDataSet dataset = new BasicMLDataSet(X, y2d);
```

Then, this dataset can be used for training:

```
MLTrain trainer = new ResilientPropagation(network, dataset);
double lambda = 0.01;
trainer.addStrategy(new RegularizationStrategy(lambda));

int noEpochs = 101;
for (int i = 0; i < noEpochs; i++) {
    trainer.iteration();
}
```

We won't cover Encog in much detail here, but we will come back to neural networks in `Chapter 8`, *Deep Learning with DeepLearning4j*, where we will look at a different library--DeepLearning4J.

There are a lot of other machine learning libraries available in Java. For example Weka, H2O, JavaML, and others. It is not possible to cover all of them, but you can also try them and see if you like them more than the ones we have covered.

Next, we will see how we can evaluate the classification models.

Evaluation

We have covered many machine learning libraries, and many of them implement the same algorithms such as random forest or logistic regression. Also, each individual model can have many different parameters, a logistic regression has the regularization coefficient, an SVM is configured by setting the kernel and its parameters.

How do we select the best single model out of so many possible variants?

For that, we first define some evaluation metric and then select the model which achieves the best possible performance with respect to this metric. For binary classification, there are many metrics that we can use for comparison, and the most commonly used ones are as follows:

- Accuracy and error
- Precision, recall, and F1
- AUC (AU ROC)

We use these metrics to see how well the model will be able to generalize to new unseen data. Therefore, it is important to model this situation when the data is new to the model. This is typically done by splitting the data into several parts. So, we will also cover the following:

- Result evaluation
- K-fold cross-validation
- Training, validation, and testing

Let us start with the most intuitive evaluation metric, accuracy.

Accuracy

Accuracy is the most straightforward way of evaluating a classifier: we make a prediction, look at the predicted label and then compare it with the actual value. If the values agree, then the model got it right. Then, we can do it for all the data that we have and see the ratio of the correctly predicted examples; and this is exactly what accuracy describes. So, accuracy tells us for how many examples the model predicted the correct label. Calculating it is trivial:

```
int n = actual.length;
double[] proba = // predictions;

double[] prediction = Arrays.stream(proba).map(p -> p > threshold ? 1.0 :
0.0).toArray();
int correct = 0;

for (int i = 0; i < n; i++) {
    if (actual[i] == prediction[i]) {
        correct++;
    }
}

double accuracy = 1.0 * correct / n;
```

Accuracy is the simplest evaluation metric, and it is easy to explain it to anybody, even to nontechnical people.

However sometimes, accuracy is not the best measure of model performance. Next we will see what are its problems and what to use instead.

Precision, recall, and F1

In some cases, accuracy values are deceiving: they suggest that the classifier is good, although it is not. For example, suppose we have an unbalanced dataset: there are only 1% of examples that are positive, and the rest (99%) are negative. Then, a model that always predicts negative is right in 99% of the cases, and hence will have an accuracy of 0.99. But this model is not useful.

There are alternatives to accuracy that can overcome this problem. Precision and recall are among these metrics, as they both look at the fraction of positive items that the model correctly recognized. So, if we have a large number of negative examples, we can still perform some meaningful evaluation of the model.

Precision and recall can be calculated using the confusion matrix, a table which summarizes the performance of a binary classifier:

		Actual Class y	
		Positive	Negative
$h_\theta(x)$ **Predicted outcome**	Predicted positive outcome	**True positive** (TP)	**False positive** (FP)
	Predicted negative outcome	**False negative** (FN)	**True negative** (TN)

When we use a binary classification model to predict the actual value of some data item, there are four possible outcomes:

- **True positive** (**TP**): The actual class is positive, and we predict positive
- **True negative** (**TN**): The actual class is negative, and we predict negative
- **False positive** (**FP**): The actual class is negative, but we say it is positive
- **False negative** (**FN**): The actual class is positive, but we say it is negative

The first two cases (TP and TN) are correct predictions the actual and the predicted values are the same. The last two cases (FP and FN) are incorrect classification, as we fail to predict the correct label.

Now, suppose that we have a dataset with known labels, and run our model against it. Then, let `TP` be the number of true positive examples, `TN` the true negative examples, and so on.

Then we can calculate precision and recall using these values:

- Precision is the fraction of correctly--predicted positive items among all items the model predicted positive. In terms of the confusion matrix, precision is `TP / (TP + FP)`.
- Recall is the fraction of correctly-- predicted positive items among items that are actually positive. With values from the confusion matrix, recall is `TP / (TP + FN)`.
- It is often hard to decide whether one should optimize precision or recall. But there is another metric which combines both precision and recall into one number, and it is called **F1 score**.

For calculating precision and recall, we first need to calculate the values for the cells of the confusion matrix:

```
int tp = 0, tn = 0, fp = 0, fn = 0;

for (int i = 0; i < actual.length; i++) {
    if (actual[i] == 1.0 && proba[i] > threshold) {
        tp++;
    } else if (actual[i] == 0.0 && proba[i] <= threshold) {
        tn++;
    } else if (actual[i] == 0.0 && proba[i] > threshold) {
        fp++;
    } else if (actual[i] == 1.0 && proba[i] <= threshold) {
        fn++;
    }
}
```

Then, we can use the values to calculate precision and recall:

```
double precision = 1.0 * tp / (tp + fp);
double recall = 1.0 * tp / (tp + fn);
```

Finally, `f1` can be calculated using the following formula:

```
double f1 = 2 * precision * recall / (precision + recall);
```

These metrics are quite useful when the dataset is imbalanced.

ROC and AU ROC (AUC)

The preceding metrics are good for binary classifiers which produce a hard output; they only tell whether the class should be assigned a positive label or negative. If, instead, our model outputs some score such that the higher the values of the score the more likely the item is to be positive, then the binary classifier is called a **ranking classifier**.

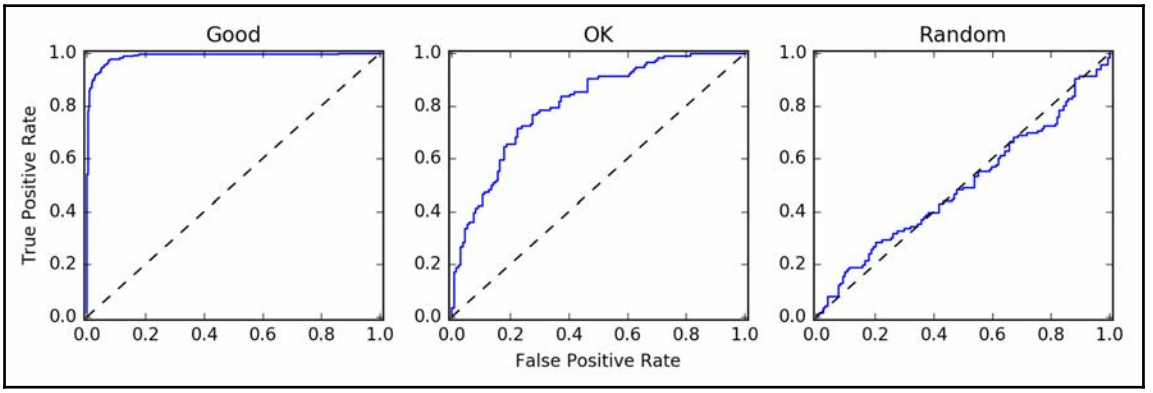
Most of the models can output probabilities of belonging to a certain class, and we can use it to rank examples such that the positives are likely to come first.

The ROC curve visually tells us how good a ranking classifier separates positive examples from negative ones. The way a ROC curve is built is as follows:

- Sort the observations by their score and then start from the origin
- Go up if the observation is positive and right if it is negative.

This way, in the ideal case, we first always go up, and then always go right and this will result in the best possible ROC curve. In this case, we can say that the separation between positive and negative examples is perfect. If the separation is not perfect, but still **OK**, the curve will go up for positive examples, but sometimes will turn right when a misclassification occurs. Finally, a bad classifier will not be able to tell positive and negative examples apart and the curve would alternate between going up and right.

Let's look at some examples:



The diagonal line on the plot represents the baseline--the performance that a random classifier would achieve. The further away the curve is from the baseline, the better.

Unfortunately, there is no available easy-to-use implementation of ROC curves in Java. It is not hard to implement the code ourselves. Here, we will outline how to do it, and you will find the implementation in the chapter code repository.

So the algorithm for drawing a ROC curve is as follows:

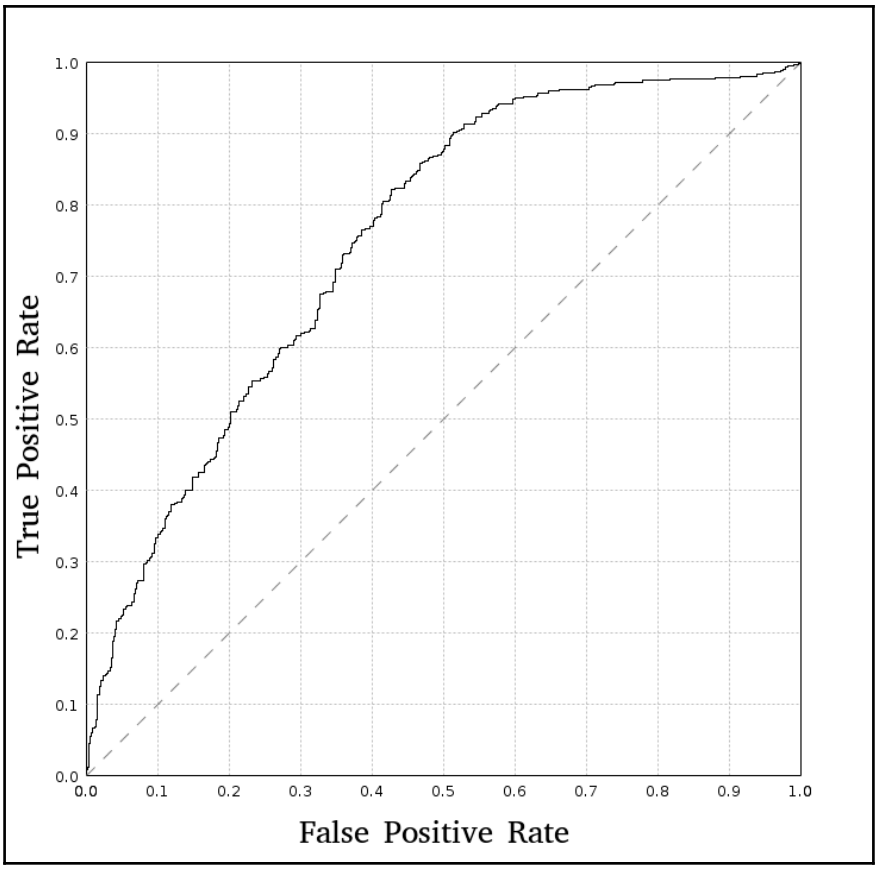
- Let POS be the number of positive labels, and NEG be the number of negative labels
- Order data by the score, decreasing
- Start from (0, 0)
- For each example in the sorted order,
    - o if the example is positive, move 1 / POS up in the graph,
    - o otherwise, move 1 / NEG right in the graph.

This is a simplified algorithm and assumes that the scores are distinct. If the scores aren't distinct, and there are different actual labels for the same score, some adjustment needs to be made.

It is implemented in the `RocCurve` class, which you will find in the source code. You can use it as follows:

```
RocCurve.plot(actual, prediction);
```

Calling it will create a plot similar to this one:



The area under the curve says how good the separation between the positive and negative examples is. If the separation is very good, then the area will be close to one. But if the classifier cannot distinguish between positive and negative examples, the curve will go around the random baseline curve, and the area will be close to **0.5**.

The area under the curve is often abbreviated as AUC, or, sometimes, AU ROC to emphasize that the curve is a ROC curve.

AUC has a very nice interpretation--the value of AUC corresponds to probability that a randomly selected positive example is scored higher than a randomly selected negative example. Naturally, if this probability is high, our classifier does a good job separating positive and negative examples.

This makes AUC a to-go evaluation metric for many cases, especially when the dataset is unbalanced in the sense that there are a lot more examples of one class than another.

Luckily, there are implementations of AUC in Java. For example, it is implemented in Smile. You can use it like this:

```
double[] predicted = ...  //
int[] truth = ... //
double auc = AUC.measure(truth, predicted);
```

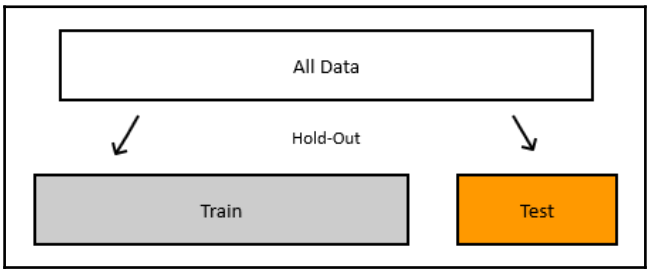
Now, when we discussed the possible evaluation metrics, we need to apply them to test our models. We need to handle it with care. If we perform the evaluation on the same data which we used for training, then the evaluation results will be overly optimistic. Next, we will see what is the correct way of doing it.

Result validation

When learning from data, there is always the danger of overfitting. Overfitting occurs when the model starts learning the noise in the data instead of detecting useful patterns. It is always important to check if a model overfits, otherwise it will not be useful when applied to unseen data.

The typical and most practical way of checking whether a model overfits or not is to emulate unseen data, that is, take a part of the available labeled data and do not use it for training.

This technique is called **hold out**, where we hold out a part of the data and use it only for evaluation.



We also shuffle the original dataset before splitting. In many cases, we make a simplifying assumption that the order of data is not important, that is, one observation has no influence on another. In this case, shuffling the data prior to splitting will remove effects that the order of items might have. On the other hand, if the data is a time series data, then shuffling it is not a good idea, because there is some dependency between observations.

So, let's implement the hold out split. We assume that the data that we have is already represented by `X`--a two-dimensional array of doubles with features and `y`--a one-dimensional array of labels.

First, we create a helper class for holding the data:

```
public class Dataset {
    private final double[][] X;
    private final double[] y;
    // constructor and getters are omitted
}
```

Splitting our dataset should produce two datasets, so let's create a class for that as well:

```
public class Split {
    private final Dataset train;
    private final Dataset test;
    // constructor and getters are omitted
}
```

Now, suppose we want to split the data into two parts: train and test. We also want to specify the size of the train set, and we will do it using a `testRatio` parameter: the percentage of items that should go to the test set.

The first thing we do is to generate an array with indexes and then split it according to `testRatio`:

```
int[] indexes = IntStream.range(0, dataset.length()).toArray();
int trainSize = (int) (indexes.length * (1 - testRatio));
int[] trainIndex = Arrays.copyOfRange(indexes, 0, trainSize);
int[] testIndex = Arrays.copyOfRange(indexes, trainSize, indexes.length);
```

We can also shuffle the indexes if we want:

```
Random rnd = new Random(seed);

for (int i = indexes.length - 1; i > 0; i--) {
    int index = rnd.nextInt(i + 1);
    int tmp = indexes[index];
    indexes[index] = indexes[i];
    indexes[i] = tmp;
}
```

Then we can select instances for the training set as follows:

```
int trainSize = trainIndex.length;
double[][] trainX = new double[trainSize][];
double[] trainY = new double[trainSize];
for (int i = 0; i < trainSize; i++) {
    int idx = trainIndex[i];
    trainX[i] = X[idx];
    trainY[i] = y[idx];
}
```

And then, finally, wrap it into our `Dataset` class:

```
Dataset train = new Dataset(trainX, trainY);
```

If we repeat the same for the test set, we can put both train and test sets into a `Split` object:

```
Split split = new Split(train, test);
```

Now we can use train fold for training and test fold for testing the models.

If we put all the previous code into a function of the `Dataset` class, for example, `trainTestSplit`, we can use it as follows:

```
Split split = dataset.trainTestSplit(0.2);

Dataset train = split.getTrain();
// train the model using train.getX() and train.getY()

Dataset test = split.getTest();
// test the model using test.getX(); test.getY();
```

Here, we train a model on the `train` dataset, and then calculate the evaluation metric on the `test` set.

K-fold cross-validation

Holding out only one part of the data may not always be the best option. What we can do instead is splitting it into *K* parts and then testing the models only on *1/K*th of the data.

This is called **k-fold cross-validation**; it not only gives the performance estimation, but also the possible spread of the error. Typically, we are interested in models that give good and consistent performance. K-fold cross-validation helps us to select such models.

Next we prepare the data for k-fold cross-validation as follows:

- First, split the data into *K* parts
- Then, for each of these parts:
    - Take one part as the validation set
    - Take the remaining *K-1* parts as the training set

If we translate this into Java, the first step will look like this:

```
int[] indexes = IntStream.range(0, dataset.length()).toArray();
int[][] foldIndexes = new int[k][];

int step = indexes.length / k;
int beginIndex = 0;

for (int i = 0; i < k - 1; i++) {
    foldIndexes[i] = Arrays.copyOfRange(indexes, beginIndex, beginIndex +
step);
    beginIndex = beginIndex + step;
}

foldIndexes[k - 1] = Arrays.copyOfRange(indexes, beginIndex,
indexes.length);
```

This creates an array of indexes for each of the *K* folds. We can also shuffle the indexes array as previously.

Now we can create splits from each fold:

```
List<Split> result = new ArrayList<>();

for (int i = 0; i < k; i++) {
    int[] testIdx = folds[i];
    int[] trainIdx = combineTrainFolds(folds, indexes.length, i);
    result.add(Split.fromIndexes(dataset, trainIdx, testIdx));
}
```

In the preceding code we have two additional methods:

- `combineTrainFolds`: This takes in *K-1* arrays with indexes and combines them into one
- `Split.fromIndexes`: This creates a split that trains and tests indexes.

We have already covered the second function when we created a simple hold-out test set.

The first function, `combineTrainFolds`, is implemented like this:

```
private static int[] combineTrainFolds(int[][] folds, int totalSize, int
excludeIndex) {
    int size = totalSize - folds[excludeIndex].length;
    int result[] = new int[size];

    int start = 0;
    for (int i = 0; i < folds.length; i++) {
        if (i == excludeIndex) {
            continue;
        }
        int[] fold = folds[i];
        System.arraycopy(fold, 0, result, start, fold.length);
        start = start + fold.length;
    }

    return result;
}
```

Again, we can put the preceding code into a function of the `Dataset` class and call it like follows:

```
List<Split> folds = train.kfold(3);
```

Now, when we have a list of `Split` objects, we can create a special function for performing cross-validation:

```
public static DescriptiveStatistics crossValidate(List<Split> folds,
        Function<Dataset, Model> trainer) {
    double[] aucs = folds.parallelStream().mapToDouble(fold -> {
        Dataset foldTrain = fold.getTrain();
        Dataset foldValidation = fold.getTest();
        Model model = trainer.apply(foldTrain);
        return auc(model, foldValidation);
    }).toArray();

    return new DescriptiveStatistics(aucs);
}
```

What this function does is, it takes a list of folds and a callback that is inside and creates a model. After the model is trained, we calculate AUC for it.

Additionally, we take advantage of Java's ability to parallelize loops and train models on each fold at the same time.

Finally, we put the AUCs calculated on each fold into a `DescriptiveStatistics` object, which can later on be used to return the mean and standard deviation of the AUCs. As you probably remember, the `DescriptiveStatistics` class comes from the Apache Commons Math library.

Let's consider an example. Suppose we want to use logistic regression from `LIBLINEAR` and select the best value for the regularization parameter, `C`. We can use the preceding function this way:

```
double[] Cs = { 0.01, 0.05, 0.1, 0.5, 1.0, 5.0, 10.0 };

for (double C : Cs) {
    DescriptiveStatistics summary = crossValidate(folds, fold -> {
        Parameter param = new Parameter(SolverType.L1R_LR, C, 0.0001);
        return LibLinear.train(fold, param);
    });

    double mean = summary.getMean();
    double std = summary.getStandardDeviation();
    System.out.printf("L1 logreg C=%7.3f, auc=%.4f &pm; %.4f%n", C, mean,
std);
}
```

Here, `LibLinear.train` is a helper method that takes a `Dataset` object and a `Parameter` object and then trains a LIBLINEAR model. This will print AUC for all the provided values of `C`, so you can see which one is the best, and pick the one with the highest mean AUC.

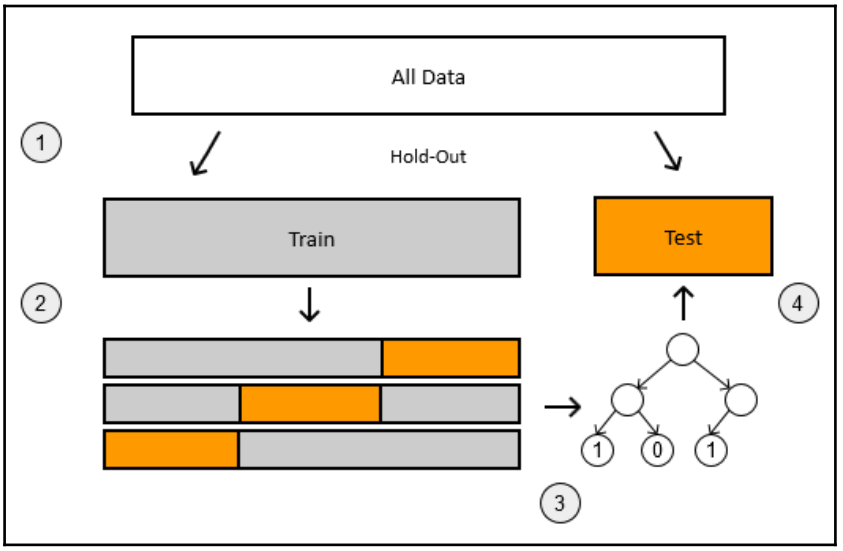
Training, validation, and testing

When doing cross-validation, there's still a danger of overfitting. Since we try a lot of different experiments on the same validation set, we might accidentally pick the model which just happened to do well on the validation set--but it may later on fail to generalize to unseen data.

The solution to this problem is to hold out a test set at the very beginning and do not touch it at all until we select what we think is the best model. And we use it only for evaluating the final model on it.

So how do we select the best model? What we can do is to do cross-validation on the remaining train data. It can be hold out or k-fold cross-validation. In general, you should prefer doing k-fold cross-validation because it also gives you the spread of performance, and you may use it in for model selection as well.

The following diagram illustrates the process:



According to the diagram, a typical data science workflow should be the following:

- **0**: Select some metric for validation, for example, accuracy or AUC
- **1**: Split all the data into train and test sets
- **2**: Split the training data further and hold out a validation dataset or split it into k folds
- **3**: Use the validation data for model selection and parameter optimization
- **4**: Select the best model according to the validation set and evaluate it against the hold out test set

It is important to avoid looking at the test set too often, it should be used very rarely, and only for final evaluation to make sure the selected model does not overfit. If the validation scheme is set up properly, the validation score should correspond to the final test score. If this happens, we can be sure that the model does not overfit and is able to generalize to unseen data.

Using the classes and the code we created previously, it translates to the following Java code:

```
Dataset data = new Dataset(X, y);
Dataset train = split.getTrain();
List<Split> folds = train.kfold(3);
// now use crossValidate(folds, ...) to select the best model

Dataset test = split.getTest();
// do final evaluation of the best model on test
```

With this information, we are ready to do a project on binary classification.

Case study - page prediction

Now we will continue with our running example, the search engine. What we want to do here is to try to predict whether a URL comes from the first page of the search engine results or not. So, it is time to use the material we have covered so far in this chapter.

In `Chapter 2`, *Data Processing Toolbox*, we created the following object to store the information about pages:

```
public class RankedPage {
    private String url;
    private int position;
    private int page;
    private int titleLength;
    private int bodyContentLength;
    private boolean queryInTitle;
    private int numberOfHeaders;
    private int numberOfLinks;
}
```

First, we can start by adding a few methods to this object, as follows:

- `isHttps`: This should tell us if the URL is HTTPS and can be implemented with `url.startsWith("https://")`
- `isComDomain`: This should tells us if the URL belongs to the COM domain and whether we can implement it with `url.contains(".com")`
- `isOrgDomain`, `isNetDomain`: This is the same as the previous one, but for ORG and NET, respectively

- `numberOfSlashes`: This is the number of slash characters in the URL and can be implemented with Guava's `CharMatcher`: `CharMatcher.is('/').countIn(url)`

These models describe each URL that we get, so we call them feature methods, and we can use the results of these methods in our machine learning models.

As previously, we have a method which reads the JSON data and creates a Joinery DataFrame from it:

```
List<RankedPage> pages = RankedPageData.readRankedPages();
DataFrame<Object> dataframe = BeanToJoinery.convert(pages,
RankedPage.class);
```

When we have the data, the first step is to extract the values of the target variable:

```
List<Object> page = dataframe.col("page");
double[] target = page.stream()
                      .mapToInt(o -> (int) o)
                      .mapToDouble(p -> (p == 0) ? 1.0 : 0.0)
                      .toArray();
```

To get the feature matrix `X`, we can use Joinery to create a two-dimensional array for us. First, we need to drop some of the variables, namely, the target variable, the URL, and also the position, because the position clearly correlates with the page. We can do it like this:

```
dataframe = dataframe.drop("page", "url", "position");
double[][] X = dataframe.toModelMatrix(0.0);
```

Next, we can use the `Dataset` class we have created in this chapter and split it into train and test parts:

```
Dataset dataset = new Dataset(X, target);
Split split = dataset.trainTestSplit(0.2);
Dataset train = split.getTrain();
Dataset test = split.getTest();
```

Additionally, for some algorithms, it is helpful to standardize the features such that they have zero mean and unit standard deviation. The reason for doing this is to help the optimization algorithms converge faster.

To do it, we calculate mean and standard deviation for each column of the matrix, and then subtract the mean from each value and divide it by the standard deviation. We omit the code of this function here for brevity, but you can find it in the chapter code repository.

The following code does this:

```
preprocessor = StandardizationPreprocessor.train(train);
train = preprocessor.transform(train);
test = preprocessor.transform(test);
```

Now we are ready to start training different models. Let's try logistic regression implementation from Smile first. We will use k-fold cross-validation to select the best value of lambda, its regularization parameter.

```
List<Fold> folds = train.kfold(3);
double[] lambdas = { 0, 0.5, 1.0, 5.0, 10.0, 100.0, 1000.0 };
for (double lambda : lambdas) {
    DescriptiveStatistics summary = Smile.crossValidate(folds, fold -> {
        return new LogisticRegression.Trainer()
                .setRegularizationFactor(lambda)
                .train(fold.getX(), fold.getYAsInt());
    });

    double mean = summary.getMean();
    double std = summary.getStandardDeviation();
    System.out.printf("logreg, λ=%8.3f, auc=%.4f &pm; %.4f%n", lambda,
mean, std);
}
```

Note that the `Dataset` class here has a new method, `getYAsInt`, which simply returns the target variable represented as an array of integers. When we run this, it produces the following output:

```
logreg, λ=   0.000, auc=0.5823 &pm; 0.0041
logreg, λ=   0.500, auc=0.5822 &pm; 0.0040
logreg, λ=   1.000, auc=0.5820 &pm; 0.0037
logreg, λ=   5.000, auc=0.5820 &pm; 0.0030
logreg, λ=  10.000, auc=0.5823 &pm; 0.0027
logreg, λ= 100.000, auc=0.5839 &pm; 0.0009
logreg, λ=1000.000, auc=0.5859 &pm; 0.0036
```

It shows the value of lambda, the AUC we got for this value, and the standard deviation of AUCs across different folds.

We see that the AUCs that we get are quite low. This should not be a surprise: using only the information that we have now is clearly not enough to fully reverse-engineer the ranking algorithm of the search engine. In the following chapters, we will learn how to extract more information from the pages and these techniques will help to increase AUC greatly.

Another thing we can notice is that the AUCs are quite similar across different values of lambda, but one of them has the lowest standard deviation. In situations like this, we always should prefer models with smallest variance.

We can also try a more complex classifier such as RandomForest:

```
DescriptiveStatistics rf = Smile.crossValidate(folds, fold -> {
    return new RandomForest.Trainer()
            .setNumTrees(100)
            .setNodeSize(4)
            .setSamplingRates(0.7)
            .setSplitRule(SplitRule.ENTROPY)
            .setNumRandomFeatures(3)
            .train(fold.getX(), fold.getYAsInt());
});

System.out.printf("random forest auc=%.4f &pm; %.4f%n", rf.getMean(),
rf.getStandardDeviation());
```

This creates the following output:

```
random forest auc=0.6093 &pm; 0.0209
```

This classifier is, on average, 2% better than the logistic regression one, but we can also notice that the standard deviation is quite high. Because it is a lot higher, we can suspect that on the test data this model may perform significantly worse than the logistic regression model.

Next, we can try training other models as well. But, let's assume we did that and at the end we came to the conclusion that the logistic regression with `lambda=100` gives the best performance. We can then take it and retrain on the entire train dataset, and then use the test set for the final evaluation:

```
LogisticRegression logregFinal = new LogisticRegression.Trainer()
        .setRegularizationFactor(100.0)
        .train(train.getX(), train.getYAsInt());

double auc = Smile.auc(logregFinal, test);
System.out.printf("final logreg auc=%.4f%n", auc);
```

This code produces the following output:

```
final logreg auc=0.5807
```

So, indeed, we can see that the AUC produced by the selected model is the same as in our cross-validation. This is a good sign that the model can generalize well and does not overfill.

For curiosity, we can also check how the RandomForest model would perform on the training set. Since it has high variance, it may perform worse than logistic regression, but also can perform way better. Let's retrain it on the entire train set:

```
RandomForest rfFinal = new RandomForest.Trainer()
        .setNumTrees(100)
        .setNodeSize(4)
        .setSamplingRates(0.7)
        .setSplitRule(SplitRule.ENTROPY)
        .setNumRandomFeatures(3)
        .train(train.getX(), train.getYAsInt());

double auc = Smile.auc(rfFinal, test);
System.out.printf("final rf auc=%.4f%n", finalAuc);
```

It prints the following:

```
final rf auc=0.5778
```

So, indeed, high variance of the model resulted in a test score lower than the cross-validation score. This is not a good sign, and such models should not be preferred.

Thus, for such a dataset, the best performing model is logistic regression.

If you wonder how to use other machine learning libraries to solve this problem, you can check the chapter's code repository. There we created some examples for JSAT, JavaML, LIBSVM, LIBLINEAR, and Encog.

With this, we conclude the part of this chapter on classification and next we will look into another supervised learning problem called **regression**.

Regression

In machine learning, regression problems deal with situations when the label information is continuous. This can be predicting the temperature for tomorrow, the stock price, the salary of a person or the rating of an item on an e-commerce website.

There are many models which can solve the regression problem:

- **Ordinary Least Squares** (**OLS**) is the usual linear regression
- Ridge regression and LASSO are the regularized variants of OLS
- Tree-based models such as RandomForest
- Neural networks

Approaching a regression problem is very similar to approaching a classification problem, and the general framework stays the same:

- First, you select an evaluation metric
- Then, you split the data into training and testing
- You train the model on training, tune parameters using cross-validation, and do the final verification using the held out testing set.

Machine learning libraries for regression

We have already discussed many machine learning libraries that can deal with classification problems. Typically, these libraries also have regression models. Let's briefly go over them.

Smile

Smile is a general purpose machine learning library, so it has regression models as well. You can have a look at the list of models, here: `https://github.com/haifengl/smile`.

For example, this is how you can create a simple linear regression:

```
OLS ols = new OLS(data.getX(), data.getY());
```

For regularized regression, you can use ridge or LASSO:

```
double lambda = 0.01;
RidgeRegression ridge = new RidgeRegression(data.getX(), data.getY(),
lambda);
LASSO lasso = new LASSO(data.getX(), data.getY(), lambda);
```

Using a RandomForest is very similar to the classification case:

```
int nbtrees = 100;
RandomForest rf = new RandomForest.Trainer(nbtrees)
        .setNumRandomFeatures(15)
        .setMaxNodes(128)
```

```
            .setNodeSize(10)
            .setSamplingRates(0.6)
            .train(data.getX(), data.getY());
```

Predicting is identical to the classification case as well. What we need to do is just use the `predict` method:

```
double result = model.predict(row);
```

JSAT

JSAT is also a general purpose library and contains a lot of implementations for solving regression problems.

As with classification, it needs a wrapper class for data and a special wrapper for regression:

```
double[][] X = ... //
double[] y = ... //
List<DataPointPair<Double>> data = new ArrayList<>(X.length);

for (int i = 0; i < X.length; i++) {
    DataPoint row = new DataPoint(new DenseVector(X[i]));
    data.add(new DataPointPair<Double>(row, y[i]));
}

RegressionDataSet dataset = new RegressionDataSet(data);
```

Once the dataset is wrapped in the right class, we can train models like this:

```
MultipleLinearRegression linreg = new MultipleLinearRegression();
linreg.train(dataset);;
```

The preceding code trains the usual OLS linear regression.



Unlike Smile, OLS does not produce a stable solution when the matrix is ill-conditioned, that is, it has some linearly dependent solutions. Use a regularized model in this case.

Training a regularized linear regression can be done with the following code:

```
RidgeRegression ridge = new RidgeRegression();
ridge.setLambda(lambda);
ridge.train(dataset);
```

Then, for predicting, we also need to do some conversion:

```
double[] row = .. . //
DenseVector vector = new DenseVector(row);
DataPoint point = new DataPoint(vector);
double result = model.regress(point);
```

Other libraries

Other libraries that we previously covered also have models for solving the regression problem.

For example, in LIBSVM, it is possible to do regression by setting the `svm_type` parameter to `EPSILON_SVR` or `NU_SVR`, and the rest of the code stays almost the same as in the classification case. Likewise, in LIBLINEAR, the regression problem is solved by choosing `L2R_L2LOSS_SVR` or `L2R_L2LOSS_SVR_DUAL` models.

It is also possible to solve the regression problem with neural networks, for example, in Encog. The only thing you need to change is the loss function: instead of minimizing a classification loss function (such as `logloss`) you should use a regression loss function, such as mean-squared error.

Since most of the code is pretty similar, there is no need to cover it in detail. As always, we have prepared some code examples in the chapter code repository, so feel free to have a look at them.

Evaluation

As with classification, we also need to evaluate the results of our models. There are some metrics that help to do that and select the best model. Let's go over the two most popular ones: **Mean Squared Error** (**MSE**) and **Mean Absolute Error** (**MAE**).

MSE

Mean Squared Error (**MSE**) is the sum of squared differences between the actual and predicted values. It is quite easy to compute it in Java:

```
double[] actual, predicted;

int n = actual.length;
double sum = 0.0;
for (int i = 0; i < n; i++) {
```

```
        diff = actual[i] - predicted[i];
        sum = sum + diff * diff;
    }

    double mse = sum / n;
```

Typically, the value of MSE is hard to interpret, which is why we often take a square root of MSE; this is called **Root Mean Squared Error** (**RMSE**). It is easier to interpret because it is in the same units as the target variable.

```
    double rmse = Math.sqrt(mse);
```

MAE

Mean Absolute Error (**MAE**), is an alternative metric for evaluating performance. Instead of taking the squared error, it only takes the absolute value of the difference between the actual and predicted value. This is how we can compute it:

```
    double sum = 0.0;
    for (int i = 0; i < n; i++) {
        sum = sum + Math.abs(actual[i] - predicted[i]);
    }

    double mae = sum / n;
```

Sometimes we have outliers in the data--the values with quite irregular values. If we have a lot of outliers, we should prefer MAE to RMSE, since it is more robust to them. If we do not have many outliers, then RMSE should be the preferred choice.

There are also other metrics such as MAPE or RMSE, but they are used less often, so we won't cover them.

While we went over the libraries for solving the regression problem only briefly, with the foundation we got from the overview for solving the classification problem, it is enough to do a project on regression.

Case study - hardware performance

In this project, we will try to predict how much time it will take to multiply two matrices on different computers.

The dataset for this project originally comes from the paper *Automatic selection of the fastest algorithm implementation* by Sidnev and Gergel (2014), and it was made available at a machine learning competition organized by Mail.RU. You can check the details at `http://mlbootcamp.ru/championship/7/`.



The content is in Russian, so if you do not speak it, it is better to use a browser with translation support.

You will find a copy of the dataset along with the code for this chapter.

This dataset has the following data:

- `m`, `k`, and `n` represent the dimensionality of the matrices, with `m*k` being the dimensionality of matrix `A` and `k*n` being the dimensionality of matrix `B`
- Hardware characteristics such as CPU speed, number of cores, whether hyper-threating is enabled or not, and the type of CPU
- The operation system

The solution for this problem can be quite useful for research, when selecting hardware to buy for running the experiment. In that case. you can can use the model for selecting a build which should result in the best performance.

So, the goal is to predict how many seconds it will take to multiply two matrices given their size and the characteristics of the environment. Although the paper uses MAPE as the evaluation metric, we will use RMSE as it is easier to implement and interpret.

First, we need to read the data. There are two files, one with features and one with labels. Let's read the target first:

```
DataFrame<Object> targetDf =
DataFrame.readCsv("data/performance/y_train.csv");
List<Double> targetList = targetDf.cast(Double.class).col("time");
double[] target = Doubles.toArray(targetList);
```

Next, we read the features:

```
DataFrame<Object> dataframe =
DataFrame.readCsv("data/performance/x_train.csv");
```

If we look at the data, we can notice that sometimes the missing values are encoded as a string `None`. We need to convert it to a real Java `null`. To do this, we can define a special function:

```
private static List<Object> noneToNull(List<Object> memfreq) {
    return memfreq.stream()
            .map(s -> isNone(s) ? null : Double.parseDouble(s.toString()))
            .collect(Collectors.toList());
}
```

Now, use it to process the original columns, then remove them, and add the transformed ones:

```
List<Object> memfreq = noneToNull(dataframe.col("memFreq"));
List<Object> memtRFC = noneToNull(dataframe.col("memtRFC"));
dataframe = dataframe.drop("memFreq", "memtRFC");
dataframe.add("memFreq", memfreq);
dataframe.add("memtRFC", memtRFC);
```

There are some categorical variables in the dataset. We can look at them. First, let's create a data frame, which contains the types of original frames:

```
List<Object> types = dataframe.types().stream()
             .map(c -> c.getSimpleName())
             .collect(Collectors.toList());
List<Object> columns = new ArrayList<>(dataframe.columns());
DataFrame<Object> typesDf = new DataFrame<>();
typesDf.add("column", columns);
typesDf.add("type", types);
```

Since we are interested only in categorical values, we need to select the features that are of type `String`:

```
DataFrame<Object> stringTypes = typesDf.select(p ->
p.get(1).equals("String"));
```

The way categorical variables are often used in the machine learning problem is called dummy-coding, or one hot encoding. In this coding scheme:

- We create as many columns as there are possible values
- For each observation, we put the number `1` for the column, which corresponds to the value of the categorical variable, and the remaining columns get `0`

Joinery can do this conversion automatically for us:

```
double[][] X = dataframe.toModelMatrix(0.0);
```

The preceding code will apply one hot encoding scheme to all categorical variables.

However, for the data that we have, some values of the categorical variables occur only once or just a few times. Typically, we are not interested in such infrequently occurring values, so we can replace them with some artificial value such as `OTHER`.

This is how we do this in Joinery:

- Remove all categorical columns from `DataFrame`
- For each column, we calculate how many times the values occur and replace infrequent with `OTHER`

Let's translate it into Java code. This way we get the categorical variables:

```
Object[] columns = stringTypes.col("column").toArray();
DataFrame<Object> categorical = dataframe.retain(columns);
dataframe = dataframe.drop(stringTypes.col("column").toArray());
```

For counting, we can use a `Multiset` collection from Guava. Then, we replace the infrequent ones with `OTHER` and put the result back to the DataFrame:

```
for (Object column : categorical.columns()) {
    List<Object> data = categorical.col(column);
    Multiset<Object> counts = HashMultiset.create(data);

    List<Object> cleaned = data.stream()
            .map(o -> counts.count(o) >= 50 ? o : "OTHER")
            .collect(Collectors.toList());

    dataframe.add(column, cleaned);
}
```

After this processing, we can convert the DataFrame into the matrix and put it into our `Dataset` object:

```
double[][] X = dataframe.toModelMatrix(0.0);
Dataset dataset = new Dataset(X, target);
```

Now we are ready to start training models. Again, we will use Smile for the implementation of machine learning algorithms, and the code for other libraries is available in the chapter code repository.

We already decided that we will use RMSE as the evaluation metrics. Now we need to set up the cross-validation scheme and hold out the data for final evaluation:

```
Split trainTestSplit = dataset.shuffleSplit(0.3);
Dataset train = trainTestSplit.getTrain();
Dataset test = trainTestSplit.getTest();
List<Split> folds = train.shuffleKFold(3);
```

We can reuse the function we wrote for the classification case and slightly adapt it to the regression case:

```
public static DescriptiveStatistics crossValidate(List<Split> folds,
        Function<Dataset, Regression<double[]>> trainer) {
    double[] aucs = folds.parallelStream().mapToDouble(fold -> {
        Dataset train = fold.getTrain();
        Dataset validation = fold.getTest();
        Regression<double[]> model = trainer.apply(train);
        return rmse(model, validation);
    }).toArray();

    return new DescriptiveStatistics(aucs);
}
```

In the preceding code, we first train a regression model and then evaluate its RMSE on the validation dataset.

Before going into modeling, let's first come with a simple baseline solution. In case of Regression, always predicting the mean can be such a baseline:

```
private static Regression<double[]> mean(Dataset data) {
    double meanTarget = Arrays.stream(data.getY()).average().getAsDouble();
    return x -> meanTarget;
}
```

Let's use it as the function for cross-validation for the baseline calculation:

```
DescriptiveStatistics baseline = crossValidate(folds, data -> mean(data));
System.out.printf("baseline: rmse=%.4f &pm; %.4f%n", baseline.getMean(),
baseline.getStandardDeviation());
```

It prints the following to the console:

```
baseline: rmse=25.1487 &pm; 4.3445
```

Our baseline solution is wrong by 25 seconds on average and the spread is 4.3 seconds.

Now we can try to train a simple OLS regression:

```
DescriptiveStatistics ols = crossValidate(folds, data -> {
    return new OLS(data.getX(), data.getY());
});

System.out.printf("ols: rmse=%.4f &pm; %.4f%n", ols.getMean(),
ols.getStandardDeviation());
```

We should note that Smile gives us a warning that the matrix is not full rank and it will use **Singular Value Decomposition** (**SVD**) to solve the OLS problem. We can either ignore it or explicitly tell it to use SVD:

```
new OLS(data.getX(), data.getY(), true);
```

In either case, it prints the following to the console:

```
ols: rmse=15.8679 &pm; 3.4587
```

When we use a regularized model, we do not typically worry about correlated columns. Let's try LASSO with different values of `lambda`:

```
double[] lambdas = { 0.1, 1, 10, 100, 1000, 5000, 10000, 20000 };
for (double lambda : lambdas) {
    DescriptiveStatistics summary = crossValidate(folds, data -> {
        return new LASSO(data.getX(), data.getY(), lambda);
    });

    double mean = summary.getMean();
    double std = summary.getStandardDeviation();
    System.out.printf("lasso λ=%9.1f, rmse=%.4f &pm; %.4f%n", lambda, mean,
std);
}
```

It produces the following output:

```
lasso λ=      0.1, rmse=15.8679 &pm; 3.4587
lasso λ=      1.0, rmse=15.8678 &pm; 3.4588
lasso λ=     10.0, rmse=15.8650 &pm; 3.4615
lasso λ=    100.0, rmse=15.8533 &pm; 3.4794
lasso λ=   1000.0, rmse=15.8650 &pm; 3.5905
lasso λ=   5000.0, rmse=16.1321 &pm; 3.9813
lasso λ=  10000.0, rmse=16.6793 &pm; 4.3830
lasso λ=  20000.0, rmse=18.6088 &pm; 4.9315
```



Note that the LASSO implementation from Smile version 1.1.0 will have a problem with this dataset because there are linearly dependent columns. To avoid this, you should use the 1.2.0 version, which, at the moment of writing, is not available from Maven Central, and you need to build it yourself if you want to use it. We have already discussed how to do this.

We can also try RidgeRegression, but its performance is very similar to OLS and LASSO, so we will omit it here.

It seems that the results of OLS is not so different from LASSO, so we select it as the final model and use it since it's the simplest model:

```
OLS ols = new OLS(train.getX(), train.getY(), true);
double testRmse = rmse(lasso, test);
System.out.printf("final rmse=%.4f%n", testRmse);
```

This gives us the following output:

```
final rmse=15.0722
```

So the performance of the selected model is consistent with our cross-validation, which means that the model is able to generalize to the unseen data well.

Summary

In this chapter, we spoke about supervised machine learning and about two common supervised problems: classification and regression. We also covered the libraries, which are commonly-used algorithms, implemented them, and learned how to evaluate the performance of these algorithms.

There is another family of machine learning algorithms that do not require the label information; these methods are called unsupervised learning--in the next chapter, we will talk about them.

5
Unsupervised Learning - Clustering and Dimensionality Reduction

In the previous chapter, covered with with machine learning in Java and discussed how to approach the supervised learning problem when the label information is provided.

Often, however, there is no label information, and all we have is just some data. In this case, it is still possible to use machine learning, and this class of problems is called **unsupervised learning**; there are no labels, hence no **supervision**. Cluster analysis belongs to one of these algorithms. Given some dataset, the goal is to group the items from there such that similar items are put into the same group.

Additionally, some unsupervised learning techniques can be useful when there is label information.

For example, the dimensionality reduction algorithm tries to *compress* the dataset such that most of the information is preserved and the dataset can be represented with fewer features. What is more, dimensionality reduction is also useful for performing cluster analysis, and cluster analysis can be used for performing dimensionality reduction.

We will see how to do it all of this in this chapter. Specifically, we will cover the following topics:

- Unsupervised dimensionality reduction methods, such as PCA and SVD
- Cluster analysis algorithms, such as k-means
- Available implementations in Java

By the end of this chapter, you will know how to cluster the data you have and how to perform dimensionality reduction in Java using Smile and other Java libraries.

Dimensionality reduction

Dimensionality reduction, as the name suggests, reduces the dimensionality of your dataset. That is, these techniques try to compress the dataset such that only the most useful information is retained, and the rest is discarded.

By dimensionality of a dataset, we mean the number of features of this dataset. When the dimensionality is high, that is, there are too many features, it can be bad due to the following reasons:

- If there are more features than the items of the dataset, the problem becomes ill-defined and some linear models, such as **ordinary least squares** (**OLS**) regression cannot handle this case
- Some features may be correlated and cause problems with training and interpreting the models
- Some of the features can turn out to be noisy or irrelevant and confuse the model
- Distances start to make less sense in high dimensions -- this problem is commonly referred to as the curse of dimensionality
- Processing a large set of features may be computationally expensive

In the case of high dimensionality, we are interested in reducing the dimensionality such that it becomes manageable. There are several ways of doing so:

- **Supervised dimensionality reduction methods such as feature selection**: We use the information we have about the labels to help us decide which features are useful and which are not
- **Unsupervised dimensionality reduction such as feature extraction**: We do not use the information about labels (either because we do not have it or would not like to do it) and try to compress the large set of features into smaller ones

In this chapter, we will speak about the second type, that is, unsupervised dimensionality reduction, and in particular, about feature extraction.

Unsupervised dimensionality reduction

The main idea behind feature extraction algorithms is that they take in some dataset with high dimensionality, process it, and return a dataset with much smaller set of new features.

Note that the returned features are new, they are **extracted** or **learned** from the data. But this extraction is performed in such a way that the new representation of data retains as much information from the original features as possible. In other words, it takes the data represented with old features, transforms it, and returns a new dataset with entirely new features.

There are many feature extraction algorithms for dimensionality reduction, including:

- **Principal Component Analysis** (**PCA**) and **Singular Value Decomposition** (**SVD**)
- **Non-Negative Matrix Factorization** (**NNMF**)
- Random projections
- **Locally Linear Embedding** (**LLE**)
- t-SNE

In this chapter we will cover PCA, SVD, and random projections. The other techniques are less popular and less often used in practice, so we will not discuss them in this book.

Principal Component Analysis

Principal Component Analysis (**PCA**) is the most famous algorithm for feature extraction. The new feature representation learned by PCA is a linear combination of the original features such that the variance within the original data is preserved as much as possible.

Let's look at this algorithm in action. First, we will consider the dataset we already used the performance prediction. For this problem, the number of features is relatively large; after encoding categorical variables with one-hot-encoding there are more than 1,000 features, and only 5,000 observations. Clearly, 1,000 features is quite a lot for such a small sample size, and this may cause problems when building a machine learning model.

Let's see if we can reduce the dimensionality of this dataset without harming the performance of our model.

But first, let's recall how PCA works. There are usually the following steps to complete:

1. First, you perform mean-normalization of the datasets -- transform the dataset such that the mean value of each column becomes zero.
2. Then, compute the covariance or correlation matrix.
3. After that, perform **Eigenvalue Decomposition** (**EVD**) or **Singular Value Decomposition** (**SVD**) of the covariance/correlation matrix.
4. The result is a set of principal components, each of which explains a part of the variance. The principal components are typically ordered such that the first components explain most of the variance, and last components explain very little of it.
5. In the last step, we throw away components that do not carry any variance on them, and keep only first principal components with large variance. To select the number of components to keep, we typically use the cumulated ratio of explained variance to the total variance.
6. We use these components to compress the original dataset by performing the projection of the original data on the basis formed by these components.
7. After doing these steps, we have a dataset with smaller number of features, but most of the information of the original dataset is preserved.

There are a number of ways we can implement PCA in Java, but we can take one of the libraries such as Smile, which offers off-the-shelf implementations. In Smile, PCA already performs mean-normalization, then computes the covariance matrix and automatically decides whether to use EVD or SVD. All we need to do is to give it a data matrix, and it will do the rest.

Typically, PCA is performed on the covariance matrix, but sometimes, when some of the original features are on a different scale, the ratio of the explained variance can become misleading.

For example, if one of the features we have is distance in kilometers, and the other one is time is milliseconds, then the second feature will have larger variance simply because the numbers are a lot higher in the second feature. Thus, this feature will have the dominant presence in the resulting components.

To overcome this problem, we can use the correlation matrix instead of the covariance one, and since the correlation coefficient is unitless, the PCA results will not be affected by different scales. Alternatively, we can perform standardization of the features in our dataset, and, in effect, computing covariance will be the same as computing correlation.

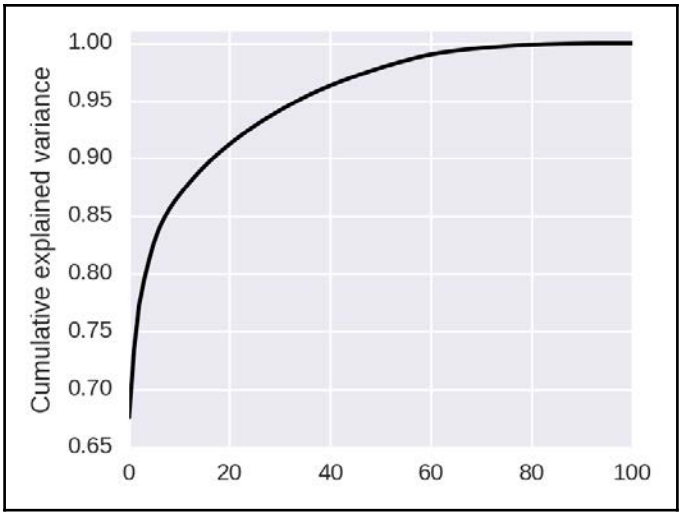
So, first we will standardize the data using the `StandardizationPreprocessor` we wrote previously:

```
StandardizationPreprocessor preprocessor =
StandardizationPreprocessor.train(dataset);
dataset = preprocessor.transform(dataset)
```

Then we can run PCA on the transformed dataset and have a look at the cumulative variance:

```
PCA pca = new PCA(dataset.getX(), false);
double[] variance = pca.getCumulativeVarianceProportion();
System.out.println(Arrays.toString(variance));
```

If we take the output and plot the first one hundred components, we will see the following picture:



As we can see, the principal component explains about 67% of variance, and the cumulative explained ratio quite quickly reaches 95% at less than **40** components, 99% at 61, and it is almost 100% at **80** components. This means that if we take only the first **80** components, it will be enough to capture almost all variance present in the dataset. This means that we should be able to safely compress our 1,000 plus dimensional datasets into only 80 dimensions.

Let's test it. First, let's try to do OLS without PCA. We will take the code from the previous chapter:

```
Dataset train = trainTestSplit.getTrain();

List<Split> folds = train.shuffleKFold(3);
DescriptiveStatistics ols = crossValidate(folds, data -> {
    return new OLS(data.getX(), data.getY());
});
```

This prints the following output:

```
ols: rmse=15.8679 &pm; 3.4587
```

Now let's try to cap the number of principal components at the 95%, 99%, and 99.9% levels and see what happens with the error:

```
double[] ratios = { 0.95, 0.99, 0.999 };

for (double ratio : ratios) {
    pca = pca.setProjection(ratio);
    double[][] projectedX = pca.project(train.getX());
    Dataset projected = new Dataset(projectedX, train.getY());

    folds = projected.shuffleKFold(3);
    ols = crossValidate(folds, data -> {
        return new OLS(data.getX(), data.getY());
    });

    double mean = ols.getMean();
    double std = ols.getStandardDeviation()
    System.out.printf("ols (%.3f): rmse=%.4f &pm; %.4f%n", ratio, mean,
std);
}
```

This produces the following output:

```
ols (0.950): rmse=18.3331 &pm; 3.6308
ols (0.990): rmse=16.0702 &pm; 3.5046
ols (0.999): rmse=15.8656 &pm; 3.4625
```

As we see, keeping 99.9% of variance with PCA gives the same performance as the OLS regression fit on the original dataset. For this dataset, 99.9% of variance is explained by only 84 principal components, and there are 1070 features in the original dataset. Thus, we managed to reduce the dimensionality of the data by keeping only 7.8% of the original data size without loosing any performance.

Sometimes, however, the implementation of PCA from Smile and other similar packages is not the best in terms of performance. Next, we will see why and how to deal with it.

Truncated SVD

The previous code (in this case, using Smile) performs full PCA via full SVD or EVD. Here, **full** means that it computes all eigenvalues and eigenvectors, which may be computationally expensive, especially when we only need the first 7.8% of the principal components. However, we do not necessarily have to always compute the full PCA, and instead we can use truncated SVD. Truncated SVD computes only the specified number of principal components, and it is usually a lot faster than the full version.

Smile also provides an implementation of truncated SVD. But before using it, let's quickly revise SVD.

SVD of a matrix *X* computes the bases for the rows and the columns of *X* such that:

$$XV = US$$

Here, the equation is explained as follows:

- The columns of *V* form the basis for the rows of *X*
- The columns of *U* form the basis for the rows of *X*
- *S* is a diagonal matrix with the singular values of *X*

Often, SVD is written in this form:

$$X = USV^T$$

So, SVD decomposes the matrix *X* into three matrices *U*, *S*, and *V*.

When SVD is truncated to dimensionality *K*, the matrices *U* and *V* have only *K* columns, and we compute only *K* singular values. If we then multiply the original matrix *X* by the truncated *V*, or, alternatively, multiply *S* by *U*, we will obtain the reduced projection of the rows of *X* to this new SVD basis.

This will bring the original matrix to the new reduced space, and we can use the results as features instead of the original one.

Now, we are ready to apply it. In Smile it will look like this:

```
double[][] X = ... // X is mean-centered
Matrix matrix = new Matrix(X);
SingularValueDecomposition svd =
SingularValueDecomposition.decompose(matrix, 100);
```

Here, `Matrix` is a class from Smile for storing dense matrices. The matrices *U*, *S*, and *V* are returned inside the `SingularValueDecomposition` object, *U* and *V* as two-dimensional double arrays and *S* as a one-dimensional array of doubles.

Now, we need to get the reduced representation of our data matrix *X*. As we discussed earlier, there are two ways of doing it:

- By computing $X \cdot V$
- By computing $U \cdot S$

First, let's have a look at computing $U \cdot S$.

In Smile, the `decompose` method from `SingularValueDecomposition` returns *S* as a one-dimensional array of doubles, so we need to convert it to the matrix form. We can take advantage of the fact that *S* is diagonal and use it for speeding up the multiplication.

Let's use the Commons Math library for that. There is a special implementation for diagonal matrices, so we will use it, and usual array-backed matrix for *U*.

```
DiagonalMatrix S = new DiagonalMatrix(svd.getSingularValues());
Array2DRowRealMatrix U = new Array2DRowRealMatrix(svd.getU(), false);
```

Now we can multiply these two matrices:

```
RealMatrix result = S.multiply(U.transpose()).transpose();
double[][] data = result.getData();
```

Note that instead of multiplying *U* by *S*, we do it in the opposite direction and then transpose: this takes advantage of *S* being diagonal, and makes the matrix multiplications a lot faster. In the end, we extract the array of doubles to be used in Smile.

If we use this code for the problem of predicting performance, it takes less than 4 seconds, and that's including the matrix multiplication part. This is a great speed improvement over the full PCA version, which, on our laptop, takes more than 1 minute.

Another way to compute the projection is to calculate $X \cdot V$. Let's once again use Commons Math for that:

```
Array2DRowRealMatrix X = new Array2DRowRealMatrix(dataX, false);
Array2DRowRealMatrix V = new Array2DRowRealMatrix(svd.getV(), false);
double[][] data = X.multiply(V).getData();
```

This takes slightly more time than computing $U \cdot S$, since neither of the matrices are diagonal. However, the difference in speed is only marginal: computing SVD and reducing the dimensionality this way takes less than 5 seconds for the performance prediction problem.

When you use SVD to reduce the dimensionality of *training* data, there is no difference between these two methods. However, we cannot apply the $U \cdot S$ approach to new unseen data, because both *U* and *S* are produced for the matrix *X*, for which we trained the SVD. Instead we use the $X \cdot V$ approach. Note that *X* in this case will be the new matrix containing the test data, not the same *X* we used for training the SVD.

In code it will look like this:

```
double[] trainX = ...;
double[] testX = ...;

Matrix matrix = new Matrix(trainX);
SingularValueDecomposition svd =
SingularValueDecomposition.decompose(matrix, 100);

double[][] trainProjected = mmult(trainX, svd.getV());
double[][] testProjected = mmult(testX, svd.getV());
```

Here, `mmult` is a method that multiplies the matrix *X* by the matrix *V*.

There is another implementation detail: in Smile's PCA implementation, we determine the number of needed dimensionality using the ratio of explained variance. Recall that we do this by invoking `getCumulativeVarianceProportion` on the `PCA` object, and usually keep the number of components high enough to get at least 95% or 99% of variance.

However, since we use SVD directly, we do not know this ratio now. It means that to be able to choose the right dimensionality, we need to implement this ourselves. Luckily, it is not very complex to do; first, we need to calculate the overall variance of the dataset, and then the variances of all the principal components. The latter can be obtained from the singular values (the matrix *S*). The singular values correspond to standard deviations, so to get the variance, we just need to square them. Finally, finding the ratio is simple, and we just need to divide one by another.

Let's see how it looks in code. First, we use Commons Math to compute the total variance:

```
Array2DRowRealMatrix matrix = new Array2DRowRealMatrix(dataset.getX(),
false);
int ncols = matrix.getColumnDimension();

double totalVariance = 0.0;
for (int col = 0; col < ncols; col++) {
    double[] column = matrix.getColumn(col);
    DescriptiveStatistics stats = new DescriptiveStatistics(column);
    totalVariance = totalVariance + stats.getVariance();
}
```

Now we can compute the cumulated ratios from singular values:

```
int nrows = X.length;
double[] singularValues = svd.getSingularValues();
double[] cumulatedRatio = new double[singularValues.length];

double acc = 0.0;
for (int i = 0; i < singularValues.length; i++) {
    double s = singularValues[i];
    double ratio = (s * s / nrows) / totalVariance;
    acc = acc + ratio;
    cumulatedRatio[i] = acc;
}
```

After running this code, the `cumulatedRatio` array will contain the desired ratios. The result should be exactly the same as from Smile's PCA implementation from `pca.getCumulativeVarianceProportion()`.

Truncated SVD for categorical and sparse data

Dimensionality reduction can be very useful for datasets with many categorical variables, especially when each of these variables have a lot of possible values.

When we have sparse matrices of very high dimensionality, computing full SVD is typically very expensive. Thus, truncated SVD is especially for this case, and here we will see how we can use it. Later on in the next chapter, we will see that this is also pretty useful for text data, and we will cover this case in the next chapter. For now, we will have a look at how to use it for categorical variables.

For this, we will use a dataset about customer complaints from Kaggle. You can download it from here: `https://www.kaggle.com/cfpb/us-consumer-finance-complaints`.

This dataset contains complaints that customers of banks and other financial institutions have filed, and also contains additional information about these complaints, as follows:

- The product for which the complaint is filed can be *Mortgage Loan*, *Student Loan*, *Debt Collection*, and so on. There are 11 types of products.
- The reported issue about the product, such as *incorrect information*, *false statements*, and so on. There are 95 types of issues.
- The company for which the complaint is filed, more than 3,000 companies.
- `submitted_via` is how the complaint was sent, 6 possible options such as, *web and e-mail*.
- The state and zipcode is 63 and 27,000 possible values respectively.
- `consumer_complaint_narrative` is a free text description of the problem.

We see that there is a large number of categorical variables in this dataset. As we have already discussed in previous chapters, the typical way of encoding categorical variables is one-hot-encoding (also called **dummy-coding**). The idea is that for each possible value of a variable, we create a separate feature, and put the value `1` there if an item has this particular value. Txhe columns for all other possible values will have `0` there.

The easiest way to implement this is to use feature hashing, which sometimes is referred as the hashing trick.

It can be done quite easily by following these steps:

- We specify the dimensionality of our sparse matrix beforehand, and for that we take some reasonably large number
- Then, for each value, we compute the hash of this value
- Using the hash, we compute the number of the column in the sparse matrix and set the value of this column to `1`

So, let's try to implement it. First, we load the dataset and keep only the categorical variables:

```
DataFrame<Object> categorical = dataframe.retain("product", "sub_product",
"issue",
        "sub_issue", "company_public_response", "company",
        "state", "zipcode", "consumer_consent_provided",
        "submitted_via");
```

Now, let's implement feature hashing for encoding them:

```
int dim = 50_000;
SparseDataset result = new SparseDataset(dim);
```

```
int ncolOriginal = categorical.size();
ListIterator<List<Object>> rows = categorical.iterrows();

while (rows.hasNext()) {
    int rowIdx = rows.nextIndex();
    List<Object> row = rows.next();
    for (int colIdx = 0; colIdx < ncolOriginal; colIdx++) {
        Object val = row.get(colIdx);
        String stringValue = colIdx + "_" + Objects.toString(val);
        int targetColIdx = Math.abs(stringValue.hashCode()) % dim;

        result.set(rowIdx, targetColIdx, 1.0);
    }
}
```

What happens here is that we first create a `SparseDataset`-- a class from Smile for keeping row-based sparse matrices. Next, we say that the matrix should have the dimensionality specified by the variable `dim`. The value of `dim` should be high enough so that the chance of collision is not very high. Typically, however, it is not a big deal if there are collisions.



If you set the value of dim to a very large number, there could be some performance problems when we later decompose the matrix.

Feature hashing is a very simple approach and often works really well in practice. There is another approach, which is more complex to implement, but it ensures that there are no hash collisions. For that, we build a map from all the possible values to column indexes, and then build the sparse matrix.

Building the map will look like this:

```
Map<String, Integer> valueToIndex = new HashMap<>();
List<Object> columns = new ArrayList<>(categorical.columns());

int ncol = 0;

for (Object name : columns) {
    List<Object> column = categorical.col(name);
    Set<Object> distinct = new HashSet<>(column);
    for (Object val : distinct) {
        String stringValue = Objects.toString(name) + "_" +
Objects.toString(val);
        valueToIndex.put(stringValue, ncol);
        ncol++;
    }
}
```

The `ncol` variable contains the number of columns, which is the dimensionality of our future sparse matrix. Now we can construct the actual matrix. It is very similar to what we had before, but, instead of hashing, we now look the indexes up in the map:

```
SparseDataset result = new SparseDataset(ncol);

ListIterator<List<Object>> rows = categorical.iterrows();
while (rows.hasNext()) {
    int rowIdx = rows.nextIndex();
    List<Object> row = rows.next();
    for (int colIdx = 0; colIdx < columns.size(); colIdx++) {
        Object name = columns.get(colIdx);
        Object val = row.get(colIdx);
        String stringValue = Objects.toString(name) + "_" +
Objects.toString(val);
        int targetColIdx = valueToIndex.get(stringValue);

        result.set(rowIdx, targetColIdx, 1.0);
    }
}
```

After doing this, we have a `SparseDataset` object, which contains the data in a row-based format. Next, we need to be able to put it to the SVD solver, and for that we need to convert it to a different column-based format. This is implemented in the `SparseMatrix` class. Luckily, there is a special method in the `SparseDataset` class which does the conversion, so we use it:

```
SparseMatrix matrix = dataset.toSparseMatrix();
SingularValueDecomposition svd =
SingularValueDecomposition.decompose(matrix, 100);
```

The decomposition is quite fast; computing SVD of the feature hashing matrix took about 28 seconds and the usual one-hot-encoding took about 24 seconds. Remember that there are 0.5 million rows in this dataset, so the speed is pretty good. To our knowledge, other Java implementations of SVD are not able to provide the same performance.

Now, when SVD is computed, we need to project the original matrix to the reduced space, like we did in the case of dense matrices previously.

The $U \cdot S$ projection can be done exactly as before, because both *U* and *S* are dense. However, *X* is sparse, so we need to find a way to multiply sparse *X* and dense *X* efficiently.

Unfortunately, neither Smile nor Commons Math has a suitable implementation for this. Therefore, we need to use another library, and this problem can be solved with **Matrix Java Toolkit** (**MTJ**). This library is based on netlib-java, which is a wrapper for low-level high-performance libraries such as BLAS, LAPACK, and ARPACK. You can read more on its GitHub page: `https://github.com/fommil/matrix-toolkits-java`.

Since we use Maven, it will take care of downloading the binary dependencies and linking them to the project. All we need need to do is to specify the following dependency:

```
<dependency>
  <groupId>com.googlecode.matrix-toolkits-java</groupId>
  <artifactId>mtj</artifactId>
  <version>1.0.2</version>
</dependency>
```

We need to multiply two matrices, X and V, with the condition that X is sparse while V is dense. Since X is on the left side of the multiplication operator, the most efficient way to store the values of X is a row-based sparse matrix representation. For V the most efficient representation is a column-based dense matrix.

But before we can do it, we first need to convert Smile's `SparseDataset` into MTJ's sparse matrix. For that we use a special builder: `FlexCompRowMatrix` class, which is good for populating a matrix with values, but not so good for multiplication. Once we constructed the matrix, we convert it to `CompRowMatrix`, which has a more efficient internal representation and is better suited for multiplication purposes.

Here's how we do it:

```
SparseDataset dataset = ... //
int ncols = dataset.ncols();
int nrows = dataset.size();
FlexCompRowMatrix builder = new FlexCompRowMatrix(nrows, ncols);

SparseArray[] array = dataset.toArray(new SparseArray[0]);
for (int rowIdx = 0; rowIdx < array.length; rowIdx++) {
    Iterator<Entry> row = array[rowIdx].iterator();
    while (row.hasNext()) {
        Entry entry = row.next();
        builder.set(rowIdx, entry.i, entry.x);
    }
}

CompRowMatrix X = new CompRowMatrix(builder);
```

The second step is to create a dense matrix. This step is simpler:

```
DenseMatrix V = new DenseMatrix(svd.getV());
```

Internally, MTJ stores dense matrices column-wise, and it is ideal for our purposes.

Next, we need to create a matrix object, which will contain the results, and then we multiply *X* by *V*:

```
DenseMatrix XV = new DenseMatrix(X.numRows(), V.numColumns());
X.mult(V, XV);
```

Finally, we need to extract the double array data from the result matrix. For performance purposes, MTJ stores the data as a one-dimensional double array, so we need to convert it to the conventional representation. We do it like this:

```
double[] data = XV.getData();
int nrows = XV.numRows();
int ncols = XV.numColumns();
double[][] result = new double[nrows][ncols];

for (int col = 0; col < ncols; col++) {
    for (int row = 0; row < nrows; row++) {
        result[row][col] = data[row + col * nrows];
    }
}
```

In the end, we have the result array, which captures most of the variability of the original dataset, and we can use it for cases where a small dense matrix is expected.

This transformation is especially useful for the second topic of this chapter: clustering. Typically, we use distances for clustering data points, but when it comes to high-dimensional spaces, the distances are no longer meaningful, and this phenomenon is known as the **curve of dimensionality**. However, in the reduced SVD space, the distances still make sense and when we apply cluster analysis, the results are typically better.

This is also a very useful method for processing natural language texts, as typically texts are represented as extremely high dimensional and very sparse matrices. We will come back to this in `Chapter 6`, *Working with Texts - Natural Language Processing and Information Retrieval*.

Note that unlike in the usual PCA case, we do not perform mean-centering here. There are a few reasons for this:

- If we do this, the matrix will become dense and will occupy too much memory, so it will not be possible to process it in a reasonable amount of time
- In sparse matrices, the mean is already very close to zero, so there is no need to perform mean normalization

Next, we will look at a different dimensionality reduction technique, which is extremely simple, requires no learning, and is pretty fast.

Random projection

PCA tries to find some structure in data and use it for reducing the dimensionality; it finds such a basis in which most of the original variance is preserved. However, there is an alternative approach instead of trying to learn the basis, just generate it randomly and then project the original data on it.

Surprisingly, this simple idea works quite well in practice. The reason for that is, this transformation preserves distances. What this means is that if we have two objects that are close to each other in the original space, then, when we apply the projection, they still remain close. Likewise, if the objects are far away from each other, then they will remain far in the new reduced space.

Smile already has implementation for random projection, which takes the input dimensionality and the desired output dimensionality:

```
double[][] X = ... // data
int inputDimension = X[0].length;
int outputDimension = 100;
smile.math.Math.setSeed(1);
RandomProjection rp = new RandomProjection(inputDimension,
outputDimension);
```

Note that we explicitly set the seed for random number generator; since the basis for random projections is generated randomly, we want to ensure reproducibility.



Setting seed is only possible in version 1.2.1, which was not available on Maven Central at the moment of writing.

It is implemented in Smile in the following way:

- First, a set of random vectors are sampled from a Gaussian distribution
- Then, the vectors are made orthonormal via the Gram-Schmidt algorithm, that is, they are first made orthogonal and then the length is normalized to 1
- The projection is made on this orthonormal basis

Let's use it for performance prediction and then fit the usual OLS:

```
double[][] X = dataset.getX();
int inputDimension = X[0].length;
int outputDimension = 100;
smile.math.Math.setSeed(1);
RandomProjection rp = new RandomProjection(inputDimension,
outputDimension);

double[][] projected = rp.project(X);
dataset = new Dataset(projected, dataset.getY());

Split trainTestSplit = dataset.shuffleSplit(0.3);
Dataset train = trainTestSplit.getTrain();

List<Split> folds = train.shuffleKFold(3);
DescriptiveStatistics ols = crossValidate(folds, data -> {
    return new OLS(data.getX(), data.getY());
});

System.out.printf("ols: rmse=%.4f &pm; %.4f%n", ols.getMean(),
ols.getStandardDeviation());
```

It is very fast (it takes less than a second on our laptop) and this code produces the following result:

```
ols: rmse=15.8455 &pm; 3.3843
```

The result is pretty much the same as in plain OLS or OLS on PCA with 99.9% variance.

However, the implementation from Smile only works with dense matrices, and at the moment of writing there is no support for sparse matrices. Since the method is pretty straightforward, it is not difficult to implement it ourselves. Let's implement a simplified version of generating the random basis.

To generate the basis, we sample from the Gaussian distribution with zero mean and a standard deviation equal to `1 / new_dimensionality`, where `new_dimensionality` is the desired dimensionality of the new reduced space.

Let's use Commons Math for it:

```
NormalDistribution normal = new NormalDistribution(0.0, 1.0 /
outputDimension);
normal.reseedRandomGenerator(seed);
double[][] result = new double[inputDimension][];

for (int i = 0; i < inputDimension; i++) {
    result[i] = normal.sample(outputDimension);
}
```

Here, we have the following parameters:

- `inputDimension`: This is the dimensionality of the matrix we want to project, that is, the number of columns of this matrix
- `outputDimension`: This is the desired dimensionality of the projection
- `seed`: This is the random number generator seed for reproducibility

First, let's sanity-check the implementation and apply it to the same performance problem. Even though it is dense, it is enough for testing purposes:

```
double[][] X = dataset.getX();
int inputDimension = X[0].length;
int outputDimension = 100;
int seed = 1;
double[][] basis = Projections.randomProjection(inputDimension,
outputDimension, seed);
double[][] projected = Projections.project(X, basis);
dataset = new Dataset(projected, dataset.getY());

Split trainTestSplit = dataset.shuffleSplit(0.3);
Dataset train = trainTestSplit.getTrain();

List<Split> folds = train.shuffleKFold(3);
DescriptiveStatistics ols = crossValidate(folds, data -> {
    return new OLS(data.getX(), data.getY());
});

System.out.printf("ols: rmse=%.4f &pm; %.4f%n", ols.getMean(),
ols.getStandardDeviation());
```

Here we have two methods:

- `Projections.randomProjection`: This generates the random basis, which we implemented previously.

- `Projections.project`: This projects the matrix *X* onto the basis, and it is implemented by multiplying the matrix *X* onto the matrix of the basis.

After running the code, we see the following output:

```
ols: rmse=15.8771 &pm; 3.4332
```

This indicates that our implementation has passed the sanity check, the results make sense, and the method is implemented correctly.

Now we need to change the projection method such that it can be applied to sparse matrices. We have already done that, but let's once again take a look at the outline:

- Put the sparse matrix into `RompRowMatrix`, **compressed row storage** (**CRS**) matrix
- Put the basis into `DenseMatrix`
- Multiply the matrices and write the results into `DenseMatrix`
- Unwrap the underlying data from `DenseMatrix` into a two-dimensional double array

For the categorical example from the complaints dataset, it will look like the following:

```
DataFrame<Object> categorical = ... // data
SparseDataset sparse = OHE.hashingEncoding(categorical, 50_000);
double[][] basis = Projections.randomProjection(50_000, 100, 0);
double[][] proj = Projections.project(sparse, basis);
```

Here, we created some helper methods:

- `OHE.hashingEncoding` : This does one-hot-encoding of categorical data from the categorical `DataFrame`
- `Projections.randomProjection` : This generates a random basis
- `Projections.project` : This projects our sparse matrix on this generated basis

We wrote the code for these methods previously, and here we have placed them in helper methods for convenience. Of course, as usual, you can see the entire code in the code bundle provided for the chapter. So far, we have covered only one set of techniques from unsupervised learning dimensionality reduction. There is also cluster analysis, which we will cover next. Interestingly, clustering can also be used for reducing the dimensionality of the dataset, and soon we will see how.

Cluster analysis

Clustering, or cluster analysis, is another family of unsupervised learning algorithms. The goal of clustering is to organize data into clusters such that the similar items end up in the same cluster, and dissimilar items in different ones.

There are many different algorithm families for performing clustering, and they differ in how they group elements.

The most common families are as follows:

- **Hierarchical**: This organizes the dataset into a hierarchy, for example, agglomerative and divisive clustering. The result is typically a dendrogram.
- **Partitioning**: This splits the dataset into *K* disjoint classes--*K* is often specified in advance--for example, *K*-means.
- **Density-based**: This organizes the items based on density regions; if there are many items in some dense regions, they form a cluster, for example, DBSCAN.
- **Graph-based**: This represents the relations between items as a graph and applies grouping algorithms from the graph theory, for example, connected components and minimal spanning trees.

Hierarchical methods

Hierarchical methods are considered the simplest clustering algorithms; they are easy to understand and interpret. There are two families of clustering method, which belong to the hierarchical family:

- Divisive clustering algorithms
- Agglomerative clustering algorithms

In the divisive approach, we put all the data items into one cluster, and at each step we pick up a cluster and then split it into halves until every element is its own cluster. For this reason, this approach is sometimes called **top-down clustering**.

The agglomerative clustering approach is the opposite; at the beginning, each data point belongs to its own cluster, and then at each step, we select two closest clusters and merge them, until there is only one big cluster left. This is also called **bottom-up** approach.

Even though there are two types of hierarchical clustering algorithms, when people say hierarchical clustering, they typically mean agglomerative clustering, these algorithms are more common. So let's have a closer look at them.

In agglomerative clustering, at each step, we merge two closest clusters, but depending on how we define closest, the result can be quite different.

The process of merging two clusters is often called **linking**, and **linkage** describes how the distance between two clusters is calculated.

There are many types of linkages, and the most common ones are as follows:

- **Single linkage**: The distance between two clusters is the distance between two closest elements.
- **Complete linkage**: The distance between two clusters is the distance between two most distant elements.
- **Average linkage (also sometimes called UPGMA linkage)**: The distance between clusters is the distance between the centroids, where a centroid is the average across all items of the cluster.

These methods are usually suitable for a dataset of smaller sizes, and they work quite well for them. But for larger datasets, they are usually less useful and take a lot of time to finish. Still, it is possible to use it even with larger datasets, but we need to take a sample of some manageable size.

Let's look at the example. We can use the dataset of complaints we used previously, with categorical variables encoded via One-Hot-Encoding. If you remember, we then translated the sparse matrix with categorical variables into a dense matrix of smaller dimensionality by using SVD. The dataset is quite large to process, so let's first sample 10,000 records from there:

```
double[] data = ... // our data in the reduced SVD space
int size = 10000; // sample size
long seed = 0; // seed number for reproducibility
Random rnd = new Random(seed);

int[] idx = rnd.ints(0, data.length).distinct().limit(size).toArray();
double[][] sample = new double[size][];
for (int i = 0; i < size; i++) {
    sample[i] = data[idx[i]];
}

data = sample;
```

What we do here is take a stream of distinct integers from the random number generator and then limit it to 10,000. Then we use these integers as indexes for the sample.
After preparing the data and taking a sample, we can try to apply agglomerative cluster analysis to this dataset. Most of the Machine Learning libraries that we previously discussed have implementations of clustering algorithms, so we can use any of them. Since we have already used Smile quite extensively, and, in this chapter, we will also use the implementation from Smile.
When we use it, the first thing we need to specify is the **linkage**. To specify the linkage and create a `Linkage` object, we first need to compute a **proximity matrix**--a matrix that contains distances between each pair of object from the dataset.
We can use any distance measure there, but we will take the most commonly used one, the Euclidean distance. Recall that the Euclidean distance is a norm of the difference between two vectors. To efficiently compute it, we can use the following decomposition:

$$||a - b||^2 = (a-b)^T(a-b) = a^Ta - 2a^Tb + b^Tb$$

We represent the square of the distance as an inner product, and then decompose it. Next, we recognize that this is a sum of the norms of the individual vectors minus their product:

$$a^Ta - 2a^Tb + b^Tb = ||a||^2 - 2a^Tb + b^Tb$$

And this is the formula we can use for efficient calculation of the proximity matrix - the matrix of distances between each pair of items. In this formula, we have the inner product of the pair, which can be efficiently computed by using the matrix multiplication.
Let's see how to translate this formula into the code. The first two components are the individual norms, so let's compute them:

```
int nrow = data.length;

double[] squared = new double[nrow];
for (int i = 0; i < nrow; i++) {
    double[] row = data[i];

    double res = 0.0;
    for (int j = 0; j < row.length; j++) {
        res = res + row[j] * row[j];
    }

    squared[i] = res;
}
```

When it comes to the inner product, it is just a matrix multiplication of the data matrix with its transpose. We can compute it with any mathematical package in Java. For example, with Commons Math:

```
Array2DRowRealMatrix m = new Array2DRowRealMatrix(data, false);
double[][] product = m.multiply(m.transpose()).getData();
```

Finally, we put these components together to calculate the proximity matrix:

```
double[][] dist = new double[nrow][nrow];

for (int i = 0; i < nrow; i++) {
    for (int j = i + 1; j < nrow; j++) {
        double d = squared[i] - 2 * product[i][j] + squared[j];
        dist[i][j] = dist[j][i] = d;
    }
}
```

Because the distance matrix is symmetric, we can save time and loop only over the half of the indexes. There is no need to cover the case when `i == j`.

There are other distance measures we can use: it does not matter for the `Linkage` class. For example, instead of using Euclidean distance, we can take another one, for example, the cosine distance.

The cosine distance is another measure of dissimilarity between two vectors, and it is based on the cosine similarity. The cosine similarity geometrically corresponds to the angle between two vectors, and it is computed using the inner product:

$$cosine(a,b) = \frac{a^T b}{\| a \| \cdot \| b \|}$$

The inner product here is divided by the norms of each individual vector. But if the vectors are already normalized, that is, they have a norm, which is equal to 1, then the cosine is just the inner product. If the cosine similarity is equal to one, then the vectors are exactly the same.

The cosine distance is the opposite of the cosine similarity: it should be equal to zero when the vectors are the same, so we can compute it by just subtracting it from one:

$$\text{cosine-dist}(a,b) = 1 - \frac{a^T b}{\|a\| \cdot \|b\|}$$

Since we have the inner product here, it is easy to calculate this distance using the matrix multiplication.

Let's implement it. First, we do unit-normalization of each row vector of our data matrix:

```
int nrow = data.length;
double[][] normalized = new double[nrow][];

for (int i = 0; i < nrow; i++) {
    double[] row = data[i].clone();
    normalized[i] = row;
    double norm = new ArrayRealVector(row, false).getNorm();
    for (int j = 0; j < row.length; j++) {
        row[j] = row[j] / norm;
    }
}
```

Now we can multiply the normalized matrices to get the cosine similarity:

```
Array2DRowRealMatrix m = new Array2DRowRealMatrix(normalized, false);
double[][] cosine = m.multiply(m.transpose()).getData();
```

Finally, we get the cosine distance by subtracting the cosine similarity from 1:

```
for (int i = 0; i < nrow; i++) {
    double[] row = cosine[i];
    for (int j = 0; j < row.length; j++) {
        row[j] = 1 - row[j];
    }
}
```

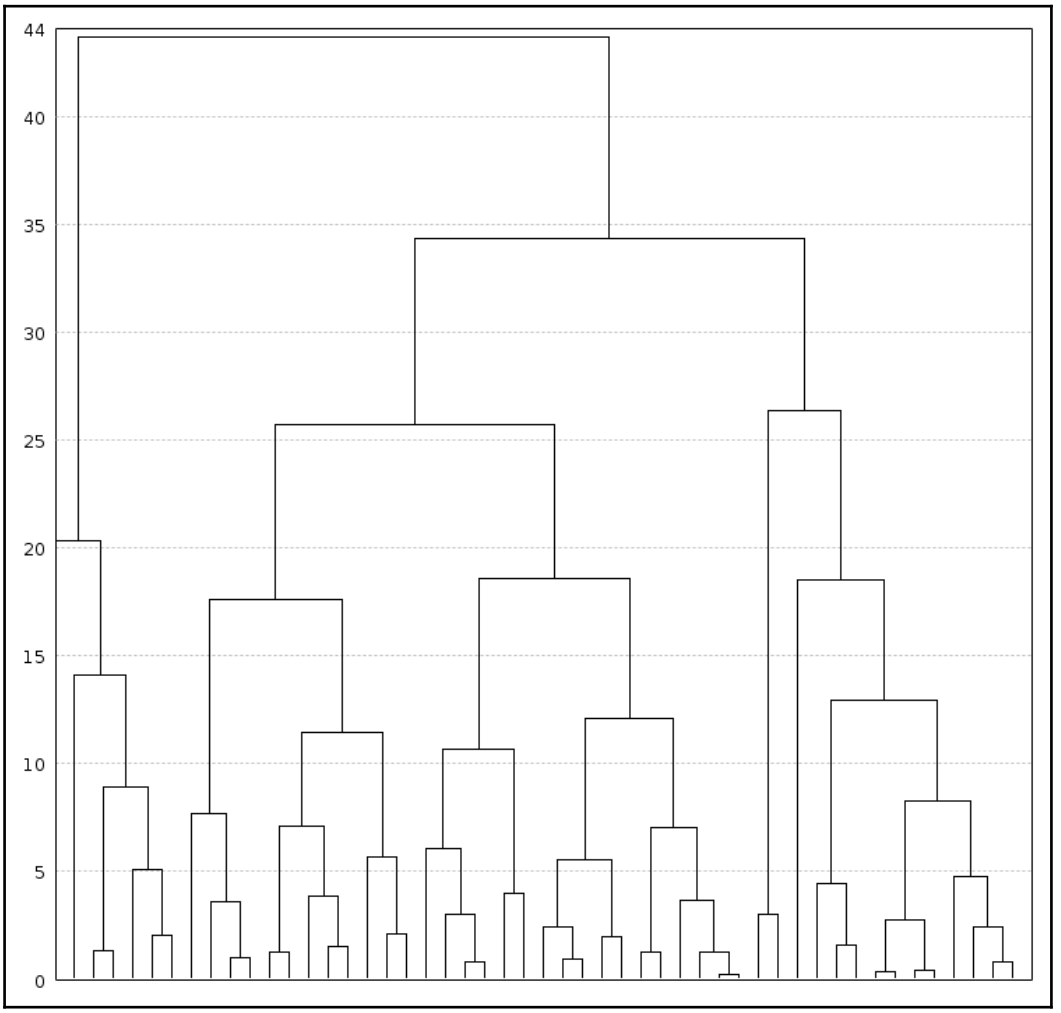
Now, it is possible to pass the computed matrix to the `Linkage` instance. As we mentioned, any distance measure can be used with hierarchical clustering, and this is a nice property, which other clustering methods often lack.

Let's now use the computed distance matrix for clustering:

```
double[][] proximity = calcualateSquaredEuclidean(data);
Linkage linkage = new UPGMALinkage(proximity);
HierarchicalClustering hc = new HierarchicalClustering(linkage);
```

In agglomerative clustering, we take two most similar clusters and merge them, and we repeat that process until there is only one cluster left. This merging process can be visualized with a dendrogram. For drawing it in Java, we can use the plotting library that comes with Smile.

To illustrate how to do it, let's first sample only a few items and apply the clustering. Then we can get something similar to the following picture:



At the bottom on the *x* axis, we have the items that are merged into clusters. On the *y* axis, we have the distance at which the clusters are merged.

To create the plot, we use the following code:

```
Frame frame = new JFrame("Dendrogram");
frame.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
```

```
PlotCanvas dendrogram = Dendrogram.plot(hc.getTree(), hc.getHeight());
frame.add(dendrogram);

frame.setSize(new Dimension(1000, 1000));
frame.setLocationRelativeTo(null);
frame.setVisible(true);
```

This visualization is quite helpful when we need to analyze the resulting clusters. Since we know the distance at which the merge was done (on the *y* axis), we can get some idea how many clusters it may make sense to extract from the data. For example, after around 21, the mergers become quite distant from each other, which may suggest that there are 5 clusters.

To get these clusters, we can cut the dendrogram at some distance threshold. If some element was merged at a distance below the threshold, then they stay within the same cluster. Otherwise, if they are merged at the distance above the threshold, they are treated as separate clusters.

For the preceding dendrogram, if we cut at the height of 23, we should get 5 separate clusters. This is how we can do it:

```
double height = 23.0;
int[] labels = hc.partition(height);
```

Alternatively, we can ask for a specific number of clusters:

```
int k = 5;
int[] labels = hc.partition(k);
```

There are several advantages of hierarchical clusters:

- It can work with any distance function, all it needs is a distance matrix, so any function can be used to create the matrix
- It is easy to come up with the number of clusters with this clustering

However, there are some disadvantages:

- It does not work well with a large number of items in the dataset--the distance matrix is hard to fit into memory.
- It is usually slower than other method, especially when some linkage is used.

There is another very popular method, which works quite well for larger datasets, and next we will talk about it.

K-means

As we mentioned previously, Agglomerative clustering methods work quite well with small datasets, but they have some problems with bigger ones. *K*-means is another popular clusterization technique, which does not suffer from this problem.

K-means is a clustering method, which belongs to the partitioning family of clustering algorithm: given the number of clusters *K*, *K*-Means splits the data into *K* disjoint groups. Grouping items into clusters is done using centroids. A centroid represents the "center" of a cluster, and for each item, we assign it to the group of its closest centroid. The quality of clustering is measured by **distortion** - the sum of distances between each item and its centroid.

As with agglomerative clustering, there are multiple implementations of *K*-Means available in Java, and, like previously, we will use the one from Smile. Unfortunately, it does not support sparse matrices, and can work only with dense ones. If we want to use it for sparse data, we either need to convert it to dense matrix, or reduce its dimensionality with SVD or random projection.

Let's again use the categorical dataset of complaints, and project it to `30` components with SVD:

```
SingularValueDecomposition svd =
SingularValueDecomposition.decompose(sparse.toSparseMatrix(), 30);
double[][] proj = Projections.project(sparse, svd.getV());
```

As we see here, the *K*-means implementation in Smile takes in four arguments:

- The matrix that we want to cluster
- The number of clusters we want to find
- The number of iterations to run
- The number of trials before selecting the best one

K-means optimizes the distortion of the dataset, and this objective function has many local optima. This means, depending on the initial configuration, you may end up with entirely different results, and some may be better than others. The problem can be mitigated by running K-means multiple times, each time with different starting position, and then choosing the clustering with the best optimum. This is why we need the last parameters, the number of trials.

Now, let's run *K*-means in Smile:

```
int k = 10;
int maxIter = 100;
```

```
int runs = 3;
KMeans km = new KMeans(proj, k, maxIter, runs);
```

Although the implementation from Smile can only work with dense matrices, the implementation from JSAT does not have this limitation, it can work with any matrix, be it dense or sparse.

The way we do it in JSAT is as follows:

```
SimpleDataSet ohe = JsatOHE.oneHotEncoding(categorical);
EuclideanDistance distance = new EuclideanDistance();
Random rand = new Random(1);
SeedSelection seedSelection = SeedSelection.RANDOM;
KMeans km = new ElkanKMeans(distance, rand, seedSelection);

List<List<DataPoint>> clustering = km.cluster(ohe);
```

In this code, we use another implementation of One-Hot-Encoding, which produces sparse JSAT datasets. It very closely follows the implementation we have for Smile. For details, you can take a look at the code in the chapter's code repository.

There are multiple implementations of *K*-Means in JSAT. One of these implementations is `ElkanKMeans`, which we used earlier. The `ElkanKMeans` parameter from JSAT is quite different from the Smile version:

- First, it takes the distance function, typically Euclidean
- It creates an instance of the random class to ensure reproducibility
- It creates the algorithm for selecting the initial seeds for clusters, with random being fastest and KPP (which is *K*-means++) being most optimal in terms of the cost function

For sparse matrices, the JSAT implementation is too slow, so it is not suitable for the problem we have. For dense matrices, the results that JSAT implementations produce are comparable to Smile, but it also takes considerably more time.

K-means has a parameter *K,* which is the number of clusters we want to have. Often, it is challenging to come up with a good value of *K*, and next we will look at how to select it.

Choosing K in K-Means

K-means has a drawback: we need to specify the number of clusters *K*. Sometimes *K* can be known from the domain problem we are trying to solve. For example, if we know that there are 10 types of clients, we probably want to look for 10 clusters.

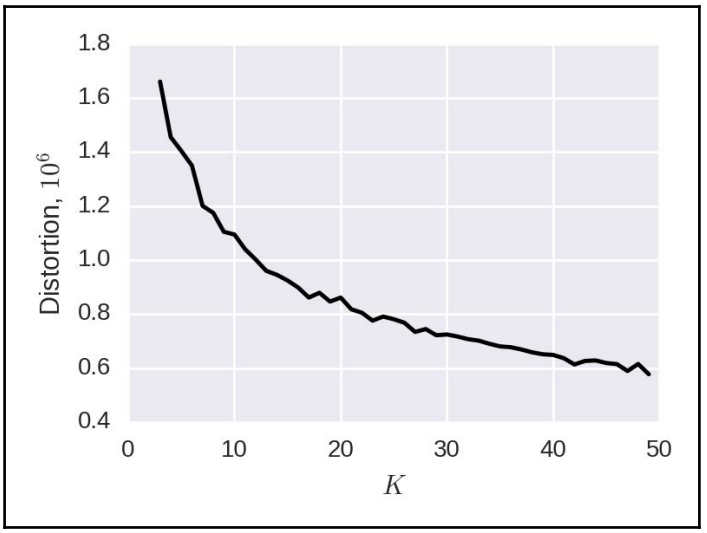
However, often we do not have this kind of domain knowledge. In situations like this, we can use a method often referred as the **elbow method**:

- Try different values of *K*, record the distortion for each
- Plot the distortion for each *K*
- Try to spot the **elbow**, the part of the graph where the error stops dropping rapidly and starts decreasing slowly

You can do it in the following way:

```
PrintWriter out = new PrintWriter("distortion.txt");

for (int k = 3; k < 50; k++) {
    int maxIter = 100;
    int runs = 3;
    KMeans km = new KMeans(proj, k, maxIter, runs);
    out.println(k + "/t" + km.distortion());
}

out.close();
```

Then, you can plot the content of the `distortion.txt` file with your favorite plotting library, and this is the result:



Here, we can see that it drops quickly initially, but then around 15-20 it starts to decrease slower. So we can select *K* from this region, for example, take *K* = 17.

Another solution would be to sample a small amount of data, and then build a dendrogram with hierarchical clustering. By looking at the dendrogram, it may become clear what is the best number of clusters *K*.

Both of these methods require human judgement and is hard to formalize. But there is another option--ask Machine Learning to choose the best *K* for us. To do it, we can use X-Means, which is an extension to the *K*-Means algorithm. X-Means tries to select the best *K* automatically using the **Bayesian Information Criterion** (**BIC**) score.

Smile already contains an implementation of X-Means, called `XMeans`, and running it is easy as follows:

```
int kmax = 300;
XMeans km = new XMeans(data, kmax);
System.out.println("selected number of clusters: " + km.getNumClusters());
```

This will output the optimal number of clusters according to BIC. JSAT also has an implementation of `XMeans`, and it works similarly.

It is never clear which approach is better, so you may need to try each of them and select the best one for a particular problem.

Apart from Agglomerative Clustering and *K*-Means, there are other clustering methods, which are also sometimes useful in practice. Next, we now look into one of them - DBSCAN.

DBSCAN

DBSCAN is another clustering quite popular technique. DBSCAN belongs to the density-based family of algorithms, and, unlike *K*-Means, it does not need to know the number of clusters, *K*, in advance.

In a few words, DBSCAN works as follows: at each step it takes an item to grow a cluster around it.

When we take an item from a high-density region, then there are many other data points close to the current item, and all these items are added to the cluster. Then the process is repeated for each newly added element of the cluster. If, however, the region is not dense enough, and there are not so many points nearby, then we do not form a cluster and say that this item is an outlier.

So, for DBSCAN to work, we need to provide the following parameters:

- The distance metric for calculating how close two items are
- The minimal number of neighbors within the radius to continue growing the cluster
- The radius around each point

As we can see, we do not need to specify *K* in advance for DBSCAN. Additionally, it naturally handles outliers, which may cause significant problems for methods like *K*-Means. There is an implementation of DBSCAN in Smile and here is how we use it:

```
double[] X = ... // data
EuclideanDistance distance = new EuclideanDistance();
int minPts = 5;
double radius = 1.0;
DBScan<double[]> dbscan = new DBScan<>(X, distance, minPts, radius);

System.out.println(dbscan.getNumClusters());
int[] assignment = dbscan.getClusterLabel();
```

In this code, we specify the following three parameters: distance, the minimum number of points around an item to be considered a cluster, and the radius.

After it is finished, we can get the cluster labels assignment using the `getClusterLabel` method. Since DBSCAN handles outliers, there is a special cluster ID for them, `Integer.MAX_VALUE`.

Agglomerative clustering, *K*-Means, and DBSCAN are one of the most commonly used clustering approaches, and they are useful when we need to group items that share some pattern. However, we can also use clustering for dimensionality reduction, and next we will see how.

Clustering for supervised learning

Like dimensionality reduction, clustering can also be useful for supervised learning.

We will talk about the following cases:

- Clustering as a feature engineering technique for creating extra features
- Clustering as a dimensionality reduction technique
- Clustering as a simple classification or regression method

Clusters as features

Clustering can be seen as a method for feature engineering, and the results of clustering can be added to a supervised model as a set of additional features.

The simplest way of doing it to use one-hot-encoding of clustering results is as follows:

- First, you run a clustering algorithm and as a result, you group the dataset into *K* clusters
- Then, you represent each datapoint as a cluster to which it belongs using the cluster ID
- Finally, you treat the IDs as a categorical feature and apply One-Hot-Encoding to it.

It looks very simple in code:

```
KMeans km = new KMeans(X, k, maxIter, runs);
int[] labels = km.getClusterLabel();

SparseDataset sparse = new SparseDataset(k);

for (int i = 0; i < labels.length; i++) {
    sparse.set(i, labels[i], 1.0);
}
```

After running it, the sparse object will contain one-hot-encoding of cluster IDs. Next, we can just append it to the existing features and run the usual supervised learning techniques on it.

Clustering as dimensionality reduction

Clustering can be seen as a special kind of dimensionality reduction. For example, if you group your data into *K* clusters, then you can compress it into *K* centroids. Once we have done it, each data point can be represented as a vector of distances to each of those centroids. If *K* is smaller than the dimensionality of your data, it can be seen as a way of reducing the dimensionality.

Let's implement this. First, let's run a *K*-means on some data. We can use the performance dataset we used previously.

We will use Smile again, and we already know how to run *K*-means. Here is the code:

```
double[][] X = ...; // data
int k = 60;
int maxIter = 10;
int runs = 1;
KMeans km = new KMeans(X, k, maxIter, runs);
```

Once it finishes, it is possible to extract the centroids for each cluster. They are stored in a two dimensional array row-wise:

```
double[][] centroids = km.centroids();
```

This should return a dataset of *K* rows (in our case, $K = 60$) with the number of columns equal to the number of features that we have in the dataset.

Next, for each observation, we can compute how far it is from each centroid. We already discussed how to implement the Euclidean distance efficiently via matrix multiplication, but previously we needed to compute the pair-wise distance between each element of the same set. Now, however, we need to compute the distance between each item from our dataset and each centroid, so we have two sets of data points. We will adapt the code slightly such that it can handle this case.

Recall the formula:

$$||a - b||^2 = ||a||^2 - 2a^T b + ||b||^2$$

We need to compute the squared norms for each vector individually and then an inner product between all items.

So, if we put all items of each set as rows of two matrices *A* and *B*, then we can use this formula to compute pair-wise distances between the two matrices via matrix multiplication.

First, we compute the norms and the product:

```
double[] squaredA = squareRows(A);
double[] squaredB = squareRows(B);

Array2DRowRealMatrix mA = new Array2DRowRealMatrix(A, false);
Array2DRowRealMatrix mB = new Array2DRowRealMatrix(B, false);
double[][] product = mA.multiply(mB.transpose()).getData();
```

Here, the `squareRows` function computes the squared norm of each row vector of the matrix:

```
public static double[] squareRows(double[][] data) {
    int nrow = data.length;

    double[] squared = new double[nrow];
    for (int i = 0; i < nrow; i++) {
        double[] row = data[i];

        double res = 0.0;
        for (int j = 0; j < row.length; j++) {
            res = res + row[j] * row[j];
        }

        squared[i] = res;
    }

    return squared;
}
```

Now we can use the formula from the preceding code to compute the distances:

```
int nrow = product.length;
int ncol = product[0].length;
double[][] distances = new double[nrow][ncol];
for (int i = 0; i < nrow; i++) {
    for (int j = 0; j < ncol; j++) {
        double dist = squaredA[i] - 2 * product[i][j] + squaredB[j];
        distances[i][j] = Math.sqrt(dist);
    }
}
```

If we wrap this into a function, for example, `distance`, we can use it like this:

```
double[][] centroids = km.centroids();
double[][] distances = distance(X, centroids);
```

Now we can use the distances array instead of the original dataset `X`, for example, like this:

```
OLS model = new OLS(distances, y);
```

Note that it does not necessarily have to be used as a dimensionality reduction technique. Instead, we can use it for engineering extra features, and just add these new features to the existing ones.

Supervised learning via clustering

Unsupervised learning can be used as a model for supervised learning, and depending on the supervised problem we have, it can be either *classification via clustering* or *regression via clustering*.

This approach is relatively straightforward. First, you associate each item with some cluster ID, and then:

- For the binary classification problem, you output the probability of seeing the positive class in the cluster
- For regression, you output the mean value across the cluster

Let's look at how we can do this for regression. At the beginning, we run *K*-means on the original data as usual:

```
int k = 250;
int maxIter = 10;
int runs = 1;

KMeans km = new KMeans(X, k, maxIter, runs);
```



Often it makes sense to pick a relative large *K*, and the optimal value, as we do usually, should be determined by cross-validation.

Next, we calculate the mean target value for each cluster from the training data. For doing this, we first group by the cluster ID, and then calculate the mean of each group:

```
double[] y = ... // target variable
int[] labels = km.getClusterLabel();

Multimap<Integer, Double> groups = ArrayListMultimap.create();
for (int i = 0; i < labels.length; i++) {
    groups.put(labels[i], y[i]);
}

Map<Integer, Double> meanValue = new HashMap<>();
for (int i = 0; i < k; i++) {
    double mean = groups.get(i).stream()
                        .mapToDouble(d -> d)
                        .average().getAsDouble();
    meanValue.put(i, mean);
}
```

Now, if we want to apply this model to test data, we can do it the following way. First, for each unseen data item, we find the closest cluster ID, and then, using this ID, we look up the mean target value.

In code it looks like this:

```
double[][] testX = ... // test data
double[] testY = ... // test target
int[] testLabels = Arrays.stream(testX).mapToInt(km::predict).toArray();

double[] testPredict = Arrays.stream(testLabels)
                             .mapToDouble(meanValue::get)
                             .toArray();
```

Now, the `testPredict` array contains the prediction for each observation from the test data.

What is more, if instead of regression, you have a binary classification problem, and you keep the labels in array of doubles, the preceding code will output probabilities of belonging to a class based on clustering, without any changes! And the `testPredict` array will contain the predicted probabilities.

Evaluation

The most complex part of unsupervised learning is evaluating the quality of models. It is very hard to objectively tell if a clustering is good or whether one result is better than another.

There are a few ways to approach this:

- Manual evaluation
- Using label information, if present
- Unsupervised metrics

Manual evaluation

Manual evaluation means looking at the results manually and using the domain expertise to assess the quality of clusters and if they make any sense.

The manual check is usually done in the following way:

- For each cluster, we sample the same data points
- Then, we look at them to see if they should belong together or not

When looking at the data, we want to ask ourselves the following questions:

- Do these items look similar?
- Does it make sense to put these items into the same group?

If the answer to both the question is yes, then the clustering results are good. Additionally, the way we sample data is also important. For example, in case of *K*-means, we should sample some items that are close to the centroid, and some items that are far away from it. Then, we can compare the ones that are close with the ones that are far. If we look at them and still can spot some similarities between them, then the clustering is good.

This sort of evaluation always makes sense even if we use other sort of cluster validation techniques, and, if possible, it should be always done to sanity-check the models. For example, if we apply it for customer segregation, we always should manually see if two customers are indeed similar within clusters, otherwise the model results will not be useful.

However, it is clear that this approach is very subjective, not reproducible, and does not scale. Unfortunately, sometimes this is the only good option, and for many problems there is no proper way to evaluate the model quality. Still, for some problems other more automatic methods can provide good results, and next we will look at some such methods.

Supervised evaluation

Manual inspection of the output is always good, but it can be quite cumbersome. Often there is some extra data, which we can use for evaluating the result of our clustering in a more automatic fashion.

For example, if we use clustering for supervised learning, then we have labels. For example, if we solve the classification problem, then we can use the class information to measure how pure (or homogeneous) the discovered clusters are. That is, we can see what is the ratio of the majority class to the rest of the classes within the cluster.

If we take the complaints dataset, there are some variables, which we did not use for clustering, for example:

- **Timely response**: This is a binary variable indicating whether the company responded to the complaint in time or not.

- **Company response to consumer**: This states what kind of response the company gave to the complaint.
- **Consumer disputed**: This states whether the customer agreed with the response or not.

Potentially, we could be interested in predicting any of these variables, so we could use them as an indication of the quality of the clustering.

For example, suppose we are interested in predicting the response of the company. So we perform the clustering:

```
int maxIter = 100;
int runs = 3;
int k = 15;
KMeans km = new KMeans(proj, k, maxIter, runs);
```

And now want to see how useful it is for predicting the response. Let's calculate the ratio of the outcomes within each cluster.

For that we first group by the cluster ID, and then calculate the ratios:

```
int[] assignment = km.getClusterLabel();
List<Object> resp = data.col("company_response_to_consumer");
Multimap<Integer, String> respMap = ArrayListMultimap.create();

for (int i = 0; i < assignment.length; i++) {
    int cluster = assignment[i];
    respMap.put(cluster, resp.get(i).toString());
}
```

Now we can print it, sorting the values within the cluster by the most frequent one:

```
List<Integer> keys = Ordering.natural().sortedCopy(map.keySet());

for (Integer c : keys) {
    System.out.print(c + ": ");

    Collection<String> values = map.get(c);
    Multiset<String> counts = HashMultiset.create(values);
    counts = Multisets.copyHighestCountFirst(counts);

    int totalSize = values.size();
    for (Entry<String> e : counts.entrySet()) {
        double ratio = 1.0 * e.getCount() / totalSize;
        String element = e.getElement();
        System.out.printf("%s=%.3f (%d), ", element, ratio, e.getCount());
    }
```

```
        System.out.println();
    }
```

This is the output for the first couple of clusters:

```
0: Closed with explanation=0.782 (12383), Closed with non-monetary
relief=0.094 (1495)...
1: Closed with explanation=0.743 (19705), Closed with non-monetary
relief=0.251 (6664)...
2: Closed with explanation=0.673 (18838), Closed with non-monetary
relief=0.305 (8536)...
```

We can see that the clustering is not really *pure:* there's one dominant class and the purity is more or less the same across the clusters. On the other hand, we see that the distribution of classes is different across clusters. For example, in cluster 2, 30% if items are *closed with non-monetary relief*, and in cluster 1, there is only 9% of them.



Even though the majority class might not be useful by itself, the distribution within each cluster can be useful for a classification model, if we use it as a feature.

This brings us to a different evaluation method; if we use the clustering as a feature engineering technique, we can evaluate the quality of the clustering by how much performance gain it gives, and select the best clustering by picking up the one with the most gain.

This brings us to the next evaluation method. If we use the results of clustering in some supervised settings (say, by using it in as a feature engineering technique), then we can evaluate the quality of the clustering by looking at how much performance it gives.

For example, we have a model that has 85% accuracy without any clustering features. Then we use two different clustering algorithms and extract the features from them, and include them to the model. The features from the first algorithm improve the score by 2%, and the second algorithm gives a 3% improvement. Then, the second algorithm is better.

Finally, there are some special metrics that we can use to assess how good the clustering is with respect to a provided label. One such metric is Rand Index and Mutual Information. These metrics are implemented in JSAT, and you will find them in the `jsat.clustering.evaluation` package.

Unsupervised Evaluation

Lastly, there are unsupervised evaluation scores for assessing the quality of clustering when no labels are known.

We already mentioned one such metric: distortion, which is the sum of distances between each item and its closest centroid. There are other metrics such as:

- Maximal pairwise distance within clusters
- Mean pairwise distance
- Sum of squared pairwise distances

These and some other metrics are also implemented in JSAT and you will find them in the `jsat.clustering.evaluation.intra` package.

Summary

In this chapter, we talked about unsupervised machine learning and about two common unsupervised learning problems, dimensionality reduction and cluster analysis. We covered the most common algorithms from each type, including PCA and K-means. We also covered the existing implementations of these algorithms in Java, and implemented some of them ourselves. Additionally, we touched some important techniques such as SVD, which are very useful in general.

The previous chapter and this chapter have given us quite a lot of information already. With these chapters, we prepared a good foundation to look at how to process textual data with machine learning and data science algorithm--and this is what we will cover in the next chapter.

6
Working with Text - Natural Language Processing and Information Retrieval

In the previous two chapters, we covered the basics of machine learning: we spoke about supervised and unsupervised problems.

In this chapter, we will take a look at how to use these methods for processing textual information, and we will illustrate most of our ideas with our running example: building a search engine. Here, we will finally use the text information from the HTML and include it into the machine learning models.

First, we will start with the basics of natural language processing, and implement some of the basic ideas ourselves, and then look into efficient implementations available in NLP libraries.

This chapter covers the following topics:

- Basics of information retrieval
- Indexing and searching with Apache Lucene
- Basics of natural language processing
- Unsupervised models for texts - dimensionality reduction, clustering, and word embeddings
- Supervised models for texts - text classification and learning to rank

By the end of this chapter you will learn how to do simple text pre-processing for machine learning, how to use Apache Lucene for indexing, how to transform words into vectors, and finally, how to cluster and classify texts.

Natural Language Processing and information retrieval

Natural Language Processing (**NLP**) is a part of computer science and computational linguistics that deals with textual data. To a computer, texts are unstructured, and NLP helps find the structure and extract useful information from them.

Information retrieval (**IR**) is a discipline that studies searching in large unstructured datasets. Typically, these datasets are texts, and the IR systems help users find what they want. Search engines such as Google or Bing are examples of such IR systems: they take in a query and provide a collection of documents ranked according to relevance with respect to the query.

Usually, IR systems use NLP for understanding what the documents are about - so later, when the user needs, these documents can be retrieved. In this chapter, we will go over the basics of text processing for information retrieval.

Vector Space Model - Bag of Words and TF-IDF

For a computer, a text is just a string of characters with no particular structure imposed on it. Hence, we call texts **unstructured data**. However, to humans, texts certainly has a structure, which we use to understand the content. What IR and NLP models try to do is similar: they find the structure in texts, use it to extract the information there, and understand what the text is about.

The simplest possible way of achieving it is called **Bag of Words**: we take a text, split it into individual words (which we call **tokens**), and then represent the text as an unordered collection of tokens along with some weights associated with each token.

Let us consider an example. If we take a document, that consists of one sentence (*we use Java for Data Science because we like Java)*, it can be represented as follows:

```
(because, 1), (data, 1), (for, 1), (java, 2), (science, 1), (use, 1), (we,
2)
```

Here, each word from the sentence is weighted by the number of times the word occurs there.

Now, when we are able to represent documents in such a way, we can use it for comparing one document to another.

For example, if we take another sentence such as *Java is good enterprise development*, we can represent it as follows:

```
(development, 1), (enterprise, 1), (for, 1), (good, 1), (java, 1)
```

We can see that there is some intersection between these two documents, which may mean that these two documents are similar, and the higher the intersection, the more similar the documents are.

Now, if we think of words as dimensions in some vector space, and weights as the values for these dimensions, then we can represent documents as vectors:

	because	data	development	enterprise	for	good	is	java	science	use	we
doc1	1	1			1			1	1	1	2
doc2			1	1	1	1	1	1			

If we take this vectorial representation, we can use the inner product between two vectors as a measure of similarity. Indeed, if two documents have a lot of common words, the inner product between them will be high, and if they share no documents, the inner product is zero.

This idea is called **Vector Space Model**, and this is what is used in many information retrieval systems: all documents as well as the user queries are represented as vectors. Once the query and the documents are in the same space, we can think of similarity between a query and a document as the relevance between them. So, we sort the documents by their similarity to the user query.

Going from raw text to a vector involves a few steps. Usually, they are as follows:

- First, we tokenize the text, that is, convert it into a collection of individual tokens.
- Then, we remove function words such as is, will, to, and others. They are often used for linking purposes only and do not carry any significant meaning. These words are called stop words.
- Sometimes we also convert tokens to some normal form. For example, we may want to map cat and cats to cat because the concept is the same behind these two different words. This is achieved through stemming or lemmatization.
- Finally, we compute the weight of each token and put them into the vector space.

Previously, we used the number of occurrences for weighting terms; this is called Term Frequency weighting. However, some words are more important than others and Term Frequency does not always capture that.

For example, *hammer* can be more important than *tool* because it is more specific. Inverse Document Frequency is a different weighting scheme that penalizes general words and favors specific ones. Inside, it is based on the number of documents that contain the term, and the idea is that more specific terms occur in fewer documents than general ones.

Finally, there is a combination of both Term Frequency and Inverse Document Frequency, which is abbreviated as TF-IDF. As the name suggests, the weight for the token `t` consists of two parts: TF and IDF:

```
weight(t) = tf(t) * idf(t)
```

Here is an explanation of the terms mentioned in the preceding equation:

- `tf(t)`: This is a function on the number of times the token `t` occurs in the text
- `idf(t)`: This is a function on the number of documents that contain the token

There are multiple ways to define these functions, but, most commonly, the following definitions are used:

- `tf(t)`: This is the number of times `t` occurs in the document
- `idf(t) = log(N / df(t))`: Here, `df(t)` is the number of documents, which contain `t`, and `N` - the total number of documents

Previously, we suggested that we can use the inner product for measuring the similarity between documents. There is a problem with this approach: it is unbounded, which means that it can take any positive value, and this makes it harder to interpret. Additionally, longer documents will tend to have higher similarity with everything else just because they contain more words.

The solution to this problem is to normalize the weights inside a vector such that its norm becomes 1. Then, computing the inner product will always result in a bounded value between 0 and 1, and longer documents will have less influence. The inner product between normalized vectors is usually called *cosine similarity* because it corresponds to the cosine of the angle that these two vectors form in the vector space.

Vector space model implementation

Now we have enough background information and are ready to proceed to the code.

First, suppose that we have a text file where each line is a document, and we want to index the content of this file and be able to query it. For example, we can take some text from `https://ocw.mit.edu/ans7870/6/6.006/s08/lecturenotes/files/t8.shakespeare.txt`and save it to `simple-text.txt`.

Then we can read it this way:

```
Path path = Paths.get("data/simple-text.txt");
List<List<String>> documents = Files.lines(path, StandardCharsets.UTF_8)
     .map(line -> TextUtils.tokenize(line))
     .map(line -> TextUtils.removeStopwords(line))
     .collect(Collectors.toList());
```

We use the `Files` class from the standard library, and then use two functions:

- The first one, `TextUtils.tokenize`, takes a String and produces a list of tokens
- The second one, `TextUtils.removeStopwords`, removes the functional words such as a, the, and so on

A simple and naive way to implement tokenization is to split a string based on a regular expression:

```
public static List<String> tokenize(String line) {
     Pattern pattern = Pattern.compile("W+");
     String[] split = pattern.split(line.toLowerCase());
     return Arrays.stream(split)
             .map(String::trim)
             .filter(s -> s.length() > 2)
             .collect(Collectors.toList());
 }
```

The expression `W+` means split the String on everything that is not a Latin character. Of course, it will fail to handle languages with non-Latin characters, but it is a fast way to implement tokenization. Also, it works quite well for English, and can be adapted to handle other European languages.

Another thing here is throwing away short tokens smaller than two characters - these tokens are often stopwords, so it is safe to discard them. The second function takes a list of tokens and removes all stopwords from it. Here is its implementation:

```
Set<String> EN_STOPWORDS = ImmutableSet.of("a", "an", "and", "are", "as",
"at", "be", ...
```

```
public static List<String> removeStopwords(List<String> line) {
     return line.stream()
             .filter(token -> !EN_STOPWORDS.contains(token))
             .collect(Collectors.toList());
 }
```

It is pretty straightforward: we keep a set of English stopwords and then for each token we just check if it is in this set or not. You can get a good list of English stopwords from `http://www.ranks.nl/stopwords`.

It is also quite easy to add token normalization to this pipeline. For now, we will skip it, but we will come back to it later in this chapter.

Now we have tokenized the texts, so the next step is to represent the tokens in the vector space. Let's create a special class for it. We will call it `CountVectorizer`.



The name `CountVectorizer` is inspired by a class with similar functionality from scikit-learn - an excellent package for doing machine learning in Python. If you are familiar with the library, you may notice that we sometimes borrow the names from there (such as names `fit()` and `transform()` for methods).

Since we cannot directly create a vector space whose dimensions are indexed by words, we will first map all distinct tokens from all the texts to some column number.

Also, it makes sense to calculate the document frequency at this step, and use it to discard tokens that appear only in a few documents. Often, such terms are misspellings, non-existing words, or too infrequent to have any impact on the results.

In code it looks like this:

```
Multiset<String> df = HashMultiset.create();
documents.forEach(list -> df.addAll(Sets.newHashSet(list)));
Multiset<String> docFrequency = Multisets.filter(df, p -> df.count(p) >=
minDf);

List<String> vocabulary =
Ordering.natural().sortedCopy(docFrequency.elementSet());
Map<String, Integer> tokenToIndex = new HashMap<>(vocabulary.size());

for (int i = 0; i < vocabulary.size(); i++) {
    tokenToIndex.put(vocabulary.get(i), i);
}
```

We use a `Multiset` from Guava to count document frequency, then we apply the filtering, with `minDf` being a parameter, which specifies the minimal document frequency. After discarding infrequent tokens, we associate a column number with each remaining one, and put this to a `Map`.

Now we can use the document frequencies to calculate IDF:

```
int numDocuments = documents.size();
double numDocumentsLog = Math.log(numDocuments + 1);
double[] idfs = new double[vocabulary.size()];

for (Entry<String> e : docFrequency.entrySet()) {
    String token = e.getElement();
    double idfValue = numDocumentsLog - Math.log(e.getCount() + 1);
    idfs[tokenToIndex.get(token)] = idfValue;
}
```

After executing it, the `idfs` array will contain the IDF part of the weight for all the tokens in our vocabulary.

Now we are ready to put the tokenized documents into a vector space:

```
int ncol = vocabulary.size();
SparseDataset tfidf = new SparseDataset(ncol);

for (int rowNo = 0; rowNo < documents.size(); rowNo++) {
    List<String> doc = documents.get(rowNo);
    Multiset<String> row = HashMultiset.create(doc);
    for (Entry<String> e : row.entrySet()) {
        String token = e.getElement();
        double tf = e.getCount();
        int colNo = tokenToIndex.get(token);
        double idf = idfs[colNo];
        tfidf.set(rowNo, colNo, tf * idf);
    }
}

tfidf.unitize();
```

Since the resulting vectors are very sparse, we use `SparseDataset` from Smile to store them. Then, for each token in a document, we compute its TF and multiply it by IDF to get the TF-IDF weights.

The last line in the code applies the length normalization to the document vectors. With this, computing the inner product between vectors will result in the Cosine Similarity score, which is a bounded value between 0 and 1.

Now, let's put the code into a class, so we can reuse it afterwards:

```
public class CountVectorizer {
    void fit(List<List<String>> documents);
    SparseDataset tranform(List<List<String>> documents);
}
```

The functions we define does the following:

- `fit` creates the mapping from tokens to column numbers and calculates the IDF
- `transform` converts a collection of documents into a sparse matrix
- The constructor should take `minDf`, which specifies the minimal document frequency for a token.

Now we can use it for vectorizing our dataset:

```
List<List<String>> documents = Files.lines(path, StandardCharsets.UTF_8)
         .map(line -> TextUtils.tokenize(line))
         .map(line -> TextUtils.removeStopwords(line))
         .collect(Collectors.toList());

int minDf = 5;
CountVectorizer cv = new CountVectorizer(minDf);
cv.fit(documents);
SparseDataset docVectors = cv.transform(documents);
```

Now imagine that we, as users, want to query this collection of documents. To be able to do it, we need to implement the following:

1. First, represent a query in the same vector space: that is, apply the exact same procedure (tokenization, stopwords removal, and so on) to the documents.
2. Then, compute the similarity between the query and each document.
3. Finally, rank the documents using the similarity score, from largest to lowest.

Suppose our query is `the probabilistic interpretation of tf-idf`. Then, map it into the vector space in a similar way:

```
List<String> query = TextUtils.tokenize("the probabilistic interpretation
of tf-idf");
query = TextUtils.removeStopwords(query);
SparseDataset queryMatrix =
vectorizer.transfrom(Collections.singletonList(query));
SparseArray queryVector = queryMatrix.get(0).x;
```

The method we created previously accepts a collection of documents, rather than a single document, so first we wrap it into a list, and then get the first row of the matrix with results.

What we have now is `docVector`, which is a sparse matrix containing our collection of documents, and `queryVector`, a sparse vector containing the query. With this, getting similarities is easy: we just need to multiply the matrix with the vector and the result will contain the similarity scores.

As in the previous chapter, we will make use of **Matrix Java Toolkit** (**MTJ**) for that. Since we are doing matrix-vector multiplication, with the matrix being on the left side, the best way of storing the values is the row-based representation. We already wrote a utility method for converting `SparseDataset` from Smile into `CompRowMatrix` from MTJ.

Here it is again:

```
public static CompRowMatrix asRowMatrix(SparseDataset dataset) {
    int ncols = dataset.ncols();
    int nrows = dataset.size();

    FlexCompRowMatrix X = new FlexCompRowMatrix(nrows, ncols);
    SparseArray[] array = dataset.toArray(new SparseArray[0]);

    for (int rowIdx = 0; rowIdx < array.length; rowIdx++) {
        Iterator<Entry> row = array[rowIdx].iterator();
        while (row.hasNext()) {
            Entry entry = row.next();
            X.set(rowIdx, entry.i, entry.x);
        }
    }

    return new CompRowMatrix(X);
}
```

Now we also need to convert a `SparseArray` object into a `SparseVector` object from MTJ.

Let's also create a method for that:

```
public static SparseVector asSparseVector(int dim, SparseArray vector) {
    int size = vector.size();
    int[] indexes = new int[size];
    double[] values = new double[size];

    Iterator<Entry> iterator = vector.iterator();
    int idx = 0;

    while (iterator.hasNext()) {
        Entry entry = iterator.next();
```

```
            indexes[idx] = entry.i;
            values[idx] = entry.x;
            idx++;
        }

        return new SparseVector(dim, indexes, values, false);
    }
```

Note that we also have to pass the dimensionality of the resulting vector to this method. This is due to a limitation of `SparseArray`, which does not store information about it.

Now we can use these methods for computing the similarities:

```
CompRowMatrix X = asRowMatrix(docVectors);
SparseVector v = asSparseVector(docVectors.ncols(), queryVector);
DenseVector result = new DenseVector(X.numRows());
X.mult(v, result);
double[] scores = result.getData();
```

The scores array now contains the cosine similarity scores of the query to each of the documents. The index of this array corresponds to the index of the original document collection. That is, to see the similarity between the query and the 10th document, we look at the 10th element of the array. So, we need to sort the array by the score, while keeping the original indexes.

Let's first create a class for it:

```
public class ScoredIndex implements Comparable<ScoredIndex> {
    private final int index;
    private final double score;

    // constructor and getters omitted

    @Override
    public int compareTo(ScoredIndex that) {
        return -Double.compare(this.score, that.score);
    }
 }
```

This class implements the `Comparable` interface, so now we can put all objects of this class into a collection, and then sort it. At the end, the first elements in the collection will have the highest score. Let's do that:

```
double minScore = 0.2;
List<ScoredIndex> scored = new ArrayList<>(scores.length);

for (int idx = 0; idx < scores.length; idx++) {
```

```
        double score = scores[idx];
        if (score >= minScore) {
            scored.add(new ScoredIndex(idx, score));
        }
    }

    Collections.sort(scored);
```

We also add a similarity threshold of 0.2 to sort fewer elements: we assume that the elements below this score are not relevant, so we just ignore them.

Finally, we can just iterate over the results and see the most relevant documents:

```
for (ScoredIndex doc : scored) {
    System.out.printf("> %.4f ", doc.getScore());
    List<String> document = documents.get(doc.getIndex());
    System.out.println(String.join(" ", document));
}
```

With this, we implemented a simple IR system ourselves, entirely from scratch. However, the implementation is quite naive. In reality, there are a quite a lot of document candidates, so it is not feasible to compute the cosine similarity of a query with every one of them. There is a special data structure called inverted index, which can be used for solving the problem, and now we will look into one of its implementations: Apache Lucene.

Indexing and Apache Lucene

Previously, we looked at how to implement a simple search engine, but it will not scale well with the number of documents.

First of all, it requires a comparison of the query with each and every document in our collection, and it becomes very time-consuming as it grows. Most of the documents, however, are not relevant to the query, and only a small fraction are. We can safely assume that, if a document is relevant to a query, it should contain at least one word from it. This is the idea behind the Inverted Index data structure: for each word it keeps track of the documents that contain it. When a query is given, it can quickly find the documents that have at least one term from it.

There also is a memory problem: at some point, the documents will no longer fit into memory, and we need to be able to store them on disk and retrieve, when needed.

Apache Lucene solves these problems: it implements a persistent Inverted Index, which is very efficient in terms of both speed and storage, and it is highly optimized and time-proven. In `Chapter 2`, *Data Processing Toolbox* we collected some raw HTML data, so let's use Lucene to build an index for it.

First, we need to include the library to our pom:

```
<dependency>
  <groupId>org.apache.lucene</groupId>
  <artifactId>lucene-core</artifactId>
  <version>6.2.1</version>
</dependency>
<dependency>
  <groupId>org.apache.lucene</groupId>
  <artifactId>lucene-analyzers-common</artifactId>
  <version>6.2.1</version>
</dependency>
<dependency>
  <groupId>org.apache.lucene</groupId>
  <artifactId>lucene-queryparser</artifactId>
  <version>6.2.1</version>
</dependency>
```

Lucene is very modular and it is possible to include only things we need. In our case this is:

- The `core` package: We always need it when using Lucene
- The `analyzers-common` module: This contains common classes for text processing
- The `queryparser`: This is the module used for parsing the query

Lucene provides several types of indexes, including in-memory and filesystem ones. We will use the filesystem one:

```
File index = new File(INDEX_PATH);
FSDirectory directory = FSDirectory.open(index.toPath());
```

Next, we need to define an Analyzer: this is a class that does all the text processing steps, including tokenization, stopwords removal, and normalization.

`StandardAnalyzer` is a basic `Analyzer`, that removes some English stopwords, but does not perform any stemming or lemmatization. It works quite well for English texts, so let's use it for building the index:

```
StandardAnalyzer analyzer = new StandardAnalyzer();
IndexWriter writer = new IndexWriter(directory, new
IndexWriterConfig(analyzer))
```

Now we are ready to index the documents!

Let's take the URLs we crawled through previously and index their content:

```
UrlRepository urls = new UrlRepository();
Path path = Paths.get("data/search-results.txt");
List<String> lines =
        FileUtils.readLines(path.toFile(), StandardCharsets.UTF_8);

for (String line : lines) {
    String[] split = line.split("t");
    String url = split[3];
    Optional<String> html = urls.get(url);
    if (!html.isPresent()) {
        continue;
    }

    org.jsoup.nodes.Document jsoupDoc = Jsoup.parse(html.get());
    Element body = jsoupDoc.body();
    if (body == null) {
        continue;
    }

    Document doc = new Document();
    doc.add(new Field("url", url, URL_FIELD));
    doc.add(new Field("title", jsoupDoc.title(), URL_FIELD));
    doc.add(new Field("content", body.text(), BODY_FIELD));
    writer.addDocument(doc);
}

writer.commit();
writer.close();
directory.close();
```

Let's have a closer look at some things here. First, `UrlRepository` is a class that stores the scraped HTML content for some URLs we created in the `Chapter 2`, *Data Processing Toolbox*. Given a URL, it returns an `Optional` object, which contains the response if the repository has the data for it; and otherwise it returns an empty `Optional`.

Then we parse the raw HTML with JSoup and extract the title and the text of the body. Now we have the text data, which we put into a Lucene `Document`.

A `Document` in Lucene consists of fields, with each `Field` object storing some information about the document. A `Field` has some properties, such as:

- Whether we store the value in the index or not. If we do, then later we can extract the content.

- Whether we index the value or not. If it is indexed, then it becomes searchable and we can query it.
- Whether it is analyzed or not. If it is, we apply the Analyzer to the content, so we can query individual tokens. Otherwise only the exact match is possible.

These and other properties are kept in the `FieldType` objects.

For example, here is how we specify the properties for `URL_FIELD`:

```
FieldType field = new FieldType();
field.setTokenized(false);
field.setStored(true);
field.freeze();
```

Here we say that we do not want to tokenize it, but want to store the value in the index. The `freeze()` method ensures that once we specify the properties, they no longer can be changed.

And here is how we specify `BODY_FIELD`:

```
FieldType field = new FieldType();
field.setStored(false);
field.setTokenized(true);
field.setIndexOptions(IndexOptions.DOCS_AND_FREQS);
field.freeze();
```

In this case, we only analyze it, but do not store the exact content of the field. This way it is still possible to query it, but since the content is not stored, the field takes less space in the index.

It quite quickly processes our dataset, and after executing it creates an index in the filesystem, which we can query. Let's do it.

```
String userQuery = "cheap used cars";

File index = new File(INDEX_PATH);
FSDirectory directory = FSDirectory.open(index.toPath());
DirectoryReader reader = DirectoryReader.open(directory);
IndexSearcher searcher = new IndexSearcher(reader);

StandardAnalyzer analyzer = new StandardAnalyzer();
AnalyzingQueryParser parser = new AnalyzingQueryParser("content",
analyzer);
Query query = parser.parse(userQuery);

TopDocs result = searcher.search(query, 10);
```

```
ScoreDoc[] scoreDocs = result.scoreDocs;

for (ScoreDoc scored : scoreDocs) {
    int docId = scored.doc;
    float luceneScore = scored.score;
    Document doc = searcher.doc(docId);
    System.out.println(luceneScore + " " + doc.get("url") + " " +
doc.get("title"));
}
```

In this code, we first open the index, then specify the analyzer for processing the query. Using this analyzer, we parse the query, and use the parsed query to extract the top 10 matching documents from the index. We stored the URL and the title, so now we can retrieve this information during the query time and present it to the user.

Natural Language Processing tools

Natural language processing is a field of computer science and computational linguistics that deals with processing texts. As we saw previously, information retrieval uses simple NLP techniques for indexing and retrieving textual information.

But NLP can do more. There are quite a few major NLP tasks, such as text summarization or machine translation, but we will not cover them and only talk about the basic ones:

- **Sentence Splitting**: Given text, we split it into sentences
- **Tokenization:** Given a sentence, split it into individual tokens
- **Lemmatization:** Given a token, we want to find its lemma. For example, for the words *cat* and *cats* the lemma is *cat.*
- **Part-of-Speech tagging (POS Tagging)**: Given a sequence of tokens, the goal is to determine what is the part-of-speech tag for each of them. For example, it means associating a tag VERB with a token like, or a tag NOUN with a token laptop.
- **Named Entity Recognition (NER)**: In a sequence of tokens, find those that correspond to named entities such as cities, countries, and other geographical names, people names, and so on. For example, it should tag Paul McCartney as a person's name, and Germany as a country name.

Let's have a look at one of the libraries that implement these basic methods: Stanford CoreNLP.

Stanford CoreNLP

There are quite a lot of mature NLP libraries in Java. For example, Stanford CoreNLP, OpenNLP, and GATE. Many libraries that we have previously covered have some NLP modules, for example, Smile or JSAT.

In this chapter, we will use Stanford CoreNLP. There is no particular reason, and it should be possible to reproduce the examples in any other library if needed.

Let's start by specifying the following dependencies in our `pom.xml`:

```
<dependency>
  <groupId>edu.stanford.nlp</groupId>
  <artifactId>stanford-corenlp</artifactId>
  <version>3.6.0</version>
</dependency>
<dependency>
  <groupId>edu.stanford.nlp</groupId>
  <artifactId>stanford-corenlp</artifactId>
  <version>3.6.0</version>
  <classifier>models</classifier>
</dependency>
```

There are two dependencies: the first one is for the NLP package itself, and the second one contains the models used by the first module. These models are for English, but there also exist models for other European languages such as German or Spanish.

The main abstraction here is a StanfordCoreNLP class, which acts as a processing pipeline. Inside it specifies a sequence of steps that are applied to the raw text.

Consider the following example:

```
Properties props = new Properties();
props.put("annotators", "tokenize, ssplit, pos, lemma");
StanfordCoreNLP pipeline = new StanfordCoreNLP(props);
```

Here we create a pipeline that takes text, tokenizes it, splits it into sentences, applies the POS model to each token, and then finds its lemma.

This is how we can use it:

```
String text = "some text";

Annotation document = new Annotation(text);
pipeline.annotate(document);
List<Word> results = new ArrayList<>();
```

```
List<CoreLabel> tokens = document.get(TokensAnnotation.class);
for (CoreLabel tokensInfo : tokens) {
    String token = tokensInfo.get(TextAnnotation.class);
    String lemma = tokensInfo.get(LemmaAnnotation.class);
    String pos = tokensInfo.get(PartOfSpeechAnnotation.class);
    results.add(new Word(token, lemma, pos));
}
```

In this code, `Word` is our class, which holds the information about tokens: the surface form (the form which appears in the text), the lemma (the normalized form) and the part of speech.

It is easy to modify the pipeline to add extra steps. For example, if we wish to add NER, then what we do is first we add `NER` to the pipeline:

```
Properties props = new Properties();
props.put("annotators", "tokenize, ssplit, pos, lemma, ner");
StanfordCoreNLP pipeline = new StanfordCoreNLP(props);
```

And then, for each token, extract the associated `NER` tag:

```
String ner = tokensInfo.get(NamedEntityTagAnnotation.class);
```

Still, the preceding code needs some manual cleaning; if we run this, we may notice that it also outputs punctuation and the stopwords. It is easy to fix by adding a few extra checks in the loop:

```
for (CoreLabel tokensInfo : tokens) {
    String token = tokensInfo.get(TextAnnotation.class);
    String lemma = tokensInfo.get(LemmaAnnotation.class);
    String pos = tokensInfo.get(PartOfSpeechAnnotation.class);
    String ner = tokensInfo.get(NamedEntityTagAnnotation.class);

    if (isPunctuation(token) || isStopword(token)) {
        continue;
    }

    results.add(new Word(token, lemma, pos, ner));
}
```

Implementation of the `isStopword` method is easy: we simply check whether the token is in the set of stopwords or not. Checking for punctuation is also not difficult:

```
public static boolean isPunctuation(String token) {
    char first = token.charAt(0);
    return !Character.isAlphabetic(first) && !Character.isDigit(first);
}
```

We just verify that the first character of the `String` is not alphabetic and not a digit. If it is the case, then it must be a punctuation.

There is another issue with NER, which we might want to fix: it does not concatenate consecutive words of the same class into one token. Consider this example: *My name is Justin Bieber, and I live in New York*. It will produce the following NER tag assignment:

- Justin -> Person
- Bieber -> Person
- New -> Location
- York -> Location
- Other tokens are mapped to `O`

We can join consecutive tokens labeled with the same `NER` tag with the following code snippet:

```
String prevNer = "O";

List<List<Word>> groups = new ArrayList<>();
List<Word> group = new ArrayList<>();

for (Word w : words) {
    String ner = w.getNer();
    if (prevNer.equals(ner) && !"O".equals(ner)) {
        group.add(w);
        continue;
    }
    groups.add(group);
    group = new ArrayList<>();
    group.add(w);

    prevNer = ner;
}

groups.add(group);
```

So we simply go through the sequence and see if the current tag is the same as previous tag or not. If it is the case, then we stop one group and start the next one. If we see `O`, then we always assume it's the next group. After that, we just need to filter empty groups and join the text fields into one, if needed.

While it does not seem a big deal for persons, it may be important for geographical names like New York: these tokens together have an entirely different meaning from separate tokens New and York, so treating them as a single token may be useful for IR systems.

Next, we will see how we can leverage NLP tools such as Stanford CoreNLP in Apache Lucene.

Customizing Apache Lucene

Apache Lucene is an old and very powerful search library. It was written back in 1999, and since then a lot of users not only have adopted it but also created many different extensions for this library.

Still, sometimes the built-in NLP capabilities of Lucene are not enough, and a specialized NLP library is needed.

For example, if we would like to include POS tags along with tokens, or find Named Entities, then we need something such as Stanford CoreNLP. It is not very difficult to include such external specialized NLP libraries in the Lucene workflow, and here we will see how to do it.

Let's use the StanfordNLP library and the tokenizer we have implemented in the previous section. We can call it `StanfordNlpTokenizer`, with one method `tokenize`, where we will put the code for tokenization we previously wrote.

We can use this class to tokenize the content of the crawled HTML data. As we did previously, we extract the text from HTML using JSoup, but now, instead of putting the title and the body directly to the document, we first preprocess it ourselves using the CoreNLP pipeline. We can do it by creating the following utility method, and then using it for tokenizing the title and the body:

```
public static String tokenize(StanfordNlpTokenizer tokenizer, String text)
{
    List<Word> tokens = tokenizer.tokenize(text);
    return tokens.stream()
                .map(Word::getLemma)
                .map(String::toLowerCase)
                .collect(Collectors.joining(" "));
}
```

Note that here we use the lemma, not the token itself, and at the end we again put everything back together to a String.

With this modification, we can use `WhitespaceAnalyzer` from Lucene. As opposed to `StandardAnalyzer`, it is very simple and all it does is split the text by a whitespace character. In our case, the String is already prepared and processed by CoreNLP, so Lucene indexes the content in the desired form.

The full modified version will look like this:

```
Analyzer analyzer = new WhitespaceAnalyzer();
IndexWriter writer =
        new IndexWriter(directory, new IndexWriterConfig(analyzer));
StanfordNlpTokenizer tokenizer = new StanfordNlpTokenizer();

for (String line : lines) {
    String[] split = line.split("t");
    String url = split[3];
    Optional<String> html = urls.get(url);
    if (!html.isPresent()) {
        continue;
    }

    org.jsoup.nodes.Document jsoupDoc = Jsoup.parse(html.get());
    Element body = jsoupDoc.body();
    if (body == null) {
        continue;
    }

    String titleTokens = tokenize(tokenizer, jsoupDoc.title());
    String bodyTokens = tokenize(tokenizer, body.text());

    Document doc = new Document();
    doc.add(new Field("url", url, URL_FIELD));
    doc.add(new Field("title", titleTokens, URL_FIELD));
    doc.add(new Field("content", bodyTokens, BODY_FIELD));
    writer.addDocument(doc);
}
```



It is possible to use Lucene's `StandardAnalyzer` for some fields, and `WhitespaceAnalyzer` with customized preprocessing for others. For that, we need to use `PerFieldAnalyzerWrapper`, where we can specify a specific `Analyzer` for each field.

This gives us a lot of flexibility in how we preprocess and analyze texts, but it does not let us change the ranking formula: the formula which Lucene uses for ordering the documents. Later in this chapter, we will also see how to do this, but first we will look at how to use machine learning in text analysis.

Machine learning for texts

Machine learning plays an important role in text processing. It allows to better understand the information hidden in the text, and extract the useful knowledge hidden there. We are already familiar with machine learning models from the previous chapters, and, in fact, we have even used some of them for texts already, for example, POS tagger and NER from Stanford CoreNLP are all machine learning based models.

In `Chapters 4`, *Supervised Learning - Clasfication and Regression* and `Chapter 5`, *Unsupervised Learning - Clustering and Dimensionality Reduction* we covered supervised and unsupervised machine learning problems. When it comes to text, both play an important role in helping to organize the texts or extract useful pieces of information. In this section, we will see how to apply them to text data.

Unsupervised learning for texts

As we know, unsupervised machine learning deals with cases when no information about labels is provided. For texts, it means just letting it process a lot of text data with no extra information about the content. Still, it often may be useful, and now we will see how to use both dimensionality reduction and clustering for texts.

Latent Semantic Analysis

Latent Semantic Analysis (**LSA**), also known as **Latent Semantic Indexing** (**LSI**), is an application of unsupervised dimensionality reduction techniques to textual data.

The problems that LSA tries to solve are the problems of:

- Synonymy: This means multiple words having the same meaning
- Polysemy: This means one word having multiple meanings

Shallow term-based techniques such as Bag of Words cannot solve these problems because they only look at the exact raw form of terms. For instance, words such as help and assist will be assigned to different dimensions of the Vector Space, even though they are very close semantically.

To solve these problems, LSA moves the documents from the usual Bag of Words Vector Space to some other Semantic Space, in which words, close in meaning, correspond to the same dimension, and the values of polysemous words are split across dimensions.

This is achieved by looking at the term-term co-occurrence matrix. The assumption is the following: if two words are often used in the same context, then they are synonymous, and vice versa, if a word is polysemous, it will be used in different contexts. Dimensionality reduction techniques can detect such co-occurrence patterns, and compress them into a vector space of smaller dimensions.

One such Dimensionality Reduction technique is **Singular Value Decomposition** (**SVD**). If X is a document-term matrix, such as the matrix we get from our `CountVectorizer`, then the SVD of X is:

$$XV = US$$

The terms in the preceding equation are explained as folliows:

- V is the basis for terms that is computed on the term-term co-occurrence matrix X^TX
- U is the basis for documents that is computed on the document-document co-occurrence matrix XX^T

So, by applying a Truncated SVD to X, we reduce the dimensionality of a term-term co-occurrence matrix X^TX, and then can use this new reduced basis V for representing our documents.

Our document matrix is stored in `SparseDataset`. If you remember, we have already used SVD on such objects: first, we transformed the SparseDataset into a column-based `SparseMatrix`, and then applied the SVD to it:

```
SparseMatrix matrix = data.toSparseMatrix();
SingularValueDecomposition svd =
SingularValueDecomposition.decompose(matrix, n);
double[][] termBasis = svd.getV();
```

And then the next step is to project our matrix onto this new term basis. We already did this in the previous chapter using the following method:

```
public static double[][] project(SparseDataset dataset, double[][] Vd) {
    CompRowMatrix X = asRowMatrix(dataset);
    DenseMatrix V = new DenseMatrix(Vd);
    DenseMatrix XV = new DenseMatrix(X.numRows(), V.numColumns());
    X.mult(V, XV);
    return to2d(XV);
}
```

Here, `asRowMatrix` converts the `SparseDataset` into a `CompRowMatrix` from MTJ, and `to2d` converts the dense matrix from MTJ to a two-dimensional array of doubles.

Once we project the original data into the LSA space, it is no longer normalized. We can fix that by implementing the following method:

```
public static double[][] l2RowNormalize(double[][] data) {
    for (int i = 0; i < data.length; i++) {
        double[] row = data[i];
        ArrayRealVector vector = new ArrayRealVector(row, false);
        double norm = vector.getNorm();
        if (norm != 0) {
            vector.mapDivideToSelf(norm);
            data[i] = vector.getDataRef();
        }
    }

    return data;
}
```

Here, we apply length normalization to each row of the input matrix, and for that we use `ArrayRealVector` from Apache Commons Math.

For convenience, we can create a special class for LSA. Let's call it `TruncatedSVD`, which will have the following signature:

```
public class TruncatedSVD {
    void fit(SparseDataset data);
    double[][] transform(SparseDataset data);
}
```

It has the following methods:

- `fit` learns the new basis for term
- `transform` reduces the dimensionality of the data by projecting it into the learned basis
- The constructor should have two parameters: `n`, the desired dimensionality and whether the result should be normalized or not

We can apply LSA to our IR system: now, instead of the cosine similarity in the Bag of Words space, we go to the LSA space and compute the cosine there. For this, we first need to map documents to this space during the indexing time, and later, during the query time, we perform the same transformation on the user queries. Then, computing the cosine is just a matrix multiplication.

So, let's first take the code we used previously:

```
List<List<String>> documents = Files.lines(path, StandardCharsets.UTF_8)
        .map(line -> TextUtils.tokenize(line))
        .map(line -> TextUtils.removeStopwords(line))
        .collect(Collectors.toList());

int minDf = 5;
CountVectorizer cv = new CountVectorizer(minDf);
cv.fit(documents);
SparseDataset docVectors = cv.transform(documents);
```

Now, we map `docVectors` to the LSA space using the `TruncatedSVD` class we just created:

```
int n = 150;
boolean normalize = true;
TruncatedSVD svd = new TruncatedSVD(n, normalize);
svd.fit(docVectors);
double[][] docsLsa = svd.transform(docVectors);
```

And we repeat the same with the query:

```
List<String> query = TextUtils.tokenize("cheap used cars");
query = TextUtils.removeStopwords(query);
SparseDataset queryVectors =
vectorizer.transfrom(Collections.singletonList(query));
double[] queryLsa = svd.transform(queryVectors)[0];
```

Like previously, we wrap the query into a list, and then extract the first row of the result. Here, however, we have a dense vector, not sparse. Now, what is left is computing the similarity, which is just a matrix-vector multiplication:

```
DenseMatrix X = new DenseMatrix(docsLsa);
DenseVector v = new DenseVector(vector);
DenseVector result = new DenseVector(X.numRows());
X.mult(v, result);
double[] scores = result.getData();
```

After executing it, the scores array will contain the similarities, and we can use the ScoredIndex class for ordering the documents by this score. This is quite useful, so let's put this into a utility method:

```
public static List<ScoredIndex> wrapAsScoredIndex(double[] scores, double
minScore) {
    List<ScoredIndex> scored = new ArrayList<>(scores.length);

    for (int idx = 0; idx < scores.length; idx++) {
        double score = scores[idx];
```

```
            if (score >= minScore) {
                scored.add(new ScoredIndex(idx, score));
            }
        }

        Collections.sort(scored);
        return scored;
    }
```

Finally, we take the first elements from the list and present them to the user, like we did previously.

Text clustering

In `Chapter 5`, *Unsupervised Learning - Clustering and Dimensionality Reduction*, we covered dimensionality reduction and clustering. We already discussed how to use dimensionality reduction for texts, but have not yet spoken about clustering.

Text clustering is also a useful technique for understanding what is a collection of documents. When we want to cluster texts, the goal is similar to non-text cases: we want to find groups of documents such that they have a lot in common: for example, the documents within such group should be on the same topic. In some cases, this can be useful for IR systems. For example, if a topic is ambiguous, we may want to group the search engine results.

K-means is a simple, yet powerful clustering algorithm, and it works quite well for texts. Let's use the crawled texts, and try to find some topics among them using *K*-means. First, we load the documents and vectorize them. We will use the *K*-Means implementation from Smile, which, if you remember, does not work with sparse matrices, so we also need to reduce the dimensionality. We will use LSA for that.

```
List<List<String>> documents = ... // read the crawl data

int minDf = 5;
CountVectorizer cv = new CountVectorizer(minDf);
cv.fit(documents);

SparseDataset docVectors = cv.transform(documents);
int n = 150;
boolean normalize = true;
TruncatedSVD svd = new TruncatedSVD(n, normalize);
svd.fit(docVectors);

double[][] docsLsa = svd.transform(docVectors);
```

The data is prepared, so we can apply *K*-means:

```
int maxIter = 100;
int runs = 3;
int k = 100;
KMeans km = new KMeans(docsLsa, k, maxIter, runs);
```

Here, `k`, as you should remember from the previous chapter, is the number of clusters we want to find. The choice of `K` here is quite arbitrary, so feel free to experiment and choose any other value of `K`.

Once it has finished, we can have a look at the resulting centroids. These centroids are, however, in the LSA space, and not in the original term space. To bring them back, we need to invert the LSA tranformation.

To go from the original space to the LSA space, we used the matrix formed by the terms basis. Thus, to do the inverse transformation, we need the inverse of that matrix. Since the basis is orthonormal, the inverse is the same as the transpose, and we will use this for inverting the LSA tranformation. This is how it looks in the code:

```
double[][] centroids = km.centroids();
double[][] termBasis = svd.getTermBasis();
double[][] centroidsOriginal = project(centroids, t(termBasis));
```

The following is how the `t` method computes the transpose:

```
public static double[][] t(double[][] M) {
    Array2DRowRealMatrix matrix = new Array2DRowRealMatrix(M, false);
    return matrix.transpose().getData();
}
```

And the project method just computes the matrix-matrix multiplication.

Now, when the centroids are in the original space, we find the most important terms of each of them.

For that, we just take a centroid and see what the largest dimensions are:

```
List<String> terms = vectorizer.vocabulary();
for (int centroidId = 0; centroidId < k; centroidId++) {
    double[] centroid = centroidsOriginal[centroidId];
    List<ScoredIndex> scored = wrapAsScoredIndex(centroid, 0.0);
    for (int i = 0; i < 20; i++) {
        ScoredIndex scoredTerm = scored.get(i);
        int position = scoredTerm.getIndex();
        String term = terms.get(position);
        System.out.print(term + ", ");
```

```
        }
        System.out.println();
    }
```

Here, terms is the list that contains the names of dimensions from the `CountVectorizer`, and `wrapAsScoredIndex` is the function we wrote previously; it takes an array of doubles, creates a list of `ScoredIndex` objects, and sorts it.

When you run it, you may see something similar to these clusters:

Cluster 1	**Cluster 2**	**Cluster 3**
Blood pressure hypotension low symptoms heart causes health disease treatment	hp printer printers printing laserjet support officejet print ink software	cars car toyota ford honda used bmw chevrolet vehicle nissan

We just took the first three clusters, and they clearly make sense. There also are some clusters which make less sense, which suggests that the algorithm could be tuned further: we can adjust *K* in *K*-Means and the number of dimensions for LSA.

Word embeddings

So far, we have covered how to apply dimensionality reduction and clustering to textual data. There is another type of unsupervised Learning, which is specific to text: word embeddings. You have probably heard about **Word2Vec,** which is one such algorithm.

The problem Word embeddings tries to solve is how to embed words into low-dimensional vector space such that semantically close words are close in this space, and different words are far apart.

For example, cat and dog should be rather close there, but laptop and sky should be quite far apart.

Here, we will implement a Word Embedding algorithm based on the co-occurrence matrix. It builds upon the ideas of LSA: there we could represent the terms by the documents they contain. So, if two words are contained in the same documents, they should be related. Document, however, is quite a broad context for a word, so we can narrow it down to a sentence, or to a few words before and after the word of interest.

For example, consider the following sentences:

We use Java for Data Science because we like Java. Java is good for enterprise development.

Then, we tokenize the text, split it into sentences, remove stopwords, and get the following:

- "we", "use", "java", "data", "science", "we", "like", "java"
- "java", "good", "enterprise", "development"

Now, suppose that for each word here we want to see what are the two words before and the two words after. This will give us the context in which each of the words is used. For this example, it will be:

- we -> use, java
- use -> we; java, data
- java -> we, use; data, science
- data -> use, java; science, we
- java -> we, like
- java -> good, enterprise
- good -> java; enterprise, development
- enterprise -> java, good; development
- development -> good, enterprise

Then, we can build a co-occurrence matrix, where we will put 1 each time a word occurs in the context of another word. So, for "we", we will add +1 to "use" and "java", and so on.

By the end, each cell will say how many times a word w_1 (from the rows of the matrix) occurred in the context of another word w_2 (from the columns of the matrix). Next, if we reduce the dimensionality of this matrix with SVD, we already get a good improvement over the plain LSA approach.

But we could go futher and replace the counts with **Pointwise Mutual Information** (**PMI**).

PMI is a measure of dependency between two random variables. It initially comes from the information theory, but it is often used in computational linguistics for measuring the degree of associations between two words. It is defined in the following way:

$$\mathtt{PMI}(w, v) = \log\left[\, p(w, v) \;/\; p(w)p(v) \,\right]$$

It checks if two words w and v co-occur by chance or not. If they happen to co-occur by chance, then the joint probability *p(w, v)* should be equal to the product of marginal probabilities *p(w) p(v)*, so PMI is 0. But if there is indeed association between two words, PMI gets values higher than 0, such that the higher the values, the stronger the association.

We typically estimate these probabilities by going through a body of text and counting:

- For marginal probabilities, we just count how many times the token occurs
- For join probabilities, we look at the co-occurrence matrix

We use the following formulas:

- `p(w) = c(w) / N`, where `c(w)` is the number of times `w` occurs in the body, and `N` is the total number of tokens
- `p(w, v) = c(w, v) / N`, where `c(w, v)` is the value from the co-occurrence matrix and `N` is the number of tokens as well

In practice, however, small values of `c(w, v)`, `c(w)`, and `c(v)` can distort the probabilities, so they are often smoothed by adding some small number `λ`:

- `p(w) = [c(w) + λ] / [N + Kλ]`, where `K` is the number of unique tokens in the corpus
- `p(w, v) = [c(w, v) + λ] / [N + Kλ]`

If we replace the PMI formula from the preceding equation, we get the following one:

$$PMI(w, v) = \log [c(w, v) + \lambda] + \log [N + K\lambda] - \log [c(w) + \lambda] - \log [c(v) + \lambda]$$

So what we can do is just replace the counts in the co-occurrence matrix with PMI and then compute the SVD of this matrix. In this case, the resulting embeddings will be of better quality.

Now, let's implement this. First, you may have noticed that we need to have the sentences, and previously we just had a stream of tokens, with no detection of sentence boundaries. As we know, Stanford CoreNLP can do it, so let's create a pipeline:

```
Properties props = new Properties();
props.put("annotators", "tokenize, ssplit, pos, lemma");
StanfordCoreNLP pipeline = new StanfordCoreNLP(props);
```

We will use the sentence splitter for detecting the sentences, and then we will take the word's lemma instead of the surface form.

But let's first create some useful classes. Previously, we used `List<List<String>>` to say that we pass a collection of documents, and each document is a sequence of tokens. Now, when we split each document into sentences, and then each sentence into tokens, it becomes `List<List<List<String>>>`, which is a bit hard to understand. We can replace this with some meaningful classes, such as `Document` and `Sentence`:

```
public class Document {
    private List<Sentence> sentences;
    // getter, setter and constructor is omitted
}

public class Sentence {
    private List<String> tokens;
    // getter, setter and constructor is omitted
}
```



Create such small classes whenever possible. Even though it may seem verbose at the beginning, it greatly helps when it comes to reading the code afterwards and understanding the intention.

Now, let's use them for tokenizing a document. We can create a `Tokenizer` class with the following method:

```
public Document tokenize(String text) {
    Annotation document = new Annotation(text);
    pipeline.annotate(document);

    List<Sentence> sentencesResult = new ArrayList<>();
    List<CoreMap> sentences = document.get(SentencesAnnotation.class);

    for (CoreMap sentence : sentences) {
        List<CoreLabel> tokens = sentence.get(TokensAnnotation.class);
        List<String> tokensResult = new ArrayList<>();

        for (CoreLabel tokensInfo : tokens) {
            String token = tokensInfo.get(TextAnnotation.class);
            String lemma = tokensInfo.get(LemmaAnnotation.class);
            if (isPunctuation(token)
                    || isStopword(token)
                    || lemma.length() <= 2) {
                continue;
            }

            tokensResult.add(lemma.toLowerCase());
        }
```

```
            if (!tokensResult.isEmpty()) {
                sentencesResult.add(new Sentence(tokensResult));
            }
        }

        return new Document(sentencesResult);
    }
```

So here we apply the sentence splitter to the text, and then, for each sentence, collect the tokens. We have already seen `isPunctuation` and `isStopword` methods - here they have the same implementation as previously.

Then we can use the crawled HTML dataset again, and apply the tokenizer on the content extracted with JSoup. We will omit this part for brevity. Now, we are ready to build the co-occurrence matrix from this data.

The first step, as in `CountVectorizer`, is to apply the document frequency filter to discard infrequent tokens, and then build a map that associates a token with some integer: the column number of the resulting sparse matrix. We know how to do it already, so we can skip this part.

Then, to estimate `p(w)` and `p(v)`, we need to know the number of times each token occurs:

```
Multiset<String> counts = HashMultiset.create();
for (Document doc : documents) {
    for (Sentence sentence : doc.getSentences()) {
        counts.addAll(sentence.getTokens());
    }
}
```

Now, we can proceed to calculating the co-occurrence matrix. For that, we can use the `Table` class from Guava:

```
Table<String, String, Integer> coOccurrence = HashBasedTable.create();
for (Document doc : documents) {
    for (Sentence sentence : doc.getSentences()) {
        processWindow(sentence, window, coOccurrence);
    }
}
```

Here, we define the `processWindow` function with the following content:

```
List<String> tokens = sentence.getTokens();

for (int idx = 0; idx < tokens.size(); idx++) {
    String token = tokens.get(idx);
```

```
        Map<String, Integer> tokenRow = coOccurrence.row(token);

        for (int otherIdx = idx - window;
                otherIdx <= idx + window;
                otherIdx++) {

            if (otherIdx < 0
                    || otherIdx >= tokens.size()
                    || otherIdx == idx) {
                continue;
            }

            String other = tokens.get(otherIdx);
            int currentCnt = tokenRow.getOrDefault(other, 0);
            tokenRow.put(other, currentCnt + 1);
        }
    }
```

Here we slide a window of a specified size over each sentence of the document. Then, for a word in the center of this window, we look at the words before and after, and for each one of them increase the co-occurrence count by 1.

The next step is to create a matrix with PMI values from this data. Like we did previously, we will use the `SparseDataset` class from Smile to keep these values:

```
int vocabularySize = vocabulary.size();

double logTotalNumTokens = Math.log(counts.size() + vocabularySize *
smoothing);
SparseDataset result = new SparseDataset(vocabularySize);

for (int rowIdx = 0; rowIdx < vocabularySize; rowIdx++) {
    String token = vocabulary.get(rowIdx);
    double logMainTokenCount = Math.log(counts.count(token) + smoothing);
    Map<String, Integer> tokenCooc = coOccurrence.row(token);

    for (Entry<String, Integer> otherTokenEntry : tokenCooc.entrySet()) {
        String otherToken = otherTokenEntry.getKey();
        double logOtherTokenCount = Math.log(counts.count(otherToken) +
smoothing);
        double logCoOccCount = Math.log(otherTokenEntry.getValue() +
smoothing);

        double pmi = logCoOccCount + logTotalNumTokens
                   - logMainTokenCount - logOtherTokenCount;

        if (pmi > 0) {
            int colIdx = tokenToIndex.get(otherToken);
```

```
                result.set(rowIdx, colIdx, pmi);
            }
        }

    }
```

In this code, we just apply the PMI formula to the co-occurrence counts we have. Finally, we perform SVD of this matrix, and for that we just use the `TruncatedSVD` class we created previously.

Now, we can see if the embedding we trained make sense. To do this, we can select some terms and, for each, find the most similar ones. This can be achieved in the following way:

- First, for a given token, we look up its vector representation
- Then, we compute the similarity of this token with the rest of the vectors. As we know, this can be done by matrix-vector multiplication
- Finally, we sort the results of the multiplication by score, and show the tokens with the highest score.

By now we have done the exact same procedure a few times, so we can skip the code. Of course, it is available in the code bundle for the chapter.

But let's anyway have a look at the results. We have selected a few words: **cat**, **germany** and **laptop**, and the following words are the most similar ones, according to the embeddings we just trained:

Cat	**Germany**	**Laptop**
0.835 pet	0.829 country	0.882 notebook
0.812 dog	0.815 immigrant	0.869 ultrabook
0.796 kitten	0.813 united	0.866 desktop
0.793 funny	0.808 states	0.865 pro
0.788 puppy	0.802 brazil	0.845 touchscreen
0.762 animal	0.789 canada	0.842 lenovo
0.742 shelter	0.777 german	0.841 gaming
0.727 friend	0.776 australia	0.836 tablet
0.727 rescue	0.760 europe	0.834 asus
0.726 picture	0.759 foreign	0.829 macbook

Even though it's not ideal, the result still makes sense. It can be improved further by training these embeddings on a lot more text data, or fine-tuning the parameters such as the dimensionality of SVD, minimal document frequency, and the amount of smoothing.



When training word embeddings, getting more data is always a good idea. Wikipedia is a good source of textual data; it is available in many languages, and they regularly publish dumps at `https://dumps.wikimedia.org/`. If Wikipedia is not enough, you can use *Common Crawl* (`http://commoncrawl.org/`), where they crawl everything on the Internet and make it available to anyone for free. We will also talk about Common Crawl in `Chapter 9`, *Scaling Data Science*.

Finally, there are a lot of pretrained word embeddings available on the Internet.

For example, you can have a look at the collection here: `https://github.com/3Top/word2vec-api`. It is quite easy to load embeddings from there.

To do this, let's first create a class to store the vectors:

```
public class WordEmbeddings {
    private final double[][] embeddings;
    private final List<String> vocabulary;
    private final Map<String, Integer> tokenToIndex;
    // constructor and getters are omitted

    List<ScoredToken> mostSimilar(String top, int topK, double
minSimilarity);
    Optional<double[]> representation(String token);
}
```

This class has the following fields and methods:

- `embeddings`: This is the array which stores the vectors
- `vocabulary`: This is the list of all the tokens
- `tokenToIndex`: This is the mapping from a token to the index at which the vector is stored
- `mostSimilar`: This returns top K other tokens most similar to the provided one
- `representation`: This returns a vector representation for a term or Optional.absent if there is no vector for it

Of course, we can put the PMI-based embeddings there. But let's see how we can load the existing GloVe and Word2Vec vectors from the preceding link.

The text file format for storing the vector is quite similar for both Word2Vec and GloVe, so we can cover only one of them. GloVe is a bit simpler, so let's use it as follows:

- First, download the pretrained embeddings from `http://nlp.stanford.edu/data/glove.6B.zip`
- Unpack it; there are several files trained on the same corpus of different dimensionality
- Let's use `glove.6B.300d.txt`

The storage format is straightforward; on each line, there is a token followed by a sequence of numbers. The numbers are obviously the embedding vector for the token. Let's read them:

```
List<Pair<String, double[]>> pairs =
        Files.lines(file.toPath(), StandardCharsets.UTF_8)
             .parallel()
             .map(String::trim)
             .filter(StringUtils::isNotEmpty)
             .map(line -> parseGloveTextLine(line))
             .collect(Collectors.toList());

List<String> vocabulary = new ArrayList<>(pairs.size());
double[][] embeddings = new double[pairs.size()][];

for (int i = 0; i < pairs.size(); i++) {
    Pair<String, double[]> pair = pairs.get(i);
    vocabulary.add(pair.getLeft());
    embeddings[i] = pair.getRight();
}

embeddings = l2RowNormalize(embeddings);
WordEmbeddings result = new WordEmbeddings(embeddings, vocabulary);
```

Here, we parse each line of the text file, then create the vocabulary list and normalize the length of the vectors. The `parseGloveTextLine` has the following content:

```
List<String> split = Arrays.asList(line.split(" "));
String token = split.get(0);
double[] vector = split.subList(1, split.size()).stream()
        .mapToDouble(Double::parseDouble).toArray();
Pair<String, double[]> result = ImmutablePair.of(token, vector);
```

Here, `ImmutablePair` is an object from Apache Commons Lang.

Let's take the same words and have a look at their neighbors using these GloVe embeddings. This is the result:

Cat	**Germany**	**Laptop**
- 0.682 dog	- 0.749 german	- 0.796 laptops
- 0.682 cats	- 0.663 austria	- 0.673 computers
- 0.587 pet	- 0.646 berlin	- 0.599 phones
- 0.541 dogs	- 0.597 europe	- 0.596 computer
- 0.490 feline	- 0.586 munich	- 0.580 portable
- 0.488 monkey	- 0.579 poland	- 0.562 desktop
- 0.473 horse	- 0.577 switzerland	- 0.547 cellphones
- 0.463 pets	- 0.575 germans	- 0.546 notebooks
- 0.461 rabbit	- 0.559 denmark	- 0.544 pcs
- 0.459 leopard	- 0.557 france	- 0.529 cellphone

The results indeed make sense, and, in some cases, it is better than the embeddings we trained ourselves with.

As we mentioned, the text format for word2vec vectors is pretty similar to the GloVe ones, so only minor modifications are needed for reading them. There is, however, a binary format for storing word2vec embeddings. It is a bit more complex, but if you would like to know how to read it, have a look at the code bundle for this chapter.

Later in this chapter, we will see how we can apply word embeddings to solve supervised learning problems.

Supervised learning for texts

Supervised machine learning methods are also quite useful for text data. Like in the usual settings, here we have the label information, which we can use to understand the information within texts.

A very common example of such application of supervised learning to texts is spam detection: every time you hit the spam button in your e-mail client, this data is collected and then put in a classifier. Then, this classifier is trained to tell apart spam versus nonspam e-mails.

In this section, we will look into how to use Supervised methods for text on two examples: first, we will build a model for sentiment analysis, and then we will use a ranking classifier for reranking search results.

Text classification

Text Classification is a problem where given a collection of texts and labels, it trains a model that can predict these labels for new unseen text. So the settings here are usual for supervised learning, except that now we have text data.

There are many possible classification problems, as follows:

- **Spam detection**: This predicts whether an e-mail is spam or not
- **Sentiment analysis**: This predicts whether the sentiment of the text is positive or negative
- **Language detection**: Given a text, this detects its language

The general workflow for text classification is similar in almost all cases:

- We tokenize and vectorize the text
- Then we fit a linear classifier treating each token as a feature

As we know that if we vectorize the text, the resulting vector is quite sparse. This is why it is a good idea to use linear models: they are very fast and can easily deal with sparsity and high dimensionality of the text data.

So let's solve one of these problems.

For example, we can take a sentiment analysis problem, and build a model, which predicts the polarity of texts, that is, whether the text is positive or negative.

We can take the data from here: `http://ai.stanford.edu/~amaas/data/sentiment/`. This dataset contains 50.000 labeled movie reviews extracted from IMDB, and the authors provide a predefined train-test split. To store the reviews from there, we can create a class for it:

```
public class SentimentRecord {
    private final String id;
    private final boolean train;
    private final boolean label;
    private final String content;
    // constructor and getters omitted
}
```

We will not go into details of the code for reading the data from the archive, but, as usual, you are welcome to check the code bundle.

As for the model, we will use LIBLINEAR--as you already know from `Chapter 4`, *Supervised Learning - Classification and Regression*. It is a library with fast implementation of linear classifiers such as logistic regression and Linear SVM.

Now, let's read the data:

```
List<SentimentRecord> data = readFromTagGz("data/aclImdb_v1.tar.gz");

List<SentimentRecord> train = data.stream()
        .filter(SentimentRecord::isTrain)
        .collect(Collectors.toList());

List<List<String>> trainTokens = train.stream()
        .map(r -> r.getContent())
        .map(line -> TextUtils.tokenize(line))
        .map(line -> TextUtils.removeStopwords(line))
        .collect(Collectors.toList());
```

Here, we read the data from the archive, and then tokenize the train data. Next, we vectorize the texts:

```
int minDf = 10;
CountVectorizer cv = new CountVectorizer(minDf);
cv.fit(trainTokens);
SparseDataset trainData = cv.transform(trainTokens);
```

So far, nothing new. But now we need to convert `SparseDataset` into the LIBLINEAR format. Let's create a couple of utility methods for this:

```
public static Feature[][] wrapX(SparseDataset dataset) {
    int nrow = dataset.size();
    Feature[][] X = new Feature[nrow][];

    int i = 0;
    for (Datum<SparseArray> inrow : dataset) {
        X[i] = wrapRow(inrow);
        i++;
    }
    return X;
}


public static Feature[] wrapRow(Datum<SparseArray> inrow) {
    SparseArray features = inrow.x;
```

```
        int nonzero = features.size();
        Feature[] outrow = new Feature[nonzero];
        Iterator<Entry> it = features.iterator();

        for (int j = 0; j < nonzero; j++) {
            Entry next = it.next();
            outrow[j] = new FeatureNode(next.i + 1, next.x);
        }

        return outrow;
    }
```

The first method, `wrapX`, takes a `SparseDataset` and creates a two-dimensional array of `Feature` objects. This is the LIBLINEAR's format for storing the data. The second method, `wrapRow`, takes one particular row of `SparseDataset`, and wraps it into a one-dimensional array of `Feature` objects.

Now, we need to extract the label information and create an instance of the `Problem` class, which describes the data:

```
double[] y = train.stream().mapToDouble(s -> s.getLabel() ? 1.0 :
0.0).toArray();
Problem problem = new Problem();
problem.x = wrapX(dataset);
problem.y = y;
problem.n = dataset.ncols() + 1;
problem.l = dataset.size();
```

Then, we define the parameters and train the model:

```
Parameter param = new Parameter(SolverType.L1R_LR, 1, 0.001);
Model model = Linear.train(problem, param);
```

Here, we specify a Logistic Regression model with L1 regularization and the cost parameter `C=1`.



Linear classifiers such as logistic regression or SVM with L1 regularization are very good for approaching high sparsity problems such as Text Classification. The L1 penalty makes sure the model converges very fast, and, additionally, it forces the solution to be sparse: that is, it performs feature selection and only keeps the most informative words.

For predicting the probability, we can create another utility method, which takes a model and a test dataset, and returns a one-dimensional array of probabilities:

```
public static double[] predictProba(Model model, SparseDataset dataset) {
    int n = dataset.size();
    double[] results = new double[n];
    double[] probs = new double[2];
    int i = 0;
    for (Datum<SparseArray> inrow : dataset) {
        Feature[] row = wrapRow(inrow);
        Linear.predictProbability(model, row, probs);
        results[i] = probs[1];
        i++;
    }

    return results;
}
```

Now we can test the model. So we take the test data, tokenize and vectorize it, and then invoke the predictProba method for checking the output. Finally, we can evaluate the performance using some evaluation metric such as AUC. In this particular case, the AUC is 0.89, which is reasonably good performance for this dataset.

Learning to rank for information retrieval

Learning to rank is a family of algorithms that deal with ordering data. This family is a part of supervised machine learning; to order the data, we need to know which items are more important and need to be shown first.

Learning to rank is often used in the context of building search engines; based on some relevance evaluations, we build a model that tries to rank relevant items higher than nonrelevant ones. In the unsupervised ranking case, such as cosine on TF-IDF weights, we typically have only one feature, by which we order the documents. However, there could be a lot more features, which we may want to include in the model and let it combine them in the best possible way.

There are several types of learning to rank models. Some of them are called Point-wise--they are applied to each document individually and consider them in isolation from the rest of the training data. Even though this is a strict assumption, these algorithms are easy to implement and they work well in practice. Typically, it amounts to using either classification or a regression model, and then order the items by the score.

Let's get back to our running example of building the search engine and include more features into it. Previously, it was unsupervised; we just ranked the items by one feature, the cosine. But we can add more features and make it a supervised learning problem.

However, for that, we need to know the labels. We already have the positive labels: for a query we know about 30 relevant documents. But we do not know the negative labels: the search engine, which we used, returns only the relevant pages. So we need to get negative examples, and then it will be possible to train a binary classifier which will tell relevant pages apart from irrelevant.

There is a technique we can use to obtain the negative data, and it is called negative sampling. This idea is based on the assumption that most of the documents in the corpus are not relevant, so if we randomly sample some of them from there and say that they are not relevant, we will be right most of the time. If a sampled document turns to be relevant, then nothing bad happens; this would be just a noisy observation which should not affect the overall result.

So, we do the following:

- First, we read the ranking data and group the documents there based on the query
- Then, we split the queries into two non-overlapping groups: one for training and one for validation
- Next, within each group, we take a query and randomly sample 9 negative examples. These URLs from the negative queries are assigned the negative label
- Finally, we train a model based on these labeled document/query pairs



At the negative sampling step, it is important that for training we do not take negative examples from the validation group and vice versa. If we sample only within the train/validation group, then we can be sure that our model generalizes well to unseen queries and documents.

Negative sampling is quite easy to implement, so let's do it:

```
private static List<String> negativeSampling(String positive, List<String>
collection,
                int size, Random rnd) {
    Set<String> others = new HashSet<>(collection);
    others.remove(positive);
    List<String> othersList = new ArrayList<>(others);
    Collections.shuffle(othersList, rnd);
    return othersList.subList(0, size);
}
```

The idea is the following: first, we take the entire collection of queries, and remove the one we are considering currently. Then, we shuffle this collection and take the top *N* ones from there.

Now that we have both positive and negative examples, we need to extract features, which we will put into the model. Let's create a `QueryDocumentPair` class, which will contain the information about the user query as well as the data about the document:

```
public class QueryDocumentPair {
    private final String query;
    private final String url;
    private final String title;
    private final String bodyText;
    private final String allHeaders;
    private final String h1;
    private final String h2;
    private final String h3;
    // getters and constructor omitted
}
```

The objects of this class can be created by parsing the HTML content with JSoup and extracting the title, the text of the body, all the header text together (h1-h6), and h1, h2, h3 headers separately.

We will use these fields for computing the features.

For example, we can compute the following ones:

- Bag-of-word TF-IDF similarity between the query and all other text fields
- LSA similarity between the query and all other text fields
- Embeddings similarity between the query and the title and h1, h2, and h3 headers.

We already know how to compute the first two types of features:

- We vectorize each of the fields separately using `CountVectorizer` and use the transform method to vectorize the query
- For LSA, we use the `TruncatedSVD` class in the same way; we train it on the text fields and then apply it to the query
- Then, we compute the cosine similarity between the text fields and the query in both Bag of Words and LSA spaces

However, we have not covered the last one here, using word embeddings. The idea is as follows:

- For query, get the vector for each token and put them together into a matrix
- For title (or other text field), do the same
- Compute the similarity of each query vector with each title vector via matrix multiplication
- Look at the distribution of similarities and take some characteristics of this distribution such as min, mean, max, and standard deviation. We can use these values as features
- Also, we can take the average query vector and the average title vector and compute the similarity between them

Let's implement this. First, create a method for getting the vectors for a collection of tokens:

```
public static double[][] wordsToVec(WordEmbeddings we, List<String> tokens)
{
    List<double[]> vectors = new ArrayList<>(tokens.size());
    for (String token : tokens) {
        Optional<double[]> vector = we.representation(token);
        if (vector.isPresent()) {
            vectors.add(vector.get());
        }
    }

    int nrows = vectors.size();
    double[][] result = new double[nrows][];
    for (int i = 0; i < nrows; i++) {
        result[i] = vectors.get(i);
    }

    return result;
}
```

Here, we use the `WordsEmbeddings` class we created previously, and then for each token we look up its representation, and if it's present, we put it into a matrix.

Then, getting all the similarities is just a multiplication of the two embedding matrices:

```
private static double[] similarities(double[][] m1, double[][] m2) {
    DenseMatrix M1 = new DenseMatrix(m1);
    DenseMatrix M2 = new DenseMatrix(m2);
    DenseMatrix M1M2 = new DenseMatrix(M1.numRows(), M2.numRows());
    M1.transBmult(M2, M1M2);
    return M1M2.getData();
```

```
}
```

As we know, MTJ stores the values of a matrix column-wise in a one-dimensional data array, and previously, we converted it to a two-dimensional array. In this case, we don't really need to do it, so we take these values as is.

Now, given a list of queries, and a list of tokens from some other field (for example, title), we compute the distribution features:

```
int size = query.size();

List<Double> mins = new ArrayList<>(size);
List<Double> means = new ArrayList<>(size);
List<Double> maxs = new ArrayList<>(size);
List<Double> stds = new ArrayList<>(size);

for (int i = 0; i < size; i++) {
    double[][] queryEmbed = wordsToVec(glove, query.get(i));
    double[][] textEmbed = wordsToVec(glove, text.get(i));
    double[] similarities = similarities(queryEmbed, textEmbed);

    DescriptiveStatistics stats = new DescriptiveStatistics(similarities);
    mins.add(stats.getMin());
    means.add(stats.getMean());
    maxs.add(stats.getMax());
    stds.add(stats.getStandardDeviation());
}
```

Of course, here we could add even more features like 25th or 75th percentiles, but these four features are enough for now. Note that sometimes either queryEmbed or textEmbed can be empty and we need to handle this case by adding multiple `NaN` instances to each list.

We also mentioned another useful feature, the similarity between the average vectors. We compute it in a similar manner:

```
List<Double> avgCos = new ArrayList<>(size);
for (int i = 0; i < size; i++) {
    double[] avgQuery = averageVector(wordsToVec(glove, query.get(i)));
    double[] avgText = averageVector(wordsToVec(glove, text.get(i)));
    avgCos.add(dot(avgQuery, avgText));
}
```

Here, the dot is the inner product between two vectors, and `averageVector` is implemented in the following way:

```
private static double[] averageVector(double[][] rows) {
    ArrayRealVector acc = new ArrayRealVector(rows[0], true);
```

```
        for (int i = 1; i < rows.length; i++) {
            ArrayRealVector vec = new ArrayRealVector(rows[0], false);
            acc.combineToSelf(1.0, 1.0, vec);
        }

        double norm = acc.getNorm();
        acc.mapDivideToSelf(norm);
        return acc.getDataRef();
    }
```

Once we have computed all these features, we can put them into an array of doubles and use it for training a classifier. There are many possible models we can choose from.

For example, we can use the Random Forest classifier from Smile: typically, tree-based methods are quite good at discovering complex interactions between features, and these methods work well for Learning to Rank tasks.

There is another thing we have not yet discussed: how to evaluate the ranking results. There are special evaluation metrics for ranking models, such as **Mean Average Precision** (**MAP**) or **Normalized Discounted Cumulative Gain** (**NDCG**), but for our current case AUC is more than sufficient. Recall that one possible interpretation of AUC is that it corresponds to the probability that a randomly-chosen positive example will be ranked higher than a randomly chosen negative one.

So, AUC fits quite well to this task, and in our experiments, a random forest model achieves the AUC of 98%. In this section, we omitted some code, but, as usual, the full code is available in the code bundle, and you can go through the feature extraction pipeline in more detail.

Reranking with Lucene

In this chapter, we already mentioned that Lucene can be customized, and we already took a look at how to do preprocessing outside of Lucene and then seamlessly integrate the results in the Lucene workflow.

When it comes to reranking search results, the situation is more or less similar. The common approach to this is taking the Lucene ranking as is and retrieve the top 100 (or more) results. Then we take these already retrieved documents and apply the ranking model to this for reordering.

If we have such a reranking model, we need to make sure that we store all the data we used for training. In our case, it was a `QueryDocumentPair` class from which we extracted the relevance features. So let's create an index:

```
FSDirectory directory = FSDirectory.open(index);
WhitespaceAnalyzer analyzer = new WhitespaceAnalyzer();
IndexWriter writer = new IndexWriter(directory, new
IndexWriterConfig(analyzer));

List<HtmlDocument> docs = // read documents for indexing

for (HtmlDocument htmlDoc : docs.) {
    String url, title, bodyText, ... // extract the field values
    Document doc = new Document();
    doc.add(new Field("url", url, URL_FIELD));
    doc.add(new Field("title", title, TEXT_FIELD));
    doc.add(new Field("bodyText", bodyText, TEXT_FIELD));
    doc.add(new Field("allHeaders", allHeaders, TEXT_FIELD));
    doc.add(new Field("h1", h1, TEXT_FIELD));
    doc.add(new Field("h2", h2, TEXT_FIELD));
    doc.add(new Field("h3", h3, TEXT_FIELD));

    writer.addDocument(doc);
}

writer.commit();
writer.close();
directory.close();
```

In this code, `HtmlDocument` is a class which stores the details about the documents-- their title, body, header, and so on. We iterate over all our documents and put this information into Lucene's index.

In this example, all the fields are stored, because, later on, during query time, we will need to retrieve these values and use them for computing the features.

So, the index is built, and now, let's query it:

```
RandomForest rf = load("project/random-forest-model.bin");

FSDirectory directory = FSDirectory.open(index);
DirectoryReader reader = DirectoryReader.open(directory);
IndexSearcher searcher = new IndexSearcher(reader);

WhitespaceAnalyzer analyzer = new WhitespaceAnalyzer();
AnalyzingQueryParser parser = new AnalyzingQueryParser("bodyText",
analyzer);
```

```
String userQuery = "cheap used cars";
Query query = parser.parse(userQuery);

TopDocs result = searcher.search(query, 100);

List<QueryDocumentPair> data = wrapResults(userQuery, searcher, result);
double[][] matrix = extractFeatures(data);
double[] probs = predict(rf, matrix);

List<ScoredIndex> scored = wrapAsScoredIndex(probs);
for (ScoredIndex idx : scored) {
    QueryDocumentPair doc = data.get(idx.getIndex());
    System.out.printf("%.4f: %s, %s%n", idx.getScore(), doc.getTitle(),
doc.getUrl());
}
```

In this code, we first read the model we previously trained and saved, and then we read the index. Next, a user gives a query, we parse it, and retrieve the top 100 results from the index. All the values we need are stored in the index, so we get them and put them into `QueryDocumentPair`+--this is what happens inside the `wrapResults` method. Then we extract the features, apply the random forest model, and use the scores for reranking the results before presenting them to the user.



At the feature extraction step, it is very important to follow the exact same procedure we used for training. Otherwise, the model results may be meaningless or misleading. The best way of achieving this is creating a special method for extracting features and use it for both training the model and during the query time.
If you need to return more than 100 results, you can perform reranking for the top 100 entires returned by Lucene, but keep the original order for 100 plus entries. In reality, users rarely go past the first page, so the probability of reaching the 100th entry is pretty less, so we usually do not need to bother with reordering the documents there.

Let's take a closer look at the content of the `wrapResults` method:

```
List<QueryDocumentPair> data = new ArrayList<>();

for (ScoreDoc scored : result.scoreDocs) {
    int docId = scored.doc;
    Document doc = searcher.doc(docId);

    String url = doc.get("url");
    String title = doc.get("title");
    String bodyText = doc.get("bodyText");
    String allHeaders = doc.get("allHeaders");
```

```
        String h1 = doc.get("h1");
        String h2 = doc.get("h2");
        String h3 = doc.get("h3");

        QueryDocumentPair pair = new QueryDocumentPair(userQuery,
                url, title, bodyText, allHeaders, h1, h2, h3);
        data.add(pair);
    }
```

Since all the fields are stored, we can get them from the index and build the `QueryDocumentPair` objects. Then we just apply the exact same procedure for feature extraction and put them into our model.

With this, we have created a search engine based on Lucene and then used a machine learning model for reranking the query results. There is a lot of room for further improvements: it can be adding more features or getting more training data, or it can be trying to use a different model. In the next chapter, we will talk about XGBoost, which also can be used for Learning to Rank tasks.

Summary

In this chapter, we covered a lot of ground in the information retrieval and NLP fields, including the basics of IR and how to apply machine learning to text. While doing this, we implemented a naive search engine first, and then used a learning to rank approach on top of Apache Lucene for an industrial-strength IR model.

In the next chapter, we will look at Gradient Boosting Machines, and at XGBoost, an implementation of this algorithm. This library provides state-of-the-art performance for many Data Science problems, including classification, regression, and ranking.

7
Extreme Gradient Boosting

By now we should have become quite familiar with machine learning and data science in Java: we have covered both supervised and unsupervised learning and also considered an application of machine learning to textual data.

In this chapter, we continue with supervised machine learning and will discuss a library which gives state-of-the-art performance in many supervised tasks: XGBoost and Extreme Gradient Boosting. We will look at familiar problems such as predicting whether a URL ranks for the first page or not, performance prediction, and ranking for the search engine, but this time we will use XGBoost to solve the problem.

The outline of this chapter is as follows:

- Gradient Boosting Machines and XGBoost
- Installing XGBoost
- XGBoost for classification
- XGBoost for regression
- XGBoost for learning to rank

By the end of this chapter, you will learn how to build XGBoost from the sources and use it for solving data science problems.

Gradient Boosting Machines and XGBoost

Gradient Boosting Machines (**GBM**) is an ensembling algorithm. The main idea behind GBM is to take some base model and then fit this model, over and over, to the data, gradually improving the performance. It is different from Random Forest models because GBM tries to improve the results at each step, while random forest builds multiple independent models and takes their average.

The main idea behind GBM can be best illustrated with a Linear Regression example. To fit several linear regressions to data, we can do the following:

1. Fit the base model to the original data.
2. Take the difference between the target value and the prediction of the first model (we call it the residuals of Step 1) and use this for training the second model.
3. Take the difference between the residuals of step 1 and predictions of step 2 (this is the residuals of Step 2) and fit the 3rd model.
4. Continue until you train *N* models.
5. For predicting, sum up the predictions of all individual models.

So, as you can see, at each step of the algorithm, the model tries to improve the results of the previous step, and by the end, it takes all the models and combines their prediction into the final one.

Essentially, any model can be used as the base model, not only Linear Regression. For example, it could be Logistic Regression or Decision Tree. Typically, tree-based models are very good and show excellent performance on a variety of problems. When we use trees in GBM, the overall model is typically called **Gradient Boosted Trees**, and depending on the type of the trees, it can be **Gradient Boosted Regression Trees** or **Gradient Boosted Classification Trees**.

Extreme Gradient Boosting, **XGBoost**, or **XGB** for short, is an implementation of Gradient Boosting Machines, and it provides a few base models, including decision trees. The tree-based XGBoost models are very powerful: they do not make any assumptions about the dataset and the distribution of values in its features, they naturally handle missing values, and they are extremely fast and can efficiently utilize all the available CPUs.

XGBoost can achieve excellent performance and can squeeze as much as possible from the data. If you know `https://www.kaggle.com/`, a website for hosting data science competitions, then you have probably already heard about XGBoost. It is the library the winners very often use in their solutions. Of course, it performs just as well outside of Kaggle and helps many Data Scientists in their daily job.

The library is originally written in C++, but there exist bindings for other languages like R and Python. Quite recently, a wrapper for Java was created as well, and in this chapter, we will see how to use it in our Java applications. This wrapper library is called **XGBoost4j,** and it is implemented via **Java Native Interface** (**JNI**) bindings, so it uses C++ underneath. But before we can use it, we need to be able to build it and install it. Now we will see how it can be done.

Installing XGBoost

As we have already mentioned, XGBoost is written in C++, and there is a Java library that allows using XGBoost in Java via JNI. Unfortunately, at the time of writing, XGBoost4J is not available on Maven Central, which means that it needs to be built locally and then published to the local Maven repository. There are plans to release the library to the central repository, and you can see the progress at `https://github.com/dmlc/xgboost/issues/1807`.

Even when it is released to Maven Central, it is still useful to know how to build it to get the bleeding edge version with the latest changes and bugfixes. So, let's see how to build the XGBoost library itself and then how to build the Java wrapper for it. For that, you can follow the official instruction from `https://xgboost.readthedocs.io/en/latest/build.html`, and here we will give an unofficial summary.

XGBoost mostly targets Linux systems, so building it on Linux is trivial:

```
git clone --recursive https://github.com/dmlc/xgboost
cd xgboost
make -j4
```

By executing the previous commands, we installed the base XGBoost library, but now we need to install the XGBoost4J bindings. To do so, perform the following sequence of steps:

- First, make sure you have the `JAVA_HOME` environment variable set and that it points to your JDK
- Then, go to the `jvm-packages` directory
- Finally, run `mvn -DskipTests install` here

The last command builds the XGBoost4J JNI bindings, then compiles the Java code and publishes everything to the local Maven repository.

Now, all we need to do for using XGBoost4J in our projects is to include the following dependency:

```
<dependency>
  <groupId>ml.dmlc</groupId>
  <artifactId>xgboost4j</artifactId>
  <version>0.7</version>
</dependency>
```

The installation process for OS X is pretty similar. However, when it comes to Windows, it is more complex.

To build it for Windows, we need to first download the 64-bit GCC compilers from `https://sourceforge.net/projects/mingw-w64/`. When installing, it is important to select `x86_64` architecture, and not `i686`, as only 64-bit platforms are supported by XGBoost. If, for some reason, the installer does not work, we can directly download the `x86_64-6.2.0-release-posix-seh-rt_v5-rev1.7z` archive with binaries from `https://goo.gl/CVcb8d`and then just unpack them.



When building XGBoost on Windows, it is important to avoid directory names with spaces there. It is therefore best to create a folder in the root, for example, `C:/soft`, and perform all the installations from there.

Next, we clone XGBoost and `make` it. Here we assume that you use the Git Windows console:

```
git clone --recursive https://github.com/dmlc/xgboost
PATH=/c/soft/mingw64/bin/:$PATH
alias make='mingw32-make'
cp make/mingw64.mk config.mk
make -j4
```

Finally, we need to build the XGBoost4J JNI binaries. You need the content of your JDK. However, there is a problem in Windows: by default, JDK is installed to the `Program Files` folder, which has a space in it, and this will cause problems during installation. One possible solution is to copy the JDK to some other place.

After doing this, we are ready to build the library:

```
export JAVA_HOME=/c/soft/jdk1.8.0_102
make jvm
cd jvm-packages
mvn -DskipTests install
```

If your Maven complains about the style and aborts the build, you can disable it by passing the `-Dcheckstyle.skip` flag:

```
mvn -DskipTests -Dcheckstyle.skip install
```

After successfully performing this step, the XGBoost4J library should be published to the local maven repository and we can use the same dependency that we used previously.

To test if a library is built correctly, try to execute this line of code:

```
Class<Booster> boosterClass = Booster.class;
```

If you see that the code terminates correctly, then you are ready to go. However if you get `UnsatisfiedLinkError` with a message similar to `xgboost4j.dll: Can't find dependent libraries`, then make sure that the `mingw64/bin` folder is on the system `PATH` variable.

XGBoost in practice

After we have successfully built and installed the library, we can use it for creating machine learning models, and in this chapter we will cover three cases: binary classification, regression, and learning to rank models. We will also talk about the familiar use cases: predicting whether a URL is on the first page or search engine results, predicting the performance of a computer, and ranking for our own search engine.

XGBoost for classification

Now let's finally use it for solving a classification problem!
In `Chapter 4`, *Supervised Learning - Classification and Regression*, we tried to predict whether a URL is likely to appear on the first page of search results or not. Previously, we created a special object for keeping the features:

```
public class RankedPage {
    private String url;
    private int position;
    private int page;
    private int titleLength;
    private int bodyContentLength;
    private boolean queryInTitle;
    private int numberOfHeaders;
    private int numberOfLinks;
    public boolean isHttps();
    public boolean isComDomain();
    public boolean isOrgDomain();
    public boolean isNetDomain();
    public int getNumberOfSlashes();
}
```

As you can see, there are a number of features, but none of them really involved text. If you remember, with these features we achieved around 0.58 AUC on a held-out test set.

As a first step, let's try to reproduce this result with XGBoost. Because this is a binary classification, we set the objective parameter to `binary:logistic`, and since the last time we used AUC for evaluation, we will stick to this choice and set `eval_metric` to `auc` as well. We set the parameters via a map:

```
Map<String, Object> params = new HashMap<>();
params.put("objective", "binary:logistic");
params.put("eval_metric", "logloss");
params.put("nthread", 8);
params.put("seed", 42);
params.put("silent", 1);

// default values:
params.put("eta", 0.3);
params.put("gamma", 0);
params.put("max_depth", 6);
params.put("min_child_weight", 1);
params.put("max_delta_step", 0);
params.put("subsample", 1);
params.put("colsample_bytree", 1);
params.put("colsample_bylevel", 1);
params.put("lambda", 1);
params.put("alpha", 0);
params.put("tree_method", "approx");
```

Here, most of the parameters are set to their default values, with the exception of objective, `eval_metric`, `nthread`, `seed`, and `silent`.

As you see, XGBoost is a very configurable implementation of the Gradient Boosting Machines algorithm, and there are a lot of parameters that we can change. We will not include all the parameters here; you can refer to the official documentation at `https://github.com/dmlc/xgboost/blob/master/doc/parameter.md`for the full list. In this chapter, we will only use tree-based methods, so let's review some of their parameters:

Parameter name	**Range**	**Description**
`nthread`	1 and more	This is the number of threads to use when building the trees
`eta`	from 0 to 1	This is the weight of each model in the ensemble
`max_depth`	1 and more	This is the maximal depth of each tree
`min_child_weight`	1 and more	This is the minimal number of observations per leaf
`subsample`	from 0 to 1	This is the fraction of observations to be used at each step

`colsample_bytree`	from 0 to 1	This is the fraction of features to be used at each step
`objective`		This defines the task (regression or classification)
`eval_metric`		This is the evaluation metric for the task
`seed`	integer	This sets the seed for reproducibility
`silent`	0 or 1	Here, 1 turns off the debugging output during training

Then, we read the data and create the train, validation, and test sets. We already have special functions for this, which we will use here as well:

```
Dataset dataset = readData();

Split split = dataset.trainTestSplit(0.2);
Dataset trainFull = split.getTrain();
Dataset test = split.getTest();

Split trainSplit = trainFull.trainTestSplit(0.2);
Dataset train = trainSplit.getTrain();
Dataset val = trainSplit.getTest();
```



Previously, we applied the Standardization (or Z-Score transformation) to our data. For tree-based algorithms, including XGBoost, this is not required: these methods are insensitive to such monotone transformations, so we can skip this step.

Next, we need to wrap our dataset into XGBoost's internal format: `DMatrix`. Let's create a utility method for this:

```
public static DMatrix wrapData(Dataset data) throws XGBoostError {
    int nrow = data.length();
    double[][] X = data.getX();
    double[] y = data.getY();

    List<LabeledPoint> points = new ArrayList<>();

    for (int i = 0; i < nrow; i++) {
        float label = (float) y[i];
        float[] floatRow = asFloat(X[i]);
        LabeledPoint point = LabeledPoint.fromDenseVector(label, floatRow);
        points.add(point);
    }

    String cacheInfo = "";
    return new DMatrix(points.iterator(), cacheInfo);
}
```

Now we can use it for wrapping the datasets:

```
DMatrix dtrain = XgbUtils.wrapData(train);
DMatrix dval = XgbUtils.wrapData(val);
```

XGBoost gives us a convenient way to monitor the performance of our model via the so-called watchlist. In essence, this is analogous to learning curves, where we can see how the evaluation metric evolves at each step. If, during training, we see that the values of training and evaluation metrics diverge significantly, then it may indicate that we are likely to overfit. Likewise, if at some step the validation metric starts decreasing while the train metric values keep increasing, then we overfit.

A watchlist is defined via a map where we associate some name with every dataset we are interested in:

```
Map<String, DMatrix> watches = ImmutableMap.of("train", dtrain, "val",
dval);
```

Now we are ready to train an XGBoost model:

```
int nrounds = 30;
IObjective obj = null;
IEvaluation eval = null;
Booster model = XGBoost.train(dtrain, params, nrounds, watches, obj, eval);
```

It is possible to provide custom objective and evaluation functions in XGBoost, but since we only use the standard ones, these parameters are set to null.

As we discussed, the training process can be monitored via a watchlist, and this is what you will see during the training process: at each step it will compute the evaluation function on the provided datasets and output the values to the console:

```
[0]    train-auc:0.735058    val-auc:0.533165
[1]    train-auc:0.804517    val-auc:0.576641
[2]    train-auc:0.842617    val-auc:0.561298
[3]    train-auc:0.860178    val-auc:0.567264
[4]    train-auc:0.875294    val-auc:0.570171
[5]    train-auc:0.888918    val-auc:0.575836
[6]    train-auc:0.896271    val-auc:0.573969
[7]    train-auc:0.904762    val-auc:0.577094
[8]    train-auc:0.907462    val-auc:0.580005
[9]    train-auc:0.911556    val-auc:0.580033
[10]    train-auc:0.922488    val-auc:0.575021
[11]    train-auc:0.929859    val-auc:0.579274
[12]    train-auc:0.934084    val-auc:0.580852
[13]    train-auc:0.941198    val-auc:0.577722
[14]    train-auc:0.951749    val-auc:0.582231
```

```
[15]    train-auc:0.952837    val-auc:0.579925
```

If, during training, we want to build a lot of trees, then digesting the text output from the console is hard, and it often helps to visualize these curves. In our case, however, we only had 30 iterations, so it is possible to make some judgment of the performance. If we take a careful look, we may notice that in step 8 the validation score stopped increasing, while the train score was still getting better and better. The conclusion that we can make from this is that at some point it starts overfitting. To avoid that, we can only use the first nine trees when making predictions:

```
boolean outputMargin = true;
int treeLimit = 9;
float[][] res = model.predict(dval, outputMargin, treeLimit);
```

Note two things here:

- If we set `outputMargin` to false, the un-normalized values will be returned, not probabilities. Setting it to true will apply the logistic transformation to the values, and it will make sure that the results look like probabilities.
- The results are a two-dimensional array of floats, not a one-dimensional array of doubles.

Let's write a utility function for transforming these results into doubles:

```
public static double[] unwrapToDouble(float[][] floatResults) {
    int n = floatResults.length;
    double[] result = new double[n];
    for (int i = 0; i < n; i++) {
        result[i] = floatResults[i][0];
    }
    return result;
}
```

Now we can use other methods we developed previously, for example, a method for checking AUC:

```
double[] predict = XgbUtils.unwrapToDouble(res);
double auc = Metrics.auc(val.getY(), predict);
System.out.printf("auc: %.4f%n", auc);
```

If we do not specify the number of trees in predict, then it uses all the available trees and performs the normalization of values by default:

```
float[][] res = model.predict(dval);
double[] predict = unwrapToDouble(res);
double auc = Metrics.auc(val.getY(), predict);
System.out.printf("auc: %.4f%n", auc);
```

In the previous chapters, we have created some code for K-Fold cross-validation. We can use it here as well:

```
int numFolds = 3;
List<Split> kfold = trainFull.kfold(numFolds);
double aucs = 0;

for (Split cvSplit : kfold) {
    DMatrix dtrain = XgbUtils.wrapData(cvSplit.getTrain());

    Dataset validation = cvSplit.getTest();
    DMatrix dval = XgbUtils.wrapData(validation);
    Map<String, DMatrix> watches = ImmutableMap.of("train", dtrain, "val",
dval);

    Booster model = XGBoost.train(dtrain, params, nrounds, watches, obj,
eval);
    float[][] res = model.predict(dval);
    double[] predict = unwrapToDouble(res);

    double auc = Metrics.auc(validation.getY(), predict);
    System.out.printf("fold auc: %.4f%n", auc);
    aucs = aucs + auc;
}

aucs = aucs / numFolds;
System.out.printf("cv auc: %.4f%n", aucs);
```

However, XGBoost has built-in capabilities for performing Cross-Validation: all we need to do is to provide `DMatrix`, and then it will split the data and run the evaluation automatically. Here is how we can use it:

```
DMatrix dtrainfull = wrapData(trainFull);
int nfold = 3;
String[] metric = {"auc"};
XGBoost.crossValidation(dtrainfull, params, nrounds, nfold, metric, obj,
eval);
```

And we will see the following evaluation log:

```
[0]    cv-test-auc:0.556261    cv-train-auc:0.714733
[1]    cv-test-auc:0.578281    cv-train-auc:0.762113
[2]    cv-test-auc:0.584887    cv-train-auc:0.792096
[3]    cv-test-auc:0.592273    cv-train-auc:0.824534
[4]    cv-test-auc:0.593516    cv-train-auc:0.841793
[5]    cv-test-auc:0.593855    cv-train-auc:0.856439
[6]    cv-test-auc:0.593967    cv-train-auc:0.875119
[7]    cv-test-auc:0.588910    cv-train-auc:0.887434
[8]    cv-test-auc:0.592887    cv-train-auc:0.897417
[9]    cv-test-auc:0.589738    cv-train-auc:0.906296
[10]   cv-test-auc:0.588782    cv-train-auc:0.915271
[11]   cv-test-auc:0.586081    cv-train-auc:0.924716
[12]   cv-test-auc:0.586461    cv-train-auc:0.935201
[13]   cv-test-auc:0.584988    cv-train-auc:0.940725
[14]   cv-test-auc:0.586363    cv-train-auc:0.945656
[15]   cv-test-auc:0.585908    cv-train-auc:0.951073
```

After we chose the best parameters (the number of trees in this case), we can retrain the model on the entire train part of the data and then evaluate it on the test:

```
int bestNRounds = 9;
Map<String, DMatring> watches = Collections.singletonMap("dtrainfull",
dtrainfull);

Booster model = XGBoost.train(dtrainfull, params, bestNRounds, watches,
obj, eval);

DMatrix dtest = XgbUtils.wrapData(test);
float[][] res = model.predict(dtest);
double[] predict = XgbUtils.unwrapToDouble(res);

double auc = Metrics.auc(test.getY(), predict);
System.out.printf("final auc: %.4f%n", auc);
```

Finally, we can save the model and use it afterwards:

```
Path path = Paths.get("xgboost.bin");
try (OutputStream os = Files.newOutputStream(path)) {
    model.saveModel(os);
}
```

Reading the saved model is also simple:

```
Path path = Paths.get("xgboost.bin");
try (InputStream is = Files.newInputStream(path)) {
     Booster model = XGBoost.loadModel(is);
}
```



These models are compatible with other XGBoost bindings. So, we can train a model in Python or R, and then import it into Java - or the other way round.

Here, we used only the default parameters, which are often not ideal. Let's look at how we can modify them to achieve the best performance.

Parameter tuning

So far we have discussed three ways to perform Cross-Validation with XGBoost: hold-out dataset, manual K-Fold, and XGBoost K-Fold. Any of these ways can be used for selecting the best performance.

The implementations from XGBoost are typically better-suited for this task because they can show the performance at each step, and the training process can be manually stopped once you see that the learning curves diverge too much.

If your dataset is relatively large (for example, more that 100k examples), then simply selecting a hold-out dataset may be the best and fastest option. On the other hand, if your dataset is smaller, it may be a good idea to perform the *K*-Fold Cross-Validation.

Once we have decided on the validation scheme, we can start tuning the model. Since XGBoost has a lot of parameters, it is quite a complex problem because it is not computationally feasible to try all the possible combinations. There are, however, a few approaches that may help to achieve reasonably good performance.

The general approach is to change one parameter at a time and then run the training process with a watchlist. When doing so, we closely monitor the validation values and take a note of the largest ones. Finally, we select the combination of parameters, which gives the best validation performance. If two combinations give comparable performance, then we should opt for the simpler one (for example, less deep, with more instances in leaves, and so on).

Here is one such algorithm for tuning the parameters:

1. For the start, select a very large value for the number of trees, like 2,000 or 3,000. Never grow all these trees and stop the training process when you see that the validation scores stop growing or start decreasing.
2. Take the default parameters and change one at a time.
3. If your dataset is quite small, it may make sense to pick a smaller `eta` at the beginning, for example, 0.1. If the dataset is big enough, then the default value is fine.
4. First, we tune the `depth` parameter. Train the model with the default value (6), then try with small value (3) and large value (10). Depending on which one performs better, move in the appropriate direction.
5. Once the tree depth is settled, try changing the `subsample` parameter. First, try the default value (1) and then try decreasing it to 0.8, 0.6, and 0.4, and then move it to the appropriate direction. Typically, values around 0.6-0.7 work reasonably well.
6. Next, tune `colsample_bytree`. The approach is the same as for subsample, and values around 0.6-0.7 also work quite well.
7. Now, we tune `min_child_weight`. You can try values such as 10, 20, 50, and then move to the appropriate direction.
8. Finally, set `eta` to some small value (such as 0.01, 0.05, or 0.1 depending on the size of the dataset) and see what is the iteration number where the validation performance stops increasing. Use this number for selecting the number of iterations for the final model.

There are alternative ways of doing this. For example:

- Initialize `depth` to 10, `eta` to 0.1, and `min_child_weight` to 5
- As previously, first find the best `depth` by trying smaller and larger values
- Then, tune the `subsample` parameter
- After that, tune `min_child_weight`
- The last parameter to tune is `colsample_bytree`
- Finally, we set `eta` to a smaller number and watch the validation performance to select the number of trees

These are simple heuristics and do not touch many available parameters, but they can, nonetheless, give a reasonably good model. You can also tune the regularization parameters such as `gamma`, `alpha`, and `beta`. For example, for high `depth` values (more than 10), you may want to increase the `gamma` parameter a bit and see what happens.

Unfortunately, none of these algorithms gives a 100% guarantee to find the best solution, but you should try them and find the one you personally like the most - which probably be a combination of these ones, or maybe even something completely different.



If you do not have a lot of data and do not want to tune the parameters manually, then try setting the parameters randomly, repeat it multiple times, and select the best model based on Cross-Validation. This is called Random Search Parameter Optimization: it does not require hand tuning and often works well in practice.

It may seem very overwhelming at the beginning, so do not worry. After doing it several times, you will develop some intuition as to how these parameters depend on each other and what is the best way to tune them.

Text features

In the previous chapter, we learned a lot of things that could be applied to text data and used some of the ideas when building the search engine. Let's take these features, include them into our model, and see how our AUC changes.

Recall that we previously created these features:

- Cosine similarity in the TF-IDF space between the query and the text fields of the documents, such as the title, the body content, and the h1, h2, and h3 headers
- LSA similarity between the query and all other text fields

We also used GloVe features in the previous chapter, but we will skip them here.In addition we won't include the implementation of the previous features in this chapter. For information on how to do it, refer to `Chapter 6`, *Working with Text - Natural Language Processing and Information Retrieval*.

Once we have added the features, we can play with parameters a bit. For example, we can end up using these parameters:

```
Map<String, Object> params = XgbUtils.defaultParams();
params.put("eval_metric", "auc");
params.put("colsample_bytree", 0.5);
params.put("max_depth", 3);
params.put("min_child_weight", 30);
params.put("subsample", 0.7);
params.put("eta", 0.01);
```

Here, `XgbUtils.defaultParams()` is a helper function, which creates a map with some parameters set to their default values, and then we can modify some of them. For example, since the performance is not very good and it is easy to overfit here, we grow smaller trees of depth 3 and ask for at least 30 observations in leaf nodes. Finally, we set the learning rate parameter `eta` to a small value because the dataset is not very large.

With these features, we can now achieve the AUC of 64.4%. This is very far from good performance, but it's a 5% improvement over the previous version with no features, which is a considerable step forward.

To avoid repetition, we have omitted a lot of code. If you feel a bit lost, you are always welcome to check the chapter's code bundle for details.

Feature importance

Finally, we can also see which features contribute most to the model, which are less important, and order our features according to their performance. XGBoost implements one such feature's important measure called **FScore,** which is the number of times a feature is used by the model.

To extract FScore, we first need to create a feature map: a file that contains the names of the features:

```
List<String> featureNames = columnNames(dataframe);
String fmap = "feature_map.fmap";
try (PrintWriter printWriter = new PrintWriter(fileName)) {
    for (int i = 0; i < featureNames.size(); i++) {
        printWriter.print(i);
        printWriter.print('t');
        printWriter.print(featureNames.get(i));
        printWriter.print('t');
        printWriter.print("q");
        printWriter.println();
    }
}
```

In this code, we first call a function, `columnNames` (not present here), which extracts the column names from a joinery dataframe. Then, we create a file where at each line we first print the feature name, and then a letter `q`, which means that the feature is quantitative and not an `i` - indicator.

Then, we call a method called `getFeatureScore`, which takes the feature map file and returns the feature's importance in a map. After getting it, we can sort the entries of the map according to their values, and this will produce a list of features ranked by their importance:

```
Map<String, Integer> scores = model.getFeatureScore(fmap);
Comparator<Map.Entry<String, Integer>> byValue =
Map.Entry.comparingByValue();
scores.entrySet().stream().sorted(byValue.reversed()).forEach(System.out::p
rintln);
```

For the classification model with text features, it will produce the following output:

```
numberOfLinks=17
queryBodyLsi=15
queryTitleLsi=14
bodyContentLength=13
numberOfHeaders=10
queryBodySimilarity=10
urlLength=7
queryTitleSimilarity=6
https=3
domainOrg=1
numberOfSlashes=1
```

We see that these new features are quite important to the model. We also see that features such as `domainOrg` or `numberOfSlashes` are rarely used, and many of the features we included are not even on this list. This means that we can safely exclude these features from our model and re-train the model without them.



FScore is not the only feature-importance measure available for tree-based methods, but the XGBoost library provides only this score. There are external libraries such as XGBFI (`https://github.com/Far0n/xgbfi`), which can use the model dump for calculating metrics such as Gain, Weighted FScore, and others, and often these scores are more informative.

XGBoost is good not only for classification purposes, but also shines when it comes to Regression. Next, we will see how to use XGBoost for it.

XGBoost for regression

Gradient Boosting is quite a general model: it can deal with both classification and regression tasks. To use it to solve the regression problem all we need to do is to change the objective and the evaluation metric.

For binary classification, we used the `binary:logistic` objective, but for regression, we just change it to `reg:linear`. When it comes to evaluation, there are the following built-in evaluation metrics:

- Root-Means-Square Error (set `eval_metric` to `rmse`)
- Mean Absolute Deviation (set `eval_metric` to `mae`)

Apart from these changes, the other parameters for tree-based models are exactly the same! We can follow the same approach for tuning the parameters, except that now we will monitor a different metric.

In `Chapter 4`, *Supervised Learning - Classification and Regression*, we used the matrix multiplication performance data for illustrating the regression problem. Let's take the same dataset again, and this time use XGBoost for building the model.

To speed things up, we can take the reduced dataset from `Chapter 5`, *Unsupervised Learning - Clustering and Dimensionality Reduction*. However, in `Chapter 6`, *Working with Text - Natural Language Processing and Information Retrieval*, we have created a special class for SVD: `TruncatedSVD`. So, let's use it for reducing the dimensionality of this dataset:

```
Dataset dataset = ... // read the data
StandardizationPreprocessor preprocessor =
StandardizationPreprocessor.train(dataset);
dataset = preprocessor.transform(dataset);

Split trainTestSplit = dataset.shuffleSplit(0.3);
Dataset allTrain = trainTestSplit.getTrain();
Split split = allTrain.trainTestSplit(0.3);
Dataset train = split.getTrain();
Dataset val = split.getTest();

TruncatedSVD svd = new TruncatedSVD(100, false)
svd.fit(train);

train = dimred(train, svd);
val = dimred(val, svd);
```

You should remember from `Chapter 5`, *Unsupervised Learning - Clustering and Dimensionality Reduction* , that if we are going to reduce the dimensionality of the dataset with PCA via SVD, we need to standardize the data before that, and the following is what happens right after we read the data. We do the usual train-validation-test split and reduce the dimensionality of all the datasets. The `dimred` function just wraps calling the `transform` method from SVD and then it puts the results back to a `Dataset` class.

Now, let's use XGBoost:

```
DMatrix dtrain = XgbUtils.wrapData(train);
DMatrix dval = XgbUtils.wrapData(val);
Map<String, DMatrix> watches = ImmutableMap.of("train", dtrain, "val",
dval);
IObjective obj = null;
IEvaluation eval = null;

Map<String, Object> params = XgbUtils.defaultParams();
params.put("objective", "reg:linear");
params.put("eval_metric", "rmse");
int nrounds = 100;

Booster model = XGBoost.train(dtrain, params, nrounds, watches, obj, eval);
```

Here, we wrap our datasets into `DMatrix`, then create a watchlist, and finally set the `objective` and `eval_metric` parameters to the appropriate ones. Now we can train the model.

Let's look at the watchlist output (for brevity, we will only show every 10th record here):

```
[0]    train-rmse:21.223036    val-rmse:18.009176
[9]    train-rmse:3.584128    val-rmse:5.860992
[19]    train-rmse:1.430081    val-rmse:5.104758
[29]    train-rmse:1.117103    val-rmse:5.004717
[39]    train-rmse:0.914069    val-rmse:4.989938
[49]    train-rmse:0.777749    val-rmse:4.982237
[59]    train-rmse:0.667336    val-rmse:4.976982
[69]    train-rmse:0.583321    val-rmse:4.967544
[79]    train-rmse:0.533318    val-rmse:4.969896
[89]    train-rmse:0.476646    val-rmse:4.967906
[99]    train-rmse:0.422991    val-rmse:4.970358
```

We can see that the validation error stopped decreasing around the 50th tree and then started increasing again. So, let's limit the model to 50 trees and apply this model to the test data:

```
DMatrix dtrainall = XgbUtils.wrapData(allTrain);
watches = ImmutableMap.of("trainall", dtrainall);
nrounds = 50;
model = XGBoost.train(dtrainall, params, nrounds, watches, obj, eval);
```

Then, we can apply this model to the test data and see the final performance:

```
Dataset test = trainTestSplit.getTest();
double[] predict = XgbUtils.predict(model, test);
double testRmse = rmse(test.getY(), predict);
System.out.printf("test rmse: %.4f%n", testRmse);
```

Here, `XgbUtils.predict` converts a dataset into `DMatrix`, then calls the predict method and finally converts the array of floats into doubles. After executing the code, we will see the following:

```
test rmse: 4.2573
```

Recall that previously it was around 15, so with XGBoost it is more than three times better than with a linear regression!

Note that in the original dataset there are categorical variables, and when we use One-Hot-Encoding (via the `toModelMatrix` method from the joinery data frame), the resulting matrix is sparse. In addition, we then compress this data with PCA. However, XGBoost can also deal with sparse data, so let's use this example to illustrate how to do it.

In `Chapter 5`, *Unsupervised Learning - Clustering and Dimensionality Reduction*, we created a class for performing One-Hot-Encoding: we used it for converting categorical variables into an object of the `SparseDataset` class from Smile. Now we can use this method for creating such `SparseDataset`, and then for constructing a `DMatrix` object for XGBoost from it.

So, let's create a method for converting `SparseDataset` into `DMatrix`:

```
public static DMatrix wrapData(SparseDataset data) {
    int nrow = data.size();
    List<LabeledPoint> points = new ArrayList<>();

    for (int i = 0; i < nrow; i++) {
        Datum<SparseArray> datum = data.get(i);
        float label = (float) datum.y;
        SparseArray array = datum.x;

        int size = array.size();
        int[] indices = new int[size];
        float[] values = new float[size];

        int idx = 0;
        for (Entry e : array) {
            indices[idx] = e.i;
            values[idx] = (float) e.x;
            idx++;
        }
```

```
            LabeledPoint point =
                    LabeledPoint.fromSparseVector(label, indices, values);
            points.add(point);
        }

        String cacheInfo = "";
        return new DMatrix(points.iterator(), cacheInfo);
    }
```

Here, the code is pretty similar to what we used for dense matrices, but now we call the `fromSparseVector` factory method instead of `fromDenseVector`. To use it, we convert each row of `SparseDataset` into an array of indexes and array of values and then use them to create a `LabeledPoint` instance, which we use to create a `DMatrix` instance.

After converting it, we run the XGBoost model on it:

```
SparseDataset sparse = readData();
DMatrix dfull = XgbUtils.wrapData(sparse);

Map<String, Object> params = XgbUtils.defaultParams();
params.put("objective", "reg:linear");
params.put("eval_metric", "rmse");

int nrounds = 100;
int nfold = 3;
String[] metric = {"rmse"};
XGBoost.crossValidation(dfull, params, nrounds, nfold, metric, null, null);
```

When we run this, we see that RMSE reaches 17.549534 and never goes down after that. This is expected, since we a small subset of features; these features are all categorical and not all of them are very informative. Still, this serves as a good illustration of how we can use XGBoost for sparse datasets.

Apart from Classification and Regression, XGBoost also provides special support for creating ranking models, and now we will see how we can use it.

XGBoost for learning to rank

Our search engine has become quite powerful. Previously, we used Lucene for the fast retrieval of documents and then used a machine learning model for reordering them. By doing this, we were solving a ranking problem. After being given a query and a collection of documents, we need to order all the documents such that the ones that are the most relevant to the query have the highest rank.

Previously, we approached this problem as a classification: we built a binary classification model to separate relevant and non-relevant documents, and we used the probability of a document being relevant for sorting. This approach works reasonably well in practice, but has a limitation: it only considers one element at a time and keeps other documents in complete isolation. In other words, when deciding whether a document is relevant, we look only at the features of this particular document and do not look at the features of other documents.

What we can do instead is to look at the positions of the documents in relation to each other. Then, for each query, we can form a group of documents, which we consider to this particular query, and optimize the ranking within all such groups.

LambdaMART is the name of a model that uses this idea. It looks at the pairs of documents and considers the relative order of the documents within the pair. If the order is wrong (an irrelevant document ranks higher than a relevant one), then the model introduces a penalty, and during training we want to make this penalty as small as possible.

MART in LambdaMART stands for **Multiple Additive Regression Trees**, so it is a tree-based method. XGBoost also implements this algorithm. To use it, we set the objective to `rank:pairwise` and then we set the evaluation measure to one of the following:

- `ndcg`: This stands for Normalized Discounted Cumulative Gain
- `ndcg@n`: NDCG at *N* is the first *N* elements of the list and evaluates NDCG on it
- `map`: This stands for Mean Average Precision
- `map@n`: This is MAP evaluated at the first *N* elements of each group

For our purposes, it is not important to know in detail what these metrics do; for now, it is enough to know that the higher the value of a metric, the better it would be. However, there is an important difference between these two metrics: MAP can only deal with binary (0/1) labels, but NDCG can work with ordinal (0, 1, 2, ...) labels.

When we built a classifier, we only had two labels: positive (`1`) and negative (`0`). It may make sense to extend the labels to include more degrees of relatedness. For example, we can assign the labels in the following fashion:

- First, 3 URLs get a relevance of 3
- Other URLs on the first page get a relevance of 2
- The remaining relevant URLs on the second and third page get a relevance of 1
- And all non-relevant documents are labeled with 0

As we have mentioned, NDCG can deal with such ordinal labels, so we will use it for evaluation. To implement this relevance assignment, we can take the `RankedPage` class we used previously and create the following method:

```
private static int relevanceLabel(RankedPage page) {
    if (page.getPage() == 0) {
        if (page.getPosition() < 3) {
            return 3;
        } else {
            return 2;
        }
    }

    return 1;
}
```

We can use this method for all the documents within one query, and for all the other documents, we just assign a relevance of 0. Apart from this method, the remaining code for creating and extracting features stays the same, so for brevity, we will omit the code.

Once the data is prepared, we wrap `Dataset` it into `DMatrix`. When doing this, we need to specify the groups, and within each we will optimize the ranking. In our case, we group the data by the query.

XGBoost expects the objects belonging to the same group to be in a consecutive order, so it needs an array of group sizes. For example, suppose in our dataset we have 12 objects: 4 from group 1, 3 from group 2, and 5 from group 3:

qid	1	1	1	1	2	2	2	3	3	3	3	3
size	4				3			5				

Then, the size array should contain the sizes of these groups: `[4, 3, 5]`.

Here, `qid` is the ID of a query: an integer, which we put in association with each query:

qid	query	url
1	adidas basketball shoes	http://www.adidas.com/us/men-basketball-shoes
1	adidas basketball shoes	http://www.adidas.com/us/basketball
1	adidas basketball shoes	http://www.eastbay.com/adidas/Basketball/Shoes/_-_/N-x8Z1e1Zne
2	angry birds	https://www.angrybirds.com/
2	angry birds	http://chrome.angrybirds.com/
2	angry birds	http://www.youtube.com/channel/UCYC2wjLop-S6Ld4raeoUVNA
2	angry birds	http://www.rovio.com/index.php?page=angry-birds
3	animal shelter	http://www.animalshelter.org/
3	animal shelter	https://www.petfinder.com/animal-shelters-and-rescues
3	animal shelter	https://www.animalhouseshelter.com/pets/berlin/
3	animal shelter	https://www.petfinder.com/
3	animal shelter	http://www.adoptapet.com/adoption_rescue/80291.html

Let's first create a utility function for calculating the size arrays:

```
private static int[] groups(List<Integer> queryIds) {
    Multiset<Integer> groupSizes = LinkedHashMultiset.create(queryIds);
    return groupSizes.entrySet().stream().mapToInt(e ->
e.getCount()).toArray();
}
```

This method takes in a list of query IDs and then it counts how many times each ID is present there. For that, we use `LinkedHashMultiset` - a multiset from Guava. This particular implementation of the multiset remembers the order in which the elements were inserted, so when getting back the counts the order is preserved.

Now we can specify the group sizes for both datasets:

```
DMatrix dtrain = XgbUtils.wrapData(trainDataset);
int[] trainGroups = queryGroups(trainFeatures.col("queryId"));
dtrain.setGroup(trainGroups);

DMatrix dtest = XgbUtils.wrapData(testDataset);
int[] testGroups = queryGroups(testFeatures.col("queryId"));
dtest.setGroup(testGroups);
```

And we are ready to train a model:

```
Map<String, DMatrix> watches = ImmutableMap.of("train", dtrain, "test",
dtest);
IObjective obj = null;
IEvaluation eval = null;

Map<String, Object> params = XgbUtils.defaultParams();
```

```
params.put("objective", "rank:pairwise");
params.put("eval_metric", "ndcg@30");

int nrounds = 500;
Booster model = XGBoost.train(dtrain, params, nrounds, watches, obj, eval);
```

Here, we change the objective to `rank:pairwise`, because we are interested in solving the ranking problem.We also set the evaluation metrics to `ndcg@30`, which means that we want to look only at NDCG of the first 30 documents, and do not really care about the documents we have after 30. The reason for this is that the users of search engines rarely look at the second and third pages of the search results, and it is very unlikely that they will look past the third page, and so we only consider the first three pages of the search results. That is, we are interested only in the top 30 documents, so we only look at NDCG at 30.

As we did previously, we start with default parameters and go through the same parameter tuning procedure as we do for classification or regression.

We can tune it a bit, for example, using the following parameters:

```
Map<String, Object> params = XgbUtils.defaultParams();
params.put("objective", "rank:pairwise");
params.put("eval_metric", "ndcg@30");
params.put("colsample_bytree", 0.5);
params.put("max_depth", 4);
params.put("min_child_weight", 30);
params.put("subsample", 0.7);
params.put("eta", 0.02);
```

With this set of parameters, we see that the best NDCG@30 of 0.632 for the hold-out data is reached at about the 220th iteration, so we should not grow more than 220 trees.

Now we can save the model with XGBoost model dumper and use it in Lucene. For that, we need to use the same code as we did previously with almost no changes; the only thing we need to change is the model. That is, the only difference is that instead of loading a random forest model, we need to load the XGBoost model. After that, we just follow the same procedure: we retrieve the top 100 documents with Lucene and rerank them with the new XGBoost model.

Therefore, with XGBoost, we are able to take into consideration the relative order of documents within each query group, and use this information to improve the model further.

Summary

In this chapter, we learned about Extreme Gradient Boosting --an implementation of Gradient Boosting Machines. We learned how to install the library and then we applied to solve a variety of supervised learning problems: classification, regression, and ranking.

XGBoost shines when the data is structured: when it is possible to extract good features from our data and put these features into a tabular format. However, in some cases, the data is quite hard to structure. For example, when dealing with images or sounds, a lot of effort is needed to extract useful features. But we do not necessarily have to do the feature extraction ourselves, instead, we can use Neural Network models which can learn the best features themselves.

In the next chapter, we will look at deeplearning4j--a deep learning library for Java.

8
Deep Learning with DeepLearning4J

In the previous chapter, we covered **Extreme Gradient Boosting** (**XGBoost**)--a library that implements the gradient boosting machine algorithm. This library provides state-of-the-art performance for many supervised machine learning problems. However, XGBoost only shines when the data is already structured and there are good handmade features.

The feature engineering process is usually quite complex and requires a lot of effort, especially when it comes to unstructured information such as images, sounds, or videos. This is the area where deep learning algorithms are usually superior to others, including XGBoost; they do not need hand-crafted features and are able to learn the structure of the data themselves.

In this chapter, we will look into a deep learning library for Java--DeepLearning4J. This library allows us to easily specify complex neural network architectures that are able to process unstructured data such as images. In particular, we will look into Convolutional Neural Networks--a special kind of neural network that is well-suited for images.

This chapter will cover the following:

- ND4J--the engine behind DeepLearning4J
- Simple neural networks for handwritten digit recognition
- Deep networks with convolutional layers for digit recognition
- A model for classifying images with dogs and cats

By the end of this chapter, you will learn how to run DeepLearning4J, apply it to image recognition problems, and use AWS and GPUs to speed it up.

Neural Networks and DeepLearning4J

Neural Networks are typically good models that give a reasonable performance on structured datasets, but they might not necessarily be better than others. However, when it comes to dealing with unstructured data, most often they are the best.

In this chapter, we will look into a Java library for designing Deep Neural Networks, called DeepLearning4j. But before we do this, we first will look into its *backend*--ND4J, which does all the number crunching and heavy lifting.

ND4J - N-dimensional arrays for Java

DeepLearning4j relies on ND4J for preforming linear algebra operations such as matrix multiplication. Previously, we covered quite a few such libraries, for example, Apache Commons Math or Matrix Toolkit Java. Why do we need yet another linear algebra library?

There are two reasons for this. First, these libraries usually deal only with vectors and matrices, but for deep learning we need tensors. A **tensor** is a generalization of vectors and matrices to multiple dimensions; we can see vectors as one-dimensional tensors and matrices as two-dimensional ones. For deep learning, this is important because we have images, which are three-dimensional; not only do they have height and width, but also multiple channels.

Another quite important reason for ND4J is its GPU support; all the operations can be executed on the graphical processors, which are designed to handle a lot of complex linear algebra operations in parallel, and this is extremely helpful for speeding up the training of neural networks.

So, before going into DeepLearning4j, let's quickly go over some basics of ND4J, even if it is not so important to know the specifics of how Deep Neural Networks are implemented, it can be useful for other purposes.

As usual, we first need to include the dependency on the `pom` file:

```
<dependency>
  <groupId>org.nd4j</groupId>
  <artifactId>nd4j-native-platform</artifactId>
  <version>0.7.1</version>
</dependency>
```

This will download the CPU version for Linux, MacOS, or Windows, depending on your platform. Note that for Linux you might need to have OpenBLAS installed. It is usually very easy, for example, for Ubuntu Linux, you can install it by executing the following command:

```
sudo apt-get install libopenblas-dev
```

After including the library to the `pom` file and installing dependencies, we are ready to start using it.



ND4J's interface is heavily inspired by NumPy, a numerical library for Python. If you already know NumPy, you will quickly recognize familiarities in ND4J.

Let's begin by creating ND4J arrays. Suppose, we want to create a `5 x 10` array filled with ones (or with zeros). This is quite simple, for that, we can use the *ones* and *zeros* utility methods from the `Nd4j` class:

```
INDArray ones = Nd4j.ones(5, 10);
INDArray zeros = Nd4j.zeros(5, 10);
```

If we already have an array of doubles, then wrapping them into `Nd4j` is easy:

```
Random rnd = new Random(10);
double[] doubles = rnd.doubles(100).toArray();
INDArray arr1d = Nd4j.create(doubles);
```

When creating an array, we can specify the resulting shape. Suppose we want to take this array with `100` elements and put it into a `10 x 10` matrix. All we need to do is specify the shape when creating the array:

```
INDArray arr2d = Nd4j.create(doubles, new int[] { 10, 10 });
```

Alternatively, we can reshape the array after creating it:

```
INDArray reshaped = arr1d.reshape(10, 10);
```

Any array of any dimensionality can be reshaped into a one-dimensional array with the `reshape` method:

```
INDArray reshaped1d = reshaped.reshape(1, -1);
```

Note that we use `-1` here; this way we ask ND4J to automatically infer the right number of elements.

If we have a two-dimensional Java array of doubles, then there is a special syntax for wrapping them into ND4J:

```
double[][] doubles = new double[3][];
doubles[0] = rnd.doubles(5).toArray();
doubles[1] = rnd.doubles(5).toArray();
doubles[2] = rnd.doubles(5).toArray();
INDArray arr2d = Nd4j.create(doubles);
```

Likewise, we can create a three-dimensional ND4J array from doubles:

```
double[] doubles = rnd.doubles(3 * 5 * 5).toArray();
INDArray arr3d = Nd4j.create(doubles, new int[] { 3, 5, 5 });
```

So far, we used Java's `Random` class for generating random numbers, but we can use ND4J's method for this:

```
int seed = 0;
INDArray rand = Nd4j.rand(new int[] { 5, 5 }, seed);
```

What is more, we can specify a distribution from which we will sample the values:

```
double mean = 0.5;
double std = 0.2;
INDArray rand = Nd4j.rand(new int[] { 3, 5, 5 }, new
NormalDistribution(mean, std));
```

As we mentioned previously, three-dimensional tensors are useful for representing images. Typically, an image is a three-dimensional array where the dimensions are the number of `channels * height * width`, and the values typically range from 0 to 255.

Let's generate an image-like array of size `2 * 5` with three channels:

```
double[] picArray = rnd.doubles(3 * 2 * 5).map(d -> Math.round(d *
255)).toArray();
INDArray pic = Nd4j.create(picArray).reshape(3, 2, 5);
```

If we print this array, we will see something like the following:

```
[[[51.00, 230.00, 225.00, 146.00, 244.00],
  [64.00, 147.00, 25.00, 12.00, 230.00]],
[[145.00, 160.00, 57.00, 202.00, 143.00],
  [170.00, 91.00, 181.00, 94.00, 92.00]],
[[193.00, 43.00, 248.00, 211.00, 27.00],
  [68.00, 139.00, 115.00, 44.00, 97.00]]]
```

Here, the output is first grouped by channels, and inside we have the pixel value information of each channel separately. To get a specific channel only, we can use the `get` method:

```
for (int i = 0; i < 3; i++) {
    INDArray channel = pic.get(NDArrayIndex.point(i));
    System.out.println(channel);
}
```

Alternatively, if we are interested in all the rows of the 0th channels with columns from 2nd to 3rd, we can use the `get` method for accessing this specific part of the array in this way:

```
INDArray slice = pic.get(NDArrayIndex.point(0), NDArrayIndex.all(),
NDArrayIndex.interval(2, 4));
System.out.println(slice);
```

The following is the output:

```
[[225.00, 146.00],
 [25.00, 12.00]]
```

This library has a lot more things such as dot product, matrix multiplication, and so on. This functionality is quite similar to what we have already covered in detail for analogous libraries, so we will not repeat ourselves here.

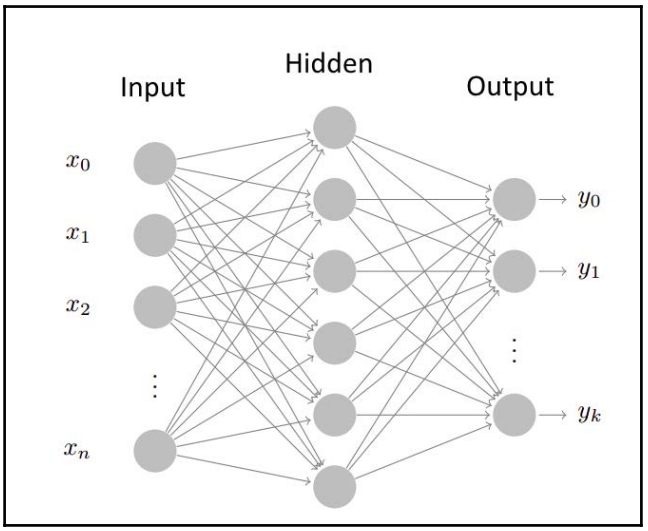
Now, let's start with neural networks!

Neural networks in DeepLearning4J

After learning some basics of ND4J, we are now ready to start using DeepLearning4j and create neural networks with it.

As you probably know already, neural networks are models where we stack individual neurons in layers. During the prediction phase, each neuron gets some input, processes it, and forwards the results to the next layer. We start from the input layer, which receives the raw data, and gradually push the values forward to the output layer, which will contain the prediction of the model for the given input.

A neural network with one hidden layer might look like this:



DeepLearning4J allows us to easily design such networks. If we take the network from the preceding figure and try to implement it with DeepLearning4j, we might end up with the following:

```
DenseLayer input = new DenseLayer.Builder().nIn(n).nOut(6).build();
nnet.layer(0, input);
OutputLayer output = new OutputLayer.Builder().nIn(6).nOut(k).build();
nnet.layer(1, output);
```

As you see, it is not difficult to read and understand. So, let's use it; for that, we first need to specify its dependency to the `pom.xml` file:

```
<dependency>
  <groupId>org.deeplearning4j</groupId>
  <artifactId>deeplearning4j-core</artifactId>
  <version>0.7.1</version>
</dependency>
```

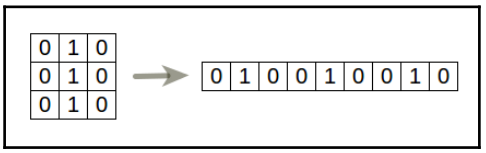
Note that the versions of DeepLearning4j and ND4J must be the same.

For the illustration, we will use the MNIST dataset; this dataset contains images of handwritten digits from 0 to 9, and the goal is to predict the depicted number given in the image:



This dataset is quite a famous one; creating a model to recognize the digits often serves as a *Hello World* for neural networks and deep learning.

This chapter starts with a simple network with only one inner layer. Since all the images are `28 * 28` pixels, the input layer should have `28 * 28` neurons (the pictures are grayscale, so there is only one channel). To be able to input the pictures into the network, we first need to *unwrap* it into a one-dimensional array:



As we already know, with ND4J, this is very easy to do; we just invoke `reshape(1, -1)`. However, we do not need to do it; DeepLearning4J will handle it automatically and reshape the input for us.

Next, we create an inner layer, and we can start with 1,000 neurons there. Since there are 10 digits, the number of neurons in the output layer should be equal to 10.

Now, let's implement this network in DeepLearning4J. Since MNIST is a very popular dataset, the library already provides a convenient loader for it, so all we need to do is use the following code:

```
int batchSize = 128;
int seed = 1;
DataSetIterator mnistTrain = new MnistDataSetIterator(batchSize, true,
seed);
DataSetIterator mnistTest = new MnistDataSetIterator(batchSize, false,
seed);
```

For the training part, there are 50,000 labeled examples, and there are 10,000 testing examples. To iterate over them, we use DeepLearning4j's abstraction-- `DataSetIterator`. What it does here is takes the entire dataset, shuffle it, and then chunks it into batches of 128 pictures.

The reason we prepare batches is that neural networks are typically trained with **Stochastic Gradient Descent** (**SGD**), and the training happens in batches; we take a batch, train a model on it, update the weights, and then take the next batch. Taking one batch and training a model on it is called an **iteration**, and iterating over all available training batches is called an **epoch**.

After getting the data, we can specify the training configuration of our network:

```
NeuralNetConfiguration.Builder config = new
NeuralNetConfiguration.Builder();
config.seed(seed);
config.optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT);
config.learningRate(0.005);
config.regularization(true).l2(0.0001);
```

In this code, we say that we want to use SGD for training with a learning rate of `0.005` and L2 regularization of `0.0001`. SGD is a reasonable default and you should stick to it.

The learning rate is the most important training configuration parameter. If we set it too high, then the training procedure will diverge, and if it is too small--it will take a lot of time before converging. For selecting the optimal learning rate, we typically run the training procedure for values such as 0.1, 0.01, 0.001, ..., 0.000001 and see when the neural network stops diverging.

Another thing we used here was L2 regularization. L1 and L2 regularization work in exactly the same way as in linear models such as logistic regression--they help to avoid overfitting by making the weights smaller, and L1 ensures the sparsity of the solution.

However, there are regularization strategies specific to neural networks--dropout and dropconnect, which *mute* a random part of the net at each training iteration. We can specify them for the entire network in the configuration:

```
config.dropOut(0.1);
```

But the preferable way is to specify them per layer--we will see later how to do it.

Once we are done with the training configuration, we can continue with specifying the architecture of the net, that is, things such as its layers and the number of neurons in each.

For that we get an object of the `ListBuilder` class:

```
ListBuilder architecture = config.list();
```

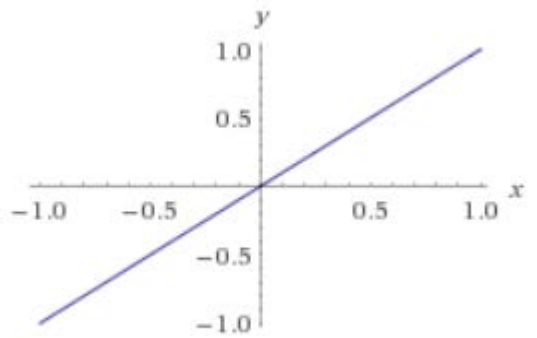
Now, let's add the first layer:

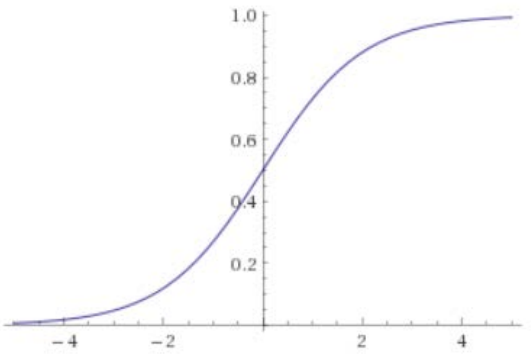
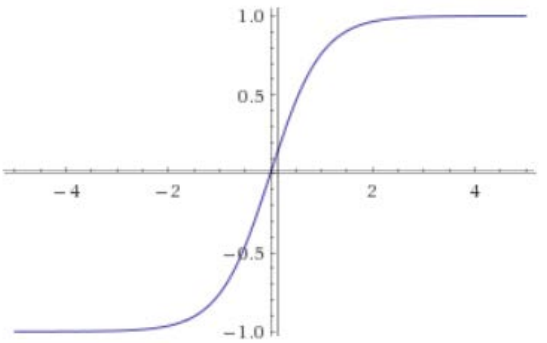
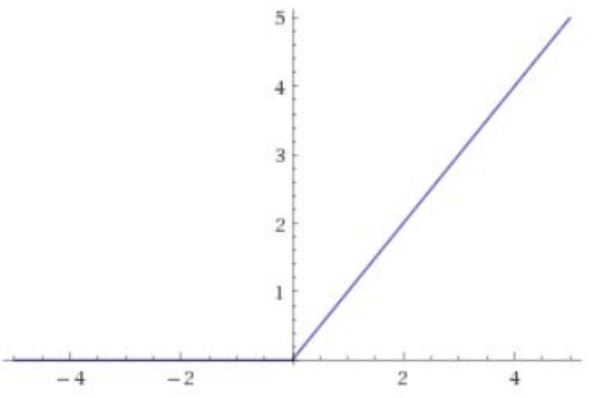
```
DenseLayer.Builder innerLayer = new DenseLayer.Builder();
innerLayer.nIn(28 * 28);
innerLayer.nOut(1000);
innerLayer.activation("tanh");
innerLayer.weightInit(WeightInit.UNIFORM);
architecture.layer(0, innerLayer.build());
```

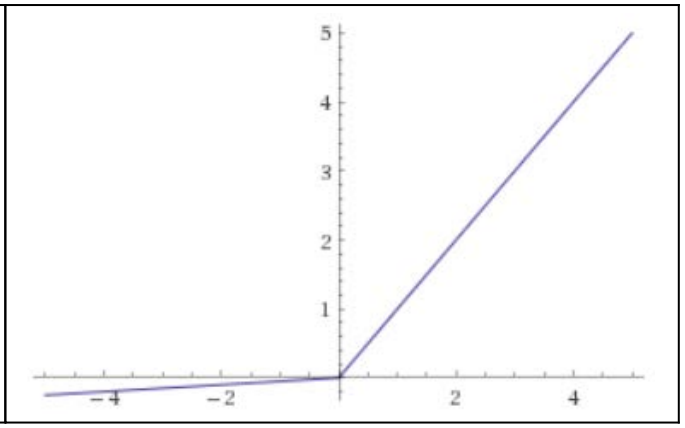
As we previously discussed, the number of neurons in the input layer should be equal to the size of the image, which is 28 times 28. Since the inner layer has 1,000 neurons, the output of this layer is 1,000.

Additionally, we specify here the activation function and the weight initialization strategy.

The activation function is the nonlinear transformation, which is applied to each neuron's output. There can be several activation functions:

Activation	**Plot**
linear: no activation	

sigmoid: `[0, 1]` range	
tanh: `[-1, 1]` range	
ReLU: `[0, infinity]` range	

Leaky ReLU: `[-infinity, infinity]`	

For this example, we used `tanh`, which was the default option for shallow networks in the pre-deep-learning era. However, for deep networks, ReLU activations should typically be preferred because they solve the vanishing gradient problem.



Vanishing gradient is a problem that occurs during the training of neural networks. For training, we calculate the gradient--the direction which we need to follow, and update the weights based on that. This problem occurs when we use `sigmoid` or `tanh` activations in deep networks--the first layers (processed last during optimization) have a very small gradient and do not get updated at all.

However, ReLU also sometimes has a problem, called **dying ReLU**, which can be solved using other activation functions such as LeakyReLU.



If the input to the ReLU function is negative, then the output is exactly zero, which means that in many cases the neuron is not activated. What is more, while training the derivative, in this case, is is zero, so following the gradient may never update the weights. This is known as the **dying ReLU** problem, many neurons never get activated and die. This problem can be solved using the LeakyReLU activation, instead of always outputting zero for negative values, it outputs something very small, so the gradient can still be calculated.

Another thing we specified here is the weight initialization. Typically, before we train a network, we need to initialize the parameters, and some initializations work better than others, but, as usual, this is case-specific and often we need to try several methods before settling on a particular one.

Weight initialization method	**Comment**
`WeightInit.ZERO`	Here, all the weights are set to zero. This is not recommended.
`WeightInit.UNIFORM`	Here, weights are set to uniform values in `[-a, a]` range, where `a` depends on the number neurons.
`WeightInit.XAVIER`	This is the Gaussian distribution with variance, which depends on the number of neurons. If in doubt, use this initialization.
`WeightInit.RELU`	This is the Gaussian distribution with higher variance than in `XAVIER`. It helps with the Dying ReLU problem.
`WeightInit.DISTRIBUTION`	This lets you specify any distribution from which the weights will be sampled. In this case, the distribution is set this way: `layer.setDist(new NormalDistribution(0, 0.01));`.
others	There are other weight initialization strategies, see the JavaDocs of the `WeightInit` class.

The `UNIFORM` and `XAVIER` methods are usually good starting points; try them first and see if they produce good results. If not, then try to experiment and choose some other methods.



If you experience the dying ReLU problem, then it is best to use the `WeightInit.RELU` initialization method. Otherwise, use `WeightInit.XAVIER`.

Next, we specify the output layer:

```
architecture.layer(1, outputLayer.build());
```

For the output layer, we need to specify the `loss` function--the function we want to optimize with the network during training. There are multiple options, but the most common ones are as follows:

- `LossFunction.NEGATIVELOGLIKELIHOOD`, which is `LogLoss`. Use this for classification.
- `LossFunction.MSE`, which is Mean Squared Error. Use it for regression.

You might have noticed that here we used a different activation function--softmax, and we have not covered this activation previously. This is a generalization of the `sigmoid` function to multiple classes. If we have a binary classification problem, and we want to predict only one value, the probability of belonging to the positive class, then we use a `sigmoid`. But if our problem is multiclass, or we output two values for the binary classification problem, then we need to use softmax. If we solve the regression problem, then we use the linear activation function.

Output activation	**When to use**
`sigmoid`	Binary classification
`softmax`	Multiclass classification
`linear`	Regression

Now, when we have established the architecture, we can build the network from it:

```
MultiLayerNetwork nn = new MultiLayerNetwork(architecture.build());
nn.init();
```

It is often useful to monitor the training progress and see the scores as the model train, and for that we can use `ScoreIterationListener`--it subscribes to the model, and after each iteration it outputs the new training score:

```
nn.setListeners(new ScoreIterationListener(1));
```

Now we are ready to train the network:

```
int numEpochs = 10;
for (int i = 0; i < numEpochs; i++) {
    nn.fit(mnistTrain);
}
```

Here, we train the network for 10 epochs, that is, we iterate over the entire training dataset 10 times, and if you remember, each epoch consists of a number of 128-sized batches.

Once the training is done, we can evaluate the performance of the model on the test. For this purpose, we create a special object of type `Evaluation`, and then we iterate over the batches of the test set, and apply the model to each batch. Every time we do this, we update the `Evaluation` object, which keeps track of the overall performance.

Once the training is done, we can evaluate the performance of the model. For this we create a special object of type `Evaluation`, and then iterate over the validation dataset and apply the model to each batch. The results are recorded by the `Evaluation` class, and in the end we can see the result:

```
while (mnistTest.hasNext()) {
    DataSet next = mnistTest.next();
    INDArray output = nn.output(next.getFeatures());
    eval.eval(next.getLabels(), output);
}

System.out.println(eval.stats());
```

If we run it for 10 epochs, it will produce this:

```
Accuracy:        0.9
Precision:       0.8989
Recall:          0.8985
F1 Score:        0.8987
```

So the performance is not very impressive, and to improve it, we can modify the architecture, for example, by adding another inner layer:

```
DenseLayer.Builder innerLayer1 = new DenseLayer.Builder();
innerLayer1.nIn(numrow * numcol);
innerLayer1.nOut(1000);
innerLayer1.activation("tanh");
innerLayer1.dropOut(0.5);
innerLayer1.weightInit(WeightInit.UNIFORM);
architecture.layer(0, innerLayer1.build());

DenseLayer.Builder innerLayer2 = new DenseLayer.Builder();
innerLayer2.nIn(1000);
innerLayer2.nOut(2000);
innerLayer2.activation("tanh");
innerLayer2.dropOut(0.5);
innerLayer2.weightInit(WeightInit.UNIFORM);
architecture.layer(1, innerLayer2.build());

LossFunction loss = LossFunction.NEGATIVELOGLIKELIHOOD;
OutputLayer.Builder outputLayer = new OutputLayer.Builder(loss);
outputLayer.nIn(2000);
```

```
outputLayer.nOut(10);
outputLayer.activation("softmax");
outputLayer.weightInit(WeightInit.UNIFORM);
architecture.layer(2, outputLayer.build());
```

As you can see, here we added an extra layer with `2000` neurons between the first layer and the output layer. We also added dropout to each layer for regularization purposes.

With this setup, we can achieve slightly better accuracy:

```
Accuracy:        0.9124
Precision:       0.9116
Recall:          0.9112
F1 Score:        0.9114
```

Of course, the improvement is only marginal, and the network is far from being well-tuned. To improve it, we can use the ReLU activation, Nesterov's updater with momentum around 0.9, and XAVIER's weight initialization. This should give an accuracy higher than 95%. In fact, you can find a very well-tuned network in the examples from the official DeepLearning4j repository; look for the class named `MLPMnistSingleLayerExample.java`.

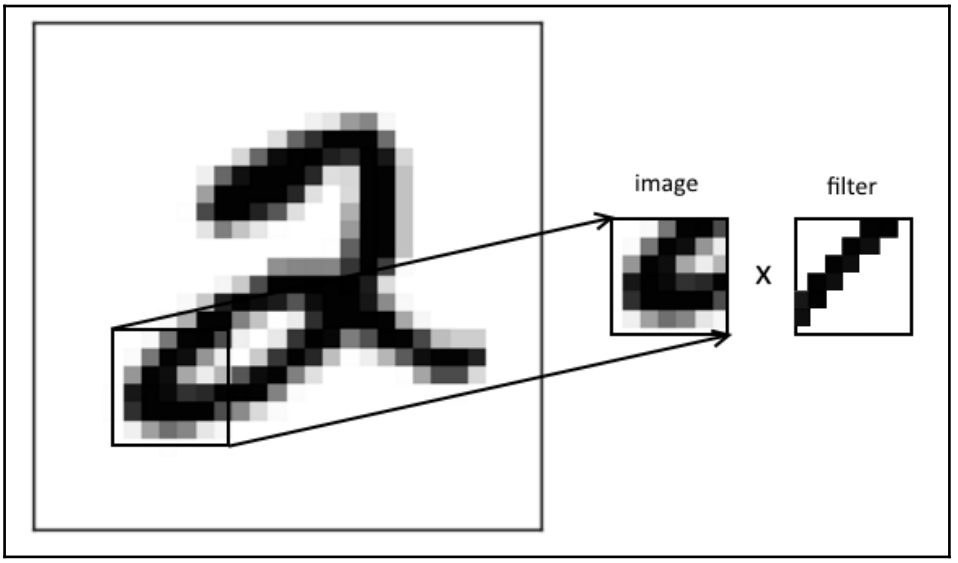
In our example, we used classical neural network; they are rather shallow (that is, they do not have many layers), and all the layers are fully connected. While for small-scale problems this might be good enough, typically it is better to use Convolutional Neural Networks for image recognition tasks, these networks take into account the image structure and can achieve better performance.

Convolutional Neural Networks

As we have already mentioned several times, neural networks can do the feature engineering part themselves, and this is especially useful for images. Now we will finally see this in action. For that we will use Convolutional Neural Networks, they are a special kind of neural networks that uses special convolutional layers. They are very well suited for image processing.

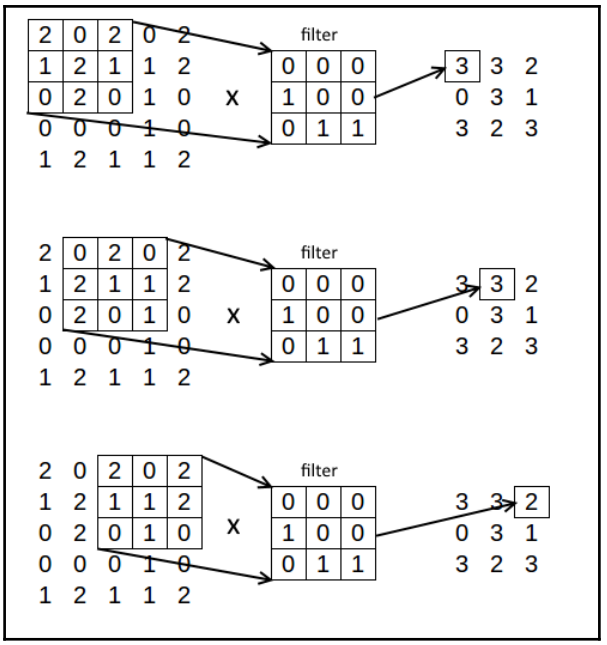
In the usual neural networks, the layers are fully connected, meaning that each neuron of a layer is connected to all the neurons from the previous layer. For `28 x 28` images such as digits from MNIST, this is not a big deal, but it starts to be a problem for larger images. Imagine that we need to process images of size `300 x 300`; in this case, the input layer will have 90,000 neurons. Then, if the next layer also has 90,000 neurons, then there will be `90000 x 90000` connections between these two layers, which is clearly a lot.

In images, however, only a small area of each pixel is important. So the preceding problem can be solved by considering only a small neighborhood for each pixel, and this is exactly what convolutional layers do; inside, they keep a set of *filters* of some small size. Then, we slide a window over the image and calculate the similarity of the content within the window to each of the filters:

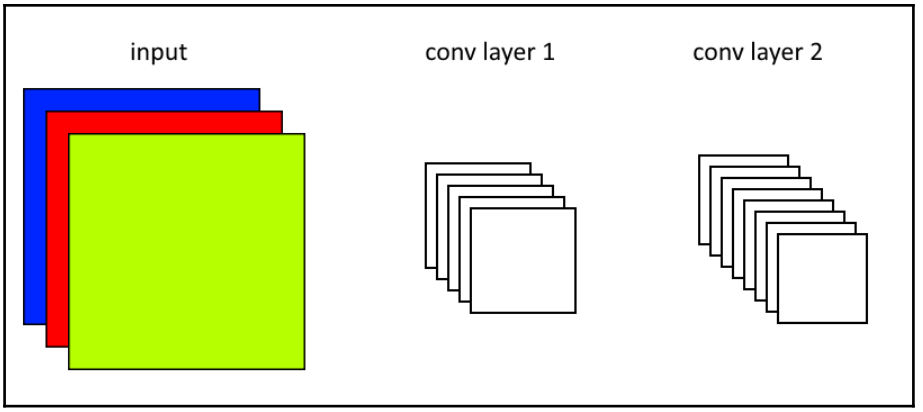


The filters are the neurons in these convolutional layers, and they are learned during training phase, in a way similar to the usual fully-connected case.

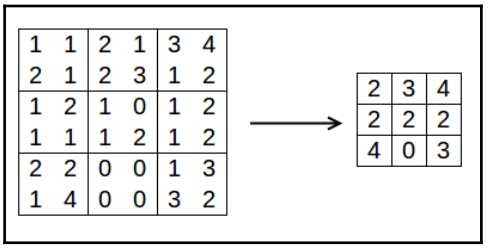
When we slide a window over an image, we calculate the similarity of the content to the filter, which is the dot product between them. For each window, we write the results to the output. We say the filter is activated when the area under consideration is similar to the filter. Clearly, if it is similar, the dot product will tend to produce higher values.



Since images usually have multiple channels, we actually deal with volumes (or 3D tensors) of dimensionality **number of channels** times **height** times **width**. When an image goes through a convolutional layer, each filter is applied in turn, and as the output, we have the volume of dimensionality **number of filters** times **height** times **width**. When we stack such layers on top of each other, we get a sequence of volumes:

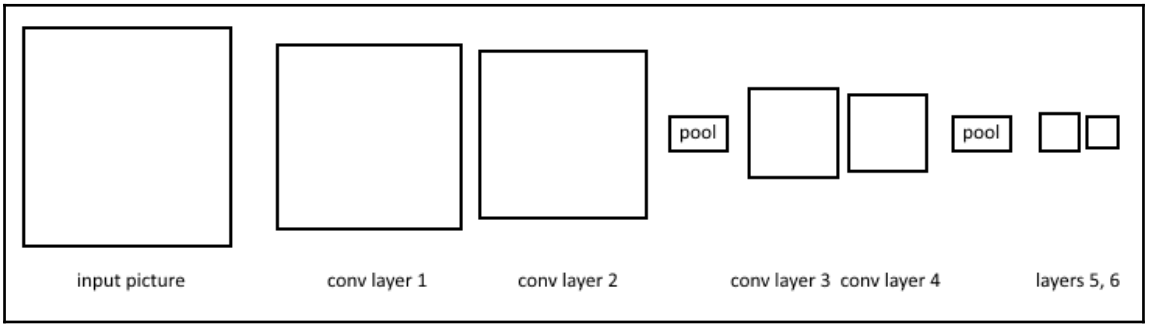


Apart from convolutional layers, another layer type is important for convolutional networks--the downsampling layer, or the pooling layer. The purpose of this layer is to reduce the dimensionality of the input, and usually each side is reduced by the factor of 2, so in total, the dimensionality is reduced 4 times. Usually, we use max pooling, which keeps the maximal value when downsampling:



The reason we do this is to decrease the number of parameters our network has, it makes the training a lot faster.

When such a layer gets a volume, it only changes the height and the width but does not change the number of filters. Typically, we put the pooling layers after convolutional layers, and often organize the architecture such that two convolutional layers are followed by one pooling layer:



Then, at some point, after we have added enough convolutional layers, we switch to fully-connected layers, the same type of layer we have in usual networks. And then, in the end, we have the output layer, as we did previously.

Let's continue with the MNIST example, but this time let's train a Convolutional Neural Network to recognize digits. For this task, there is a famous architecture called LeNet (created by researcher Yann LeCun), so let's implement it. We will base our example on the official DeepLearning4j example available in their repository.

The architecture is as follows:

- `5 x 5` convolutional layer with 20 filters
- Max pooling
- `5 x 5` convolutional layer with 50 filters
- Max pooling
- Fully connected layers with 500 neurons
- Output layer with softmax

So there are six layers in this network.

Like we did previously, first, we specify the training configuration of the network:

```
NeuralNetConfiguration.Builder config = new
NeuralNetConfiguration.Builder();
config.seed(seed);
config.regularization(true).l2(0.0005);
config.learningRate(0.01);
config.weightInit(WeightInit.XAVIER);
config.optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT);
config.updater(Updater.NESTEROVS).momentum(0.9);
```

There is almost nothing new here, except for `Updater`;we use Nesterov's update with momentum set to `0.9`. The purpose of this is faster convergence.

Now we can create the architecture:

```
ListBuilder architect = config.list();
```

First, the convolutional layer:

```
ConvolutionLayer cnn1 = new ConvolutionLayer.Builder(5, 5)
        .name("cnn1")
        .nIn(nChannels)
        .stride(1, 1)
        .nOut(20)
        .activation("identity")
        .build();
architect.layer(0, cnn1);
```

Here, in the Builder's constructor, we specify the dimensionality of the filter, which is `5 x 5`. Then the `nIn` parameter is set to the number of channels in the input pictures, which is 1 for MNIST, they are all gray scale images. The `nOut` parameter specifies the number of filters this layer has. The stride parameter specifies the step at which we slide the window over the image, and typically it is set to `1`. Finally, the layer does not use any activation.

The next layer in the architecture is the pooling layer:

```
SubsamplingLayer pool1 = new SubsamplingLayer.Builder(PoolingType.MAX)
        .name("pool1")
        .kernelSize(2, 2)
        .stride(2, 2)
        .build();
architect.layer(1, pool1);
```

When we create this layer, we first specify the way we want to downsample, and we use `MAX` since we are interested in max pooling. There are other options such as `AVG` average, and `SUM`, but they are not used very frequently in practice.

This layer has two parameters--the `kernelSize` parameter, which is the size of the window we slide over the picture, and the stride parameter, which is the step we take when sliding the window. Typically, these values are set to `2`.

Then we add the next convolutional layer and a pooling layer for it:

```
ConvolutionLayer cnn2 = new ConvolutionLayer.Builder(5, 5)
        .name("cnn2")
        .stride(1, 1)
        .nOut(50)
        .activation("identity")
        .build();
architect.layer(2, cnn2);
SubsamplingLayer pool2 = new SubsamplingLayer.Builder(PoolingType.MAX)
        .name("pool2")
        .kernelSize(2, 2)
        .stride(2, 2)
        .build();
architect.layer(3, pool2);
```

Finally, we create the fully-connected layer and the output layer:

```
DenseLayer dense1 = new DenseLayer.Builder()
        .name("dense1")
        .activation("relu")
        .nOut(500)
        .build();
architect.layer(4, dense1);
```

```
OutputLayer output = new
OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
        .name("output")
        .nOut(outputNum)
        .activation("softmax")
        .build();
architect.layer(5, output);
```

For the last two layers, there is nothing new for us, we do not use any new parameters.

Finally, before training, we need to tell the optimizer that the input is a picture, and this is done by specifying the input type:

```
architect.setInputType(InputType.convolutionalFlat(height, width,
nChannels));
```

With this, we are ready to begin the training:

```
for (int i = 0; i < nEpochs; i++) {
    model.fit(mnistTrain);
    Evaluation eval = new Evaluation(outputNum);

    while (mnistTest.hasNext()) {
        DataSet ds = mnistTest.next();
        INDArray out = model.output(ds.getFeatureMatrix(), false);
        eval.eval(ds.getLabels(), out);
    }

    System.out.println(eval.stats());
    mnistTest.reset();
}
```

For this architecture, the accuracy the network can achieve after one epoch is 97%, which is significantly better than our previous attempts. But after training it for 10 epochs, the accuracy is 99%.

Deep learning for cats versus dogs

While MNIST is a very good dataset for educational purpose, it is quite small. Let's take a look at a different image recognition problem: given a picture, we want to predict if there is a cat on the image or a dog.

For this, we will use the dataset with dogs and cats pictures from a competition run on kaggle, and the dataset can be downloaded from `https://www.kaggle.com/c/dogs-vs-cats`.

Let's start by first reading the data.

Reading the data

For the dogs versus cats competition, there are two datasets; training, with 25,000 images of dogs and cats, 50% each, and testing. For the purposes of this chapter, we only need to download the training dataset. Once you have downloaded it, unpack it somewhere.

The filenames look like the following:

`dog.9993.jpg`
`dog.9994.jpg`
`dog.9995.jpg`
	`cat.10000.jpg`
	`cat.10001.jpg`
	`cat.10002.jpg`

The label (`dog` or `cat`) is encoded into the filename.

As you know, the first thing we always do is to split the data into training and validation sets. Since all we have here is a collection of files, we just get all the filenames and then split them into two parts--training and validation.

For that we can use this simple script:

```
File trainDir = new File(root,  "train");
double valFrac = 0.2;
long seed = 1;

Iterator<File> files = FileUtils.iterateFiles(trainDir, new String[] {
```

```
"jpg" }, false);
List<File> all = Lists.newArrayList(files);
Random random = new Random(seed);
Collections.shuffle(all, random);

int trainSize = (int) (all.size() * (1 - valFrac));
List<File> train = all.subList(0, trainSize);
copyTo(train, new File(root, "train_cv"));

List<File> val = all.subList(trainSize, all.size());
copyTo(val, new File(root, "val_cv"));
```

In the code, we use the `FileUtils.iterateFiles` method from Apache Commons IO to iterate over all `.jpg` files in the training directory. Then we put all these files into a list, shuffle it, and split them into 80% and 20% parts.

The `copyTo` method just copies the files into the specified directory:

```
private static void copyTo(List<File> pics, File dir) {
    for (File pic : pics) {
        FileUtils.copyFileToDirectory(pic, dir);
    }
}
```

Here, the `FileUtils.copyFileToDirectory` method also comes from Apache Commons IO.

There are a number of things we need to do to use the data for training a network. They are as follows:

- Getting the paths to each picture
- Getting the label (`dog` or `cat` from the filename)
- Resizing the input so every picture has the same size
- Applying some normalization to the image
- Creating `DataSetIterator` from it

Getting the paths to each picture is easy and we already know how to do it, we can use the same method from Commons IO as we did previously. But now we need to get `URI` for each file as DeepLearning4j dataset iterators expect the file's `URI`, and not the file itself. For that we create a helper method:

```
private static List<URI> readImages(File dir) {
    Iterator<File> files = FileUtils.iterateFiles(dir,
                                  new String[] { "jpg" }, false);
    List<URI> all = new ArrayList<>();
```

```
        while (files.hasNext()) {
            File next = files.next();
            all.add(next.toURI());
        }

        return all;
    }
```

Getting the class name (`dog` or `cat`) from the filename is done by implementing the `PathLabelGenerator` interface:

```
    private static class FileNamePartLabelGenerator implements
    PathLabelGenerator {

        @Override
        public Writable getLabelForPath(String path) {
            File file = new File(path);
            String name = file.getName();
            String[] split = name.split(Pattern.quote("."));
            return new Text(split[0]);
        }

        @Override
        public Writable getLabelForPath(URI uri) {
            return getLabelForPath(new File(uri).toString());
        }
    }
```

Inside we just split the filename by `.` and then take the first element of the result.

Finally, we create a method, which takes in a list of `URI` and creates a `DataSetIterator`:

```
    private static DataSetIterator datasetIterator(List<URI> uris)
                        throws IOException {
        CollectionInputSplit train = new CollectionInputSplit(uris);
        PathLabelGenerator labelMaker = new FileNamePartLabelGenerator();

        ImageRecordReader trainRecordReader = new ImageRecordReader(HEIGHT,
    WIDTH, CHANNELS, labelMaker);
        trainRecordReader.initialize(train);

        return new RecordReaderDataSetIterator(trainRecordReader, BATCH_SIZE,
    1, NUM_CLASSES);
    }
```

This method uses some constants, which we initialize with the following values:

```
HEIGHT = 128;
WIDTH = 128;
CHANNELS = 3;
BATCH_SIZE = 30;
NUM_CLASSES = 2;
```

The `ImageRecordReader` will use the `HEIGHT` and `WIDTH` parameters to resize the images into the specified form, and if it is grayscale, it will artificially create RGB channels for it. The `BATCH_SIZE` specifies how many images we will consider at once during training.

In linear models, normalization plays an important role and helps the model converge faster. For neural networks as well, this is the case, so we need to normalize the image. For this, we can use a special built-in class `ImagePreProcessingScaler`. `DataSetIterator` can have a preprocessor, so we put this scaler there:

```
DataSetIterator dataSet = datasetIterator(valUris);
ImagePreProcessingScaler preprocessor = new ImagePreProcessingScaler(0, 1);
dataSet.setPreProcessor(preprocessor);
```

With this, the data preparation is done and we can proceed to create the model.

Creating the model

For the model's architecture, we will use a variation of the VGG network. This architecture is taken from a publicly available script from the forums (`https://www.kaggle.com/jeffd23/dogs-vs-cats-redux-kernels-edition/catdognet-keras-convnet-starter`) and here we will adapt this example to DeepLearning4j.



VGG is a model that took 2nd place in the image net 2014 challenge, and it uses only 3 x 3 and 2 x 2 convolutional filters.



It is always a good idea to use the existing architectures, as it solves a lot of time--coming up with a good architecture on your own is a challenging task.

The architecture we will use is as follows:

- Two layers of `3 x 3` convolution with 32 filters
- Max pooling
- Two layers of `3 x 3` convolution with 64 filters
- Max pooling
- Two layers of `3 x 3` convolution with 128 filters
- Max pooling
- A fully connected layer with 512 neurons
- A fully connected layer with 256 neurons
- The output layer with softmax activation

For our example, we will use ReLU activation for all the convolution and fully-connected layers. To avoid the dying ReLU problem, we will use the `WeightInit.RELU` weight initialization scheme. In our experiments, without using it, the network tends to produce the same result no matter what input it receives.

So, first we start with the configuration:

```
NeuralNetConfiguration.Builder config = new
NeuralNetConfiguration.Builder();
config.seed(SEED);
config.weightInit(WeightInit.RELU);
config.optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT);
config.learningRate(0.001);
config.updater(Updater.RMSPROP);
config.rmsDecay(0.99);
```

Some of the parameters should already be quite familiar by now, but there are two new things here--the `RMSPROP` updater and the `rmsDecay` parameter. Using them allows us to adaptively change the learning rate as we train. At the beginning, the learning rate is larger and we take bigger steps toward the minimum, but as we train and approach the minimum, it decreases the learning rate and we take smaller steps.

The learning rate was selected by trying different values such as 0.1, 0.001 and 0.0001 and watching when the network stops diverging. This is easy to spot because when diverging the training error varies a lot and then starts outputting Infinity or `NaN`.

Now we specify the architecture.

First, we create the convolutional layers and the pooling layers:

```
int l = 0;
ListBuilder network = config.list();

ConvolutionLayer cnn1 = new ConvolutionLayer.Builder(3, 3)
        .name("cnn1")
        .stride(1, 1)
        .nIn(3).nOut(32)
        .activation("relu").build();
network.layer(l++, cnn1);

ConvolutionLayer cnn2 = new ConvolutionLayer.Builder(3, 3)
        .name("cnn2")
        .stride(1, 1)
        .nIn(32).nOut(32)
        .activation("relu").build();
network.layer(l++, cnn2);

SubsamplingLayer pool1 = new SubsamplingLayer.Builder(PoolingType.MAX)
        .kernelSize(2, 2)
        .stride(2, 2)
        .name("pool1").build();
network.layer(l++, pool1);

ConvolutionLayer cnn3 = new ConvolutionLayer.Builder(3, 3)
        .name("cnn3")
        .stride(1, 1)
        .nIn(32).nOut(64)
        .activation("relu").build();
network.layer(l++, cnn3);

ConvolutionLayer cnn4 = new ConvolutionLayer.Builder(3, 3)
        .name("cnn4")
        .stride(1, 1)
        .nIn(64).nOut(64)
        .activation("relu").build();
network.layer(l++, cnn4);

SubsamplingLayer pool2 = new SubsamplingLayer.Builder(PoolingType.MAX)
        .kernelSize(2, 2)
        .stride(2, 2)
        .name("pool2").build();
network.layer(l++, pool2);

ConvolutionLayer cnn5 = new ConvolutionLayer.Builder(3, 3)
        .name("cnn5")
        .stride(1, 1)
```

```
        .nIn(64).nOut(128)
        .activation("relu").build();
network.layer(l++, cnn5);

ConvolutionLayer cnn6 = new ConvolutionLayer.Builder(3, 3)
        .name("cnn6")
        .stride(1, 1)
        .nIn(128).nOut(128)
        .activation("relu").build();
network.layer(l++, cnn6);

SubsamplingLayer pool3 = new SubsamplingLayer.Builder(PoolingType.MAX)
        .kernelSize(2, 2)
        .stride(2, 2)
        .name("pool3").build();
network.layer(l++, pool3);
```

There should be nothing new for us here. And then we create the fully-connected layers and the output:

```
DenseLayer dense1 = new DenseLayer.Builder()
        .name("ffn1")
        .nOut(512).build();
network.layer(l++, dense1);

DenseLayer dense2 = new DenseLayer.Builder()
        .name("ffn2")
        .nOut(256).build();
network.layer(l++, dense2);

OutputLayer output = new
OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
        .name("output")
        .nOut(2)
        .activation("softmax").build();
network.layer(l++, output);
```

Finally, as previously, we specify the input size:

```
network.setInputType(InputType.convolutionalFlat(HEIGHT, WIDTH, CHANNELS));
```

Now, we create the model from this architecture and specify the score listener for train monitoring purposes:

```
MultiLayerNetwork model = new MultiLayerNetwork(network.build())
ScoreIterationListener scoreListener = new ScoreIterationListener(1);
model.setListeners(scoreListener);
```

As for training, it happens in exactly the same was as we did previously--we train the model for a few epochs:

```
List<URI> trainUris = readImages(new File(root, "train_cv"));
DataSetIterator trainSet = datasetIterator(trainUris);
trainSet.setPreProcessor(preprocessor);

for (int epoch = 0; epoch < 10; epoch++) {
    model.fit(trainSet);
}

ModelSerializer.writeModel(model, new File("model.zip"), true);
```

In the end, we also save the model into a ZIP archive with three files inside--the coefficients of the model, the configuration (both parameters and the architecture) in a `.json` file, and the configuration for the updater--in case we would like to continue training the model in the future (the last parameter `true` tells us to save it and with `false` we cannot continue training).

Here, however, the performance monitoring is pretty primitive, we only watch the training error and do not look at the validation error at all. Next, we will look at more options for performance monitoring.

Monitoring the performance

What we did for monitoring previously was adding the listener, which outputs the training score of the model after each iteration:

```
MultiLayerNetwork model = new MultiLayerNetwork(network.build())
ScoreIterationListener scoreListener = new ScoreIterationListener(1);
model.setListeners(scoreListener);
```

This will give you some idea of the performance of the model, but only on the training data, but we typically need more than that--at least the performance on the validation set would be useful to know to see if we start overfitting or not.

So, let's read the validation dataset:

```
DataSetIterator valSet = datasetIterator(valUris);
valSet.setPreProcessor(preprocessor);
```

For training, previously we just took the dataset iterator and passed it to the fit function. We can improve this process by taking all the training data, shuffling it before each epoch, and chunking it into parts, with each chunk being equal to 20 batches. After the training on each chunk is finished, we can iterate over the validation set and see the current validation performance of the model.

In code, it looks like this:

```
for (int epoch = 0; epoch < 20000; epoch++) {
    ArrayList<URI> uris = new ArrayList<>(trainUris);
    Collections.shuffle(uris);
    List<List<URI>> partitions = Lists.partition(uris, BATCH_SIZE * 20);

    for (List<URI> set : partitions) {
        DataSetIterator trainSet = datasetIterator(set);
        trainSet.setPreProcessor(preprocessor);
        model.fit(trainSet);
        showTrainPredictions(trainSet, model);
        showLogloss(model, valSet, epoch);
    }

    saveModel(model, epoch);
}
```

So here we take the `URI`, shuffle them, and partition them into lists of 20 batches. For partitioning, we use the `Lists.partition` method from Google Guava. From each such partition, we create a dataset iterator and use it for training the model, and then, after each chunk, we look at the validation score to make sure the network is not overfitting.

Also, it is helpful to see what the network predicts for the data it was just trained on, especially to check if the network is learning anything. We do this inside the `showTrainPredictions` method. If the predictions are different for different inputs, then it is a good sign. Also, you may want to see how close the predictions are to the actual labels.

Additionally, we save the model at the end of each epoch, in case something goes wrong, we can train the process. If you noticed, we set the number of epochs to a high number, so at some point we can just stop the training (for example, when we see from the logs that we start overfitting), and just take the last good model.

Let's see how these methods are implemented:

```
private static void showTrainPredictions(DataSetIterator trainSet,
            MultiLayerNetwork model) {
    trainSet.reset();
    DataSet ds = trainSet.next();
    INDArray pred = model.output(ds.getFeatureMatrix(), false);
    pred = pred.get(NDArrayIndex.all(), NDArrayIndex.point(0));
    System.out.println("train pred: " + pred);
}
```

The `showLogLoss` method is simple, but a bit verbose because of the iterators. It does the following:

- Go over all batches in the validation dataset
- Record the prediction and the true label for each batch
- Put all predictions together in a single double array and does the same with the actual labels
- Calculate the log loss using the code we wrote in `Chapter 4`, *Supervised Learning - Classification and Regression*.

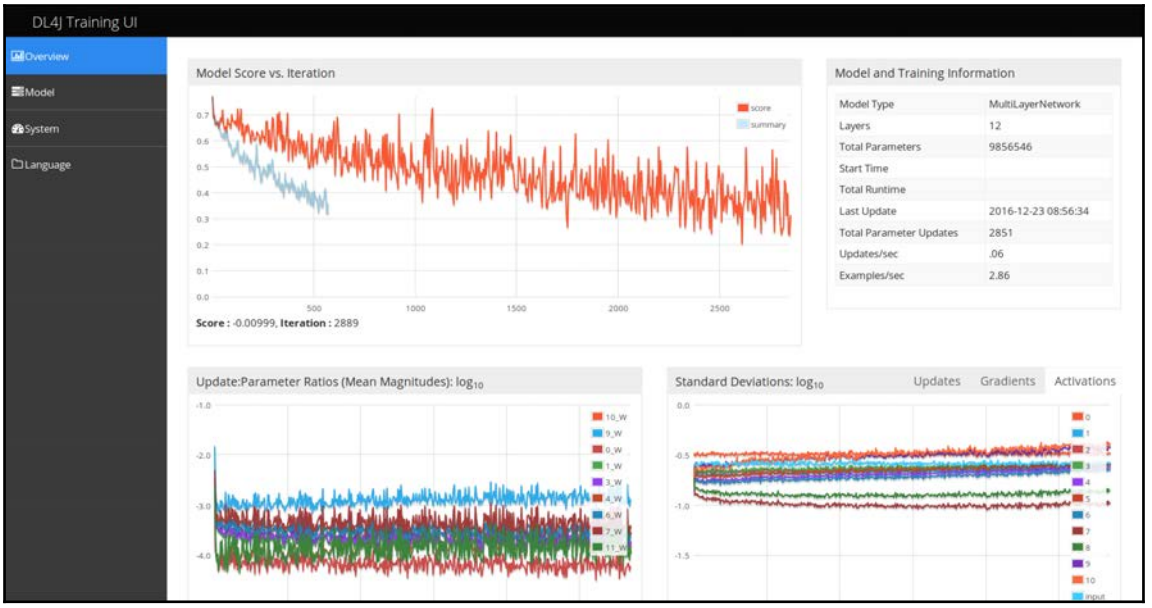
For brevity, we will omit the exact code here, but you are welcome to check the code bundle.

Saving the model is simple, and we already know how to do it. Here we just add some extra information to the filename about the epoch number:

```
private static void saveModel(MultiLayerNetwork model, int epoch) throws
IOException {
    File locationToSave = new File("models", "cats_dogs_" + epoch +
".zip");
    boolean saveUpdater = true;
    ModelSerializer.writeModel(model, locationToSave, saveUpdater);
}
```

Now, when we have a lot of information to monitor, it becomes quite difficult to comprehend everything from the logs. To make life easier for us, DeepLearning4j comes with a special graphical dashboard for monitoring.

This is how the dashboard looks:



Let's add this to our code. First, we need to add an extra dependency to our `pom`:

```
<dependency>
  <groupId>org.deeplearning4j</groupId>
  <artifactId>deeplearning4j-ui_2.10</artifactId>
  <version>0.7.1</version>
</dependency>
```

It is written in Scala, which is why there is the `_2.10` suffix at the end, it tells us that this version is written in Scala 2.10. Since we are using it from Java, it does not matter for us, so we can take any version we want.

Next, we can create the instance of the UI server, and create a special listener for the network, which will subscribe to the network's updates:

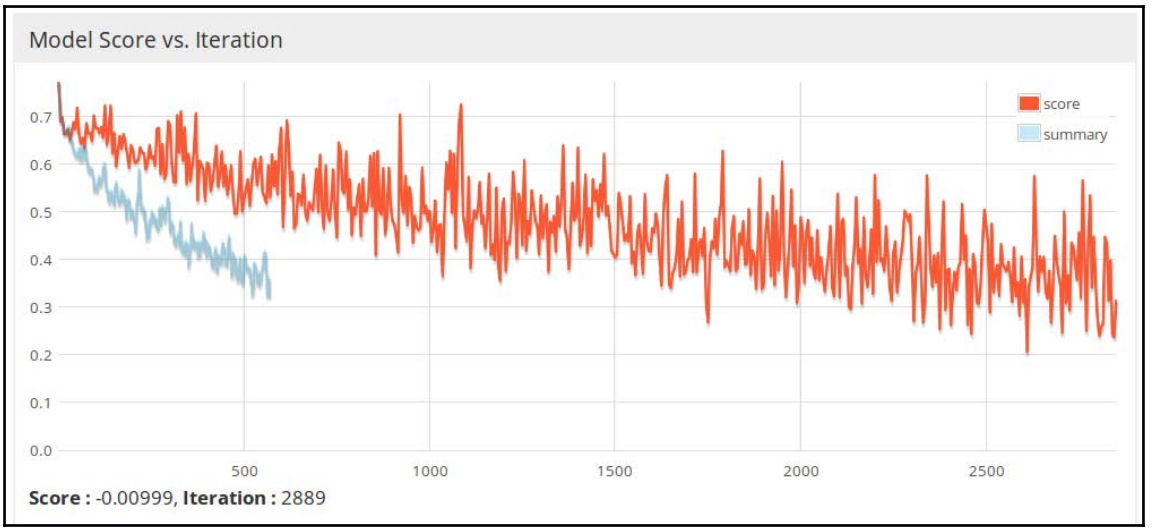
```
UIServer uiServer = UIServer.getInstance();
StatsStorage statsStorage = new InMemoryStatsStorage();
uiServer.attach(statsStorage);
StatsListener statsListener = new StatsListener(statsStorage);
```

And we use it in the same way we used `ScoreIterationListener`, we add it to the model via the `setListeners` method:

```
MultiLayerNetwork model = createNetwork();
ScoreIterationListener scoreListener = new ScoreIterationListener(1);
model.setListeners(scoreListener, statsListener);
```

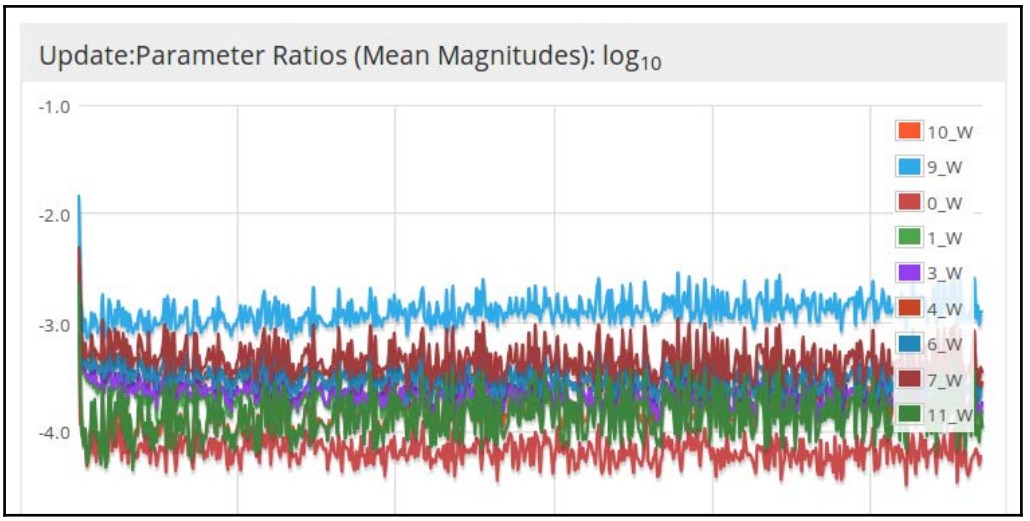
With these changes, when we run the code, it starts the UI server, which we can see if we open the browser and go to `http://localhost:9000`; this will show the dashboard from the preceding code.

These charts are quite useful. The most useful one is the chart showing the model score at each iteration. This is the training score, the same one we see in the logs from `ScoreIterationListener`, which looks like this:



Based on this chart, we can understand how the model behaves during the training--whether the training process is stable, or whether the model is learning anything at all. Ideally, we should see the downward trend as shown in the preceding screenshot. If there is no decrease in the score, then maybe there is something wrong with the network configuration, for example, the learning rate is too small, not good weight initialization or too much regularization. If there is an increase in the score, then the most likely problem is too large learning rate.

The other charts also allow to monitor the training process. The parameter ratios chart shows the changes in the parameters between each iterations, on the logarithmic scale (that is, -3.0 corresponds to 0.001 change between iterations). If you see that the change is too low, for example, below -6.0, then, probably, the network is not learning anything.



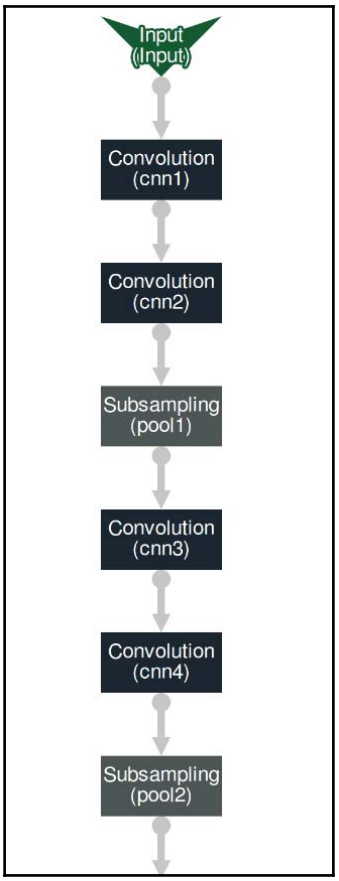
Finally, there is a chart that shows the standard deviations of all activations. The reason we may need this is to detect the so-called *vanishing* and *exploding* activations:





The **vanishing activation** problem is related to the **vanishing gradient** problem. For some activations, a change in input results is almost no change in the output, and the gradient is almost zero, so the neuron is not updated, so its activation *vanishes*. The exploding activation is the opposite, the activation score keeps growing until it reaches infinity.

In this interface, we can also see the full network on the **Models** tab. Here is a part of our model:



If we click on each individual layer, we can see some charts for this specific layer.

With these tools, we can closely monitor the performance of the model and adapt the training process and our parameters when we see that something unusual happens.

Data augmentation

For this problem, we have only 25, 000 training examples. For a deep learning model, this amount of data is usually not enough to capture all the details. No matter how sophisticated our network is and how much time we spent tuning it, at some point 25, 000 examples will not be enough to improve the performance further.

Often, getting more data is very expensive or not possible at all. But what we can do is generating more data from the data we already have, and this is called **data augmentation**. Usually, we generate new data by doing some of the following transformations:

- Rotating the image
- Flipping the image
- Randomly cropping the image
- Switching the color channels (for example, changing the red and blue channels)
- Changing color saturation, constrast, and brightness
- Adding noise

In this chapter, we will look at the first three transformations--rotation, flipping, and cropping. To do them, we will use `Scalr`--a library for image manipulation. Let's add it to the `pom` file:

```
<dependency>
  <groupId>org.imgscalr</groupId>
  <artifactId>imgscalr-lib</artifactId>
  <version>4.2</version>
</dependency>
```

It is very simple and only extends the standard Java API, in the same sense as Apache Commons Lang does--by providing useful utility methods around the standard functionality.

For rotation and flipping, we just use the `Scalr.rotate` method:

```
File image = new File("cat.10000.jpg");
BufferedImage src = ImageIO.read(image);
Rotation rotation = Rotation.CW_90;
BufferedImage rotated = Scalr.rotate(src, rotation);
File outputFile = new File("cat.10000_cw_90.jpg");
ImageIO.write(rotated, "jpg", outputFile);
```

As you see, this is pretty easy to use and quite intuitive. All we need to do is pass a `BufferedImage` and the desired `Rotation`. `Rotation` is an enum with the following values:

- `Rotation.CW_90`: This involves clockwise rotation by 90 degrees
- `Rotation.CW_180`: This involves clockwise rotation by 180 degrees
- `Rotation.CW_270`: This involves clockwise rotation by 270 degrees
- `Rotation.FLIP_HORZ`: This involves flipping the image horizontally
- `Rotation.FLIP_VERT`: This involves flipping the image vertically

Cropping is also not difficult, and it's done via the `Scalr.crop` method, which takes in four parameters--the position where the crop starts (`x` and `y` coordinates) and the size of the crop (height and width). For our problem what we can do is randomly select a coordinate in the top left corner of the image, and then randomly select the height and the width of the crop. We can do it this way:

```
int width = src.getWidth();
int x = rnd.nextInt(width / 2);
int w = (int) ((0.7 + rnd.nextDouble() / 2) * width / 2);

int height = src.getHeight();
int y = rnd.nextInt(height / 2);
int h = (int) ((0.7 + rnd.nextDouble() / 2) * height / 2);

if (x + w > width) {
    w = width - x;
}

if (y + h > height) {
    h = height - y;
}

BufferedImage crop = Scalr.crop(src, x, y, w, h);
```

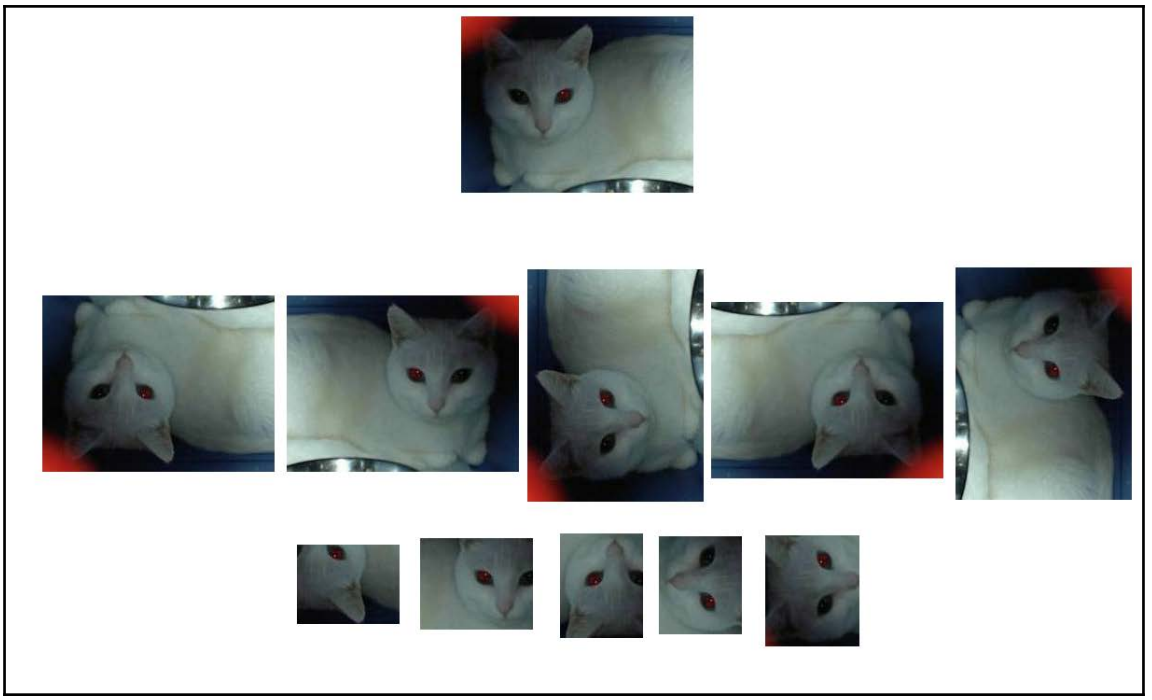
Here, we first randomly select the `x` and `y` coordinates and then select the width and the height. In the code, we select the weight and the height such that they are at least 35% of the image--but can go up to 60% of the image. Of course, you are free to play with these parameters and change them to whatever makes more sense.

Then we also check if we do not overcome the image boundaries, that is, the crop always stays within the image; and finally we call the `crop` method. Optionally, we can also rotate or flip the cropped image at the end.

So, for all the files, it may look like this:

```
for (File f : all) {
    BufferedImage src = ImageIO.read(f);
    for (Rotation rotation : Rotation.values()) {
        BufferedImage rotated = Scalr.rotate(src, rotation);
        String rotatedFile = f.getName() + "_" + rotation.name() + ".jpg";
        File outputFile = new File(outputDir, rotatedFile);
        ImageIO.write(rotated, "jpg", outputFile);

        int width = src.getWidth();
        int x = rnd.nextInt(width / 2);
        int w = (int) ((0.7 + rnd.nextDouble() / 2) * width / 2);

        int height = src.getHeight();
        int y = rnd.nextInt(height / 2);
        int h = (int) ((0.7 + rnd.nextDouble() / 2) * height / 2);

        if (x + w > width) {
            w = width - x;
        }

        if (y + h > height) {
            h = height - y;
        }

        BufferedImage crop = Scalr.crop(src, x, y, w, h);
        rotated = Scalr.rotate(crop, rotation);

        String cropppedFile = f.getName() + "_" + x + "_" + w + "_" +
                    y + "_" + h + "_" + rotation.name() + ".jpg";

        outputFile = new File(outputDir, cropppedFile);
        ImageIO.write(rotated, "jpg", outputFile);
    }
}
```

In this code, we iterate over all training files, and then we apply all the rotations to the image itself and to a random crop from this image. This code should generate 10 new images from each source image. For example, for the following image of a cat, there will be 10 images generated as follows:



We only briefly listed the possible augmentations, and if you remember, the last one was *adding random noise*. This is typically easy to implement, so here are some ideas as to what you can do:

- Replace some pixel values with 0 or with some random value
- Add or subtract the same small number from all the values
- Generate some Gaussian noise with a small variance and add it to all the channels
- Add the noise only to a part of the image
- Invert a part of the image
- Add a filled square of some random color to the image; the color could have the alpha channel (that is, it could somewhat transparent) or not
- Apply strong JPG encoding to the image

With this, you can virtually generate an infinite number of data samples for training. Of course, you probably will not need so many samples, but by using these techniques, you can augment any image dataset and considerably improve the performance of the model trained on this data.

Running DeepLearning4J on GPU

As we mentioned previously, DeepLearning4j relies on ND4J for numerical calculations. ND4J is an interface, and there are multiple possible implementations. So far, we have used the one based on OpenBLAS, but there are other ones. We also mentioned that ND4J can utilize a **Graphics Processing Unit** (**GPU**), which is a lot faster than CPUs for typical Linear Algebra operations used in neural networks such as matrix multiplication. To use it, we need to get the CUDA ND4J backend.



CUDA is an interface used for executing the computations on NVidia's GPUs, and it supports a wide range of graphical cards. Internally, ND4J uses CUDA to run numerical computations on GPUs.

If you have previously executed all the code on the CPU via BLAS, you must have noticed how slow it is. Switching the ND4J backend to CUDA should considerably improve the performance by several orders of magnitude.

This is done by including the following dependency to the `pom` file:

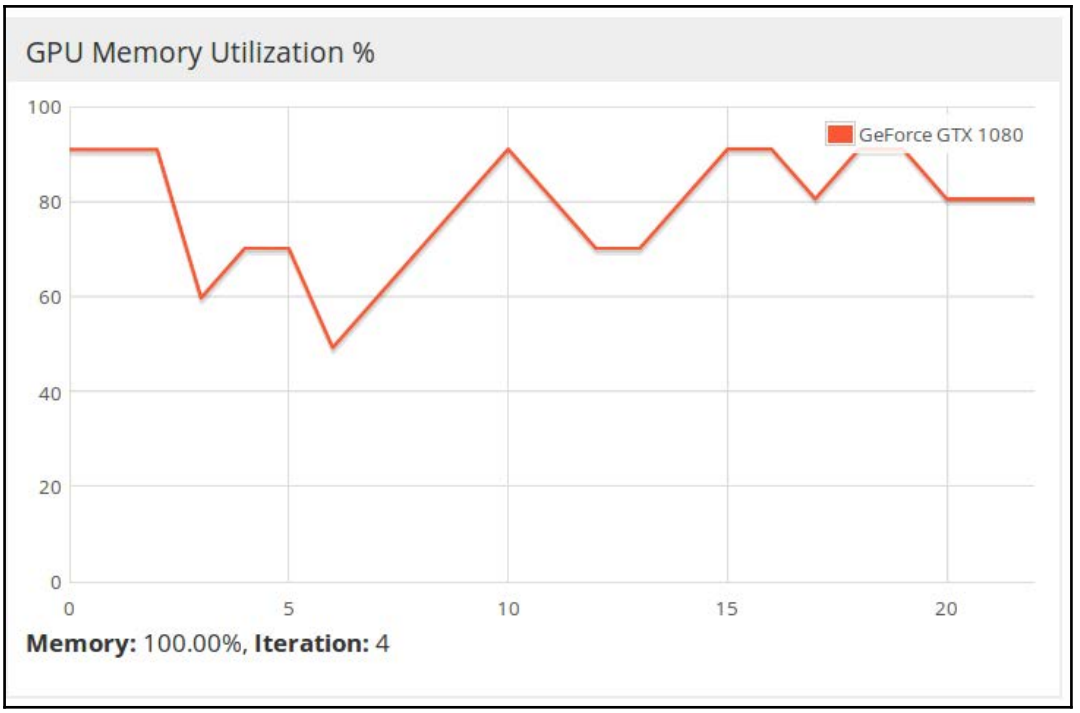
```
<dependency>
  <groupId>org.nd4j</groupId>
  <artifactId>nd4j-cuda-7.5</artifactId>
  <version>0.7.1</version>
</dependency>
```

This dependency assumes that you have CUDA 7.5 installed.

For CUDA 8.0, you should replace 7.5 with 8.0: both CUDA 7.5 and CUDA 8.0 are supported by ND4J.

If you already have a GPU with all the drivers installed, just adding this dependency is enough to use the GPU for training the networks, and when you do this, you will see a great performance boost.

What is more, you can use the UI dashboard to monitor the GPU memory usage, and if you see that it is low, you can try to utilize it better, for example, by increasing the batch size. You can find this chart on the **System** tab:



If you do not have a GPU, but do not want to wait while your CPU is crunching numbers, you can easily rent a GPU computer. There are cloud providers such as Amazon AWS who allow you to instantly get a server with a GPU even just for a few hours.

If you have never rented a server on Amazon AWS, we have prepared simple instructions on how to get started with training there.

Before renting a server, let's first prepare everything that we need; the code and the data.

For data, we just take all the files (including the augmented ones) and put them into a single archive file:

```
zip -r all-data.zip  train_cv/  val_cv/
```

Then, we need to build the code such that we also have all the `.jar` files with dependencies laying nearby. This is done with `maven-dependency-plugin`, a plugin for Maven. We have used this plugin previously in `Chapter 3`, *Exploratory Data Analysis*, so we will omit the XML configuration that we need to add to our `pom.xml` file.

Now we use Maven to compile our code and put this into a `.jar` file:

```
mvn package
```

In our case, the project is called `chapter-08-dl4j`, so executing the package goal with Maven creates a `chapter-08-dl4j-0.0.1-SNAPSHOT.jar` file in the `target` folder. But since we also use the dependency plugin, it creates a `libs` folder, where you can find all the dependencies. Let's put everything into a single `.zip` file:

```
zip -r code.zip chapter-08-dl4j-0.0.1-SNAPSHOT.jar libs/
```

After performing the preparation steps, we will have two ZIP files, `all-data.zip` and `code.zip`.

Now, when we have prepared the program and the data, we can go to `aws.amazon.com` and sign into the console or create an account if you don't have one yet. When you are in, select **EC2**, which will bring you to the EC2 dashboard. Next, you can select the region you are interested in. You can either select something geographically close or the cheapest one. Usually, N. Virginia and US West Oregon are quite cheap compared to others.

Then, find the **Launch Instance** button and click on it.



If you need a GPU computer only for a few hours, you can choose to create a spot instance--they are cheaper than the usual instances, but their price is dynamic, and at some point, such an instance can die if somebody else is willing to pay more for the instance you are using. When starting it, you can set a price threshold, and if you choose something like $1 there, the instance should last for a long time.

When creating an instance, it is possible to use an existing AMI, an image of a system with some software preinstalled. The best option here is to look for CUDA, which will give you the official NVidia CUDA 7.5 image, but you are free to choose any other image you want.



Note that some of the AMIs are not free, be careful when choosing. Also, choose the AMI provider you can trust, as sometimes there could be malicious images, which will use the computational resources for something else than your task. If in doubt, use the official NVidia image, or create an image yourself from scratch.

Once you select the image, you can choose the instance type. For our purposes, the `g2.2.xlarge` instance is enough, but there are larger and more powerful ones if you wish.

Next, you need to select the storage type; we don't need anything and can skip this step. But the next is important, here we set up the security rules. Since the UI dashboard runs on port 9,000, we need to open it, so it is accessible from the outside world. We can then add a custom TCP rule and write `9000` there.

After this step, we are done and can proceed to launching the instance before reviewing the details.

Next, it will ask you to specify the key pair (`.pem`) for ssh to the instance, and you can create and download a new one if you don't have any. Let's create a key pair named `dl4j` and save it to the home folder.

Now, the instance is launched and ready to use. To access it, go to the dashboard and find the public DNS of the instance, this is the name you can use for accessing the server from your machine. Let's put this into an environment variable:

```
EC2_HOST=ec2-54-205-18-41.compute-1.amazonaws.com
```

From now on, we will assume you are using a bash shell on Linux, but it should work well in MacOS or Windows with cygwin or MinGW.

Now, we can upload the `.jar` file we previously built along with the data. For that, we will use `sftp`. Connecting the `sftp` client using the `pem` file is done this way:

```
sftp -o IdentityFile=~/dl4j.pem ec2-user@$EC2_HOST
```

Note that you should be in the folder with the data and the program archive. Then you can upload them by executing the following commands:

```
put code.zip
put all-data.zip
```

The data is uploaded, so now we can apply `ssh` to the instance to run the program:

```
ssh -i "~/dl4j.pem" ec2-user@$EC2_HOST
```

The first thing we do is to unpack the archives:

```
unzip code.zip
unzip all-data.zip
```



If for some reason you do not have any free space left in the home folder, run the `df -h` command to see if there are any places with free space. There must be other disks with available space, where you can store the data.

By now we have unpacked everything and are ready to execute the code. But if you use the CUDA 7.5 AMI from NVidia, it only has Java 7 support. Since we used Java 8 for writing the code, we need to install Java 8:

```
sudo yum install java-1.8.0-openjdk.x86_64
```

We do not want the execution to stop when we leave the ssh session, so it is best to create `screen` there (or you can use `tmux` if you prefer):

```
screen -R dl4j
```

Now we run the code there:

```
java8 -cp chapter-08-dl4j.jar:libs/* chapter08.catsdogs.VggCatDog ~/data
```

Once you see that the model started training, you can detach the screen by pressing *Ctrl + A* followed by D. Now you can close the terminal and use the UI to watch the training process. To do it, just put `EC2_HOST:9000` to your browser, where `EC2_HOST` is the public DNS of the instance.

This is it, now you just need to wait for some time until your model converges.

There could be some problem along the way.

If it says that it cannot find `openblas` binaries, then you have several options. You can either remove the `dl4j-native` JARS from the `libs` folder, or you can install openblas. The first option might be preferrable because we don't need to use the CPU anyways.

Another issue you can potentially run into is a missing NVCC executable, which is needed for dj4j's CUDA 7.5 library. Solving it is easy, you just need to add the path to CUDA's binaries to the PATH variable:

```
PATH=/usr/local/cuda-7.5/bin:$PATH
```

Summary

In this chapter, we looked at how we can use deep learning in Java applications, learned the basics of the DeepLearning4j library, and then tried to apply it to an image recognition problem where we wanted to classify images to dogs and cats.

In the next chapter, we will cover Apache Spark--a library for distributing data science algorithms across a cluster of machines.

9
Scaling Data Science

So far we have covered a lot of material about data science, we learned how to do both supervised and unsupervised learning in Java, how to perform text mining, use XGBoost and train Deep Neural Networks. However, most of the methods and techniques we used so far were designed to run on a single machine with the assumption that all the data will fit into memory. As you should already know, this is often the case: there are very large datasets that are not possible to process with traditional techniques on a typical hardware.

In this chapter, we will see how to process such datasets--we will look at the tools that allow processing the data across several machines. We will cover two use cases: one is large scale HTML processing from Common Crawl - the copy of the Web, and another is Link Prediction for a social network.

We will cover the following topics:

- Apache Hadoop MapReduce
- Common Crawl processing
- Apache Spark
- Link prediction
- Spark GraphFrame and MLlib libraries
- XGBoost on Apache Spark

By the end of this chapter, you will learnt how to use Hadoop to extracting data from Common Crawl, how to use Apache Spark for link prediction, and how to use XGBoost in Spark.

Apache Hadoop

Apache Hadoop is a set of tools that allows you to scale your data processing pipelines to thousands of machines. It includes:

- **Hadoop MapReduce**: This is a data processing framework
- **HDFS:** This is a distributed filesystem, which allows us to store data on multiple machines
- **YARN:** This is the executor of MapReduce and other jobs

We will only cover MapReduce, as it is the core of Hadoop and it is related to data processing. We will not cover the rest, and we will also not talk about setting up or configuring a Hadoop Cluster as this is slightly beyond scope for this book. If you are interested in knowing more about it, *Hadoop: The Definitive Guide* by *Tom White* is an excellent book for learning this subject in depth.

In our experiments, we will use the local mode, that is, we will emulate the cluster, but still run the code on a local machine. This is very useful for testing, and once we are sure that it works correctly, it can be deployed to a cluster with no changes.

Hadoop MapReduce

As we already said, Hadoop MapReduce is a library, that allows you to process data in a scalable way.

There are two main abstractions in the MapReduce framework: Map and Reduce. This idea originally comes from the functional programming paradigm, where `map` and `reduce` are high-level functions:

- `map`: This takes in a function and a sequence of elements and applies the function to each of the elements in turn. The result is a new sequence.
- `reduce`: This also takes in a function and a sequence, and it uses this function to process the sequence and ultimately return a single element in the end.

In this book, we have already used the map function from the Java Stream API quite extensively, starting with `Chapter 2`, *Data Processing Toolbox*, so you must be quite familiar with it by now.

In Hadoop MapReduce, the `map` and `reduce` functions are a bit different from their predecessors:

- Map takes in an element and returns a number of key-value pairs. It can return nothing, one, or several such pairs, so it is more `flatMap` than `map`
- Then the output is grouped by key via sorting
- Finally, `reduce` takes in a group, and for each group outputs a number of key-value pairs

Typically, MapReduce is illustrated with the word count example: given a text, we want to count how many times each word appeared in the text. The solution is as follows:

- The `map` takes in text, then tokenizes it, and for each token outputs a pair `(token, 1)`, where `token` is the key, and `1` is the associated value.
- The `reducer` sums over all ones and this is the final count.

We will implement something similar: instead of just counting words, we will create TF-IDF vectors for each of the tokens in the corpus. But first, we need to get a large amount of text data from somewhere. We will use the Common Crawl dataset, which contains a copy of the Web.

Common Crawl

Common Crawl (`http://commoncrawl.org/`) is a repository of data crawled from the Internet over the last seven years. It is extremely large and, what is more is, it is available for everyone to download and analyze.

Of course, we will not be able to use all of it: even a small fraction is so large that it requires a big and powerful cluster for processing it. In this chapter, will take a few archives from the end of 2016, and extract the text ting TF-IDF.

Downloading the data is not complex and you can find the instructions at `http://commoncrawl.org/the-data/get-started/`. The data is already available in the S3 storage, so AWS users can access it easily. In this chapter, however, we will download a part of Common Crawl via HTTP without using AWS.

At the time of writing, the most recent data is from December 2016, which is located at `s3://commoncrawl/crawl-data/CC-MAIN-2016-50`. As per the instruction, we first need to get all the paths to individual archive files for this month, and they are stored in a `warc.paths.gz` file. So, in our case, we are interested in `s3://commoncrawl/crawl-data/CC-MAIN-2016-50/warc.paths.gz`.

Since we do not plan to use AWS, we need to convert it to a path downloadable via HTTP. For that, we replace `s3://commoncrawl/` with `https://commoncrawl.s3.amazonaws.com`:

```
wget
https://commoncrawl.s3.amazonaws.com/crawl-data/CC-MAIN-2016-50/warc.paths.
gz
```

Let's look at the file:

```
zcat warc.paths.gz | head -n 3
```

You will see a lot of lines like this (the suffixes are omitted for brevity):

```
.../CC-MAIN-20161202170900-00000-ip-10-31-129-80.ec2.internal.warc.gz
.../CC-MAIN-20161202170900-00001-ip-10-31-129-80.ec2.internal.warc.gz
.../CC-MAIN-20161202170900-00002-ip-10-31-129-80.ec2.internal.warc.gz
```

To download it via HTTP, we again need to append `https://commoncrawl.s3.amazonaws.com/` to every line of this file. This is easily achieved with awk:

```
zcat warc.paths.gz
  | head
  | awk '{ print "https://commoncrawl.s3.amazonaws.com/" $0}'
  > files.txt
```

Now we have the first 10 URLs from this file, so we can download them:

```
for url in $(cat files.txt); do
  wget $url;
done
```

To speed things up, we can download the files in parallel with gnu-parallel:

```
cat files.txt | parallel --gnu "wget {}"
```

Now we have downloaded somewhat biggish data: about 10 files of 1GB each. Note that there are about 50,000 lines in the path file, there are approximately 50,000 GBs of data just for December. This is a lot of data, and everybody can use it at any time!

We won't use all of it and will only concentrate on the 10 files we have already downloaded. Let's process them with Hadoop.

The first step is normal: we need to specify the dependencies to Hadoop in the `.pom` file:

```
<dependency>
  <groupId>org.apache.hadoop</groupId>
  <artifactId>hadoop-client</artifactId>
  <version>2.7.3</version>
</dependency>
<dependency>
  <groupId>org.apache.hadoop</groupId>
  <artifactId>hadoop-common</artifactId>
  <version>2.7.3</version>
</dependency>
```

Common Crawl uses WARC for storing the HTML data: this is a special format for storing the crawled data. To be able to process it, we need to add a special library for reading it:

```
<dependency>
  <groupId>org.netpreserve.commons</groupId>
  <artifactId>webarchive-commons</artifactId>
  <version>1.1.2</version>
</dependency>
```

Next, we need to tell Hadoop how to use such files. For this purpose, programmers typically need to provide implementations of `FileRecordReader` and `FileImportFormat` classes. Luckily, there are open source implementations, which we can just copy and paste to our projects. One of them is available at `https://github.com/Smerity/cc-warc-examples`in the `org.commoncrawl.warc` package. So we just copy `WARCFileInputFormat` and `WARCFileRecordReader` from there to our project. This code is also included in the code bundle of this book, in case the repository is removed.

With this, we are ready to start coding. First, we need to create a `Job` class: it specifies which mapper and reducer classes will be used to run the job and allows us to configure how this job will be executed. So, let's create an `WarcPreparationJob` class, which extends the `Configured` class and implements the `Tool` interface:

```
public class WarcPreparationJob extends Configured implements Tool {
    public static void main(String[] args) throws Exception {
        int res = ToolRunner.run(new Configuration(),
              new WarcPreparationJob(), args);
        System.exit(res);
    }

    public int run(String[] args) throws Exception {
```

```
            // implementation goes here
        }
    }
```

The Java doc for the `Tool` interface is quite informative and describes in detail how such a `Job` class should be implemented: it has overriden the `run` method, where it should specify the input and the output paths as well as the mapper and reducer classes.

We will adapt this code slightly: first, we will have a map-only Job, so we do not need a reducer. Also, since we are working with texts, it is useful to compress the output. So, let's create the `run` method with the following code. First, we create a `Job` class:

```
Job job = Job.getInstance(getConf());
```

Now we will look at the input and its format (WARC in our case):

```
Path inputPath = new Path(args[0]);
FileInputFormat.addInputPath(job, inputPath);
job.setInputFormatClass(WARCFileInputFormat.class);
```

Next, we specify the output, which is gzipped text:

```
Path outputPath = new Path(args[1];
TextOutputFormat.setOutputPath(job, outputPath);
TextOutputFormat.setCompressOutput(job, true);
TextOutputFormat.setOutputCompressorClass(job, GzipCodec.class);
job.setOutputFormatClass(TextOutputFormat.class);
```

Usually, the output is key-value pairs, but since we just want to process WARC and extract the text from there, we only output a key, and no value:

```
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(NullWritable.class);
```

Finally, we specify the mapper class and say that there will be no reducers:

```
job.setMapperClass(WarcPreparationMapper.class);
job.setNumReduceTasks(0);
```

Now, when we have specified the job, we can implement the mapper class--`WarcPreparationMapper` in our case. This class should extend the `Mapper` class. All mappers should implement the `map` method, so our mapper should have the following outline:

```
public class WarcPreparationMapper extends
        Mapper<Text, ArchiveReader, Text, NullWritable> {
```

```
    @Override
    protected void map(Text input, ArchiveReader archive, Context context)
              throws IOException, InterruptedException {
        // implementation goes here
    }

}
```

The `map` method takes in a WARC `archive` with a collection of records, so we want to process all of them. Thus, we put the following to the `map` method:

```
for (ArchiveRecord record : archive) {
    process(record, context);
}
```

And the process method does the following: it extracts the HTML from the record, then extracts the text from HTML, tokenizes it, and then, finally, writes the results to the output. In code it looks like this:

```
String url = record.getHeader().getUrl();
String html = TextUtils.extractHtml(record);
String text = TextUtils.extractText(html);
List<String> tokens = TextUtils.tokenize(text);
String result = url + "t" + String.join(" ", tokens);
context.write(new Text(result), NullWritable.get());
```

Inside we use three helper functions: `extractHtml`, `extractText`, and `tokenize`. We have already used the last two (`exctractHtml` and `tokenize`) a few times, so we will omit their implementation; refer to `Chapter 6`, *Working with Text - Natural Language Processing and Information Retrieval*.

And the first one, `extractHtml`, contains the following code:

```
byte[] rawData = IOUtils.toByteArray(r, r.available());
String rawContent = new String(rawData, "UTF-8");
String[] split = rawContent.split("(r?n){2}", 2);
String html = split[1].trim();
```

It converts the data from the archive to `String` with UTF-8 encoding (which sometimes might not be ideal because not all pages on the Internet use UTF-8), and then removes the response header and keeps only the remaining HTML.

Finally, to run these classes, we can use the following code:

```
String[] args = { /data/cc_warc", "/data/cc_warc_processed" };
ToolRunner.run(new Configuration(), new WarcPreparationJob(), args);
```

Here we manually specify the "command-line" parameters (the ones that you get in the main method) and pass them to the `ToolRunner` class, which can run Hadoop Jobs in the local model.



The results may have pornographic content. Since Common Crawl is a copy of the Web, and there are a lot of pornographic websites on the Internet, it is very likely that you will see some pornographic text in the processed results. It is quite easy to filter it out by keeping a special a list of pornographic keywords and discarding all the documents that contain any of these words.

After running this job, you will see that there are plenty of different languages in the results. If we are interested in a specific language, then we can automatically detect the language of a document, and keep only those documents that are in the language we are interested in.

Several Java libraries which can do language detection. One of them is language-detector, which can be included in our project with the following dependency snippet:

```
<dependency>
  <groupId>com.optimaize.languagedetector</groupId>
  <artifactId>language-detector</artifactId>
  <version>0.5</version>
</dependency>
```

Not surprisingly, this library uses machine learning for detecting the language. So, the first thing we need to do to use it is to load the model:

```
List<LanguageProfile> languageProfiles =
        new LanguageProfileReader().readAllBuiltIn();
LanguageDetector detector =
LanguageDetectorBuilder.create(NgramExtractors.standard())
        .withProfiles(languageProfiles)
        .build();
```

And we can use it this way:

```
Optional<LdLocale> result = detector.detect(text);
String language = "unk";
if (result.isPresent()) {
    language = result.get().getLanguage();
}
```

With this, we can just keep the articles in English (or any other language) and discard the rest. So, let's extract the text from the files we downloaded:

```
String lang = detectLanguage(text.get());
if (lang.equals("en")) {
    // process the data
}
```

Here, `detectLanguage` is a method, that contains the code for detecting what is the language of the text: we wrote this code earlier.

Once we have processed the WARC files and extracted the text from them, we can calculate IDF for every token in our corpus. For that, we need to first calculate DF - Document Frequency. This is very similar to the Word Count example:

- First, we need a `mapper` that outputs `1` for each distinct word in the document
- Then the `reducer` sums up all the ones to come up with the final count

This job will process the documents we just parsed from Common Crawl.

Let's create the mapper. It will have the following code in the `map` method:

```
String doc = value.toString();
String[] split = doc.split("t");
String joinedTokens = split[1];
Set<String> tokens = Sets.newHashSet(joinedTokens.split(" "));
LongWritable one = new LongWritable(1);

for (String token : tokens) {
    context.write(new Text(token), one);
}
```

The input to the mapper is a `Text` object (named `value`), which contains the URL and the tokens. We split the tokens and keep only distinct ones using a `HashSet`. Finally, for each distinct token, we write `1`.

For calculating IDF, we typically need to know *N*: the number of documents in our corpus. There are two ways of getting it. First, we can use the counters: create a counter and increment it for each successfully processed document. It is quite easy to do.

The first step is to create a special `enum` with the possible counters we want to use in our application. Since we need to have only one type of counter, we create an `enum` with just one element:

```
public static enum Counter {
    DOCUMENTS;
}
```

The second step is to use the `context.getCounter()` method and increment the counter:

```
context.getCounter(Counter.DOCUMENTS).increment(1);
```

Once the job is over, we can get the value of the counter with this code:

```
Counters counters = job.getCounters();
long count = counters.findCounter(Counter.DOCUMENTS).getValue();
```

But there is another option: we can just pick a large number and use it as the number of documents. It typically does not need to be exact since IDFs of all tokens share the same *N*.

Now, let's continue with the `reducer`. Since the mapper outputs `Text` and a long (via `LongWritable`), the reducer gets in a `Text` and an iterable over `LongWritable` classes--this is the token and a bunch of ones. What we can do is just sum over all of them:

```
long sum = 0;
for (LongWritable cnt : values) {
    sum = sum + cnt.get();
}
```

To keep only frequent tokens, we can add a filter, to discard all infrequent words and make the results significantly smaller:

```
if (sum > 100) {
    context.write(key, new LongWritable(sum));
}
```

Then, the code in our `job` class for running it will look like this:

```
job.setInputFormatClass(TextInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(LongWritable.class);

job.setMapperClass(DocumentFrequencyMapper.class);
job.setCombinerClass(DocumentFrequencyReducer.class);
job.setReducerClass(DocumentFrequencyReducer.class);
```

Note that we not only set mapper and reducer, but also specify a combiner: this allows us to pre-aggregate some `1`'s that we output in the mapper and spend less time sorting the data and sending the results around the network.

Finally, to convert a document to TF-IDF, we can create a third job, reduce-less again, which will read the results of the first job (where we processed WARC files) and apply the IDF weighting from the second job.

We expect that the output of the second job should be quite small to fit into memory, so what we can do is send the file to all mappers, read it during the initialization, and then go over the lines of processed WARC and previously.

The main parts of the `job` class are the same: we input and the output is `Text`--the output is compressed and the number of reducer tasks is `0`.

Now we need to send the results of the `df` job to all the mappers. This is done via the cache files:

```
Path dfInputPath = new Path(args[3]);
job.addCacheFile(new URI(dfInputPath.toUri() + "#df"));
```

So here we specify the path to the `df` job results and then put it to the cache file. Note `#df` at the end: this is the alias we will use for accessing the file later.

Inside the mapper, we can read all the results into a map (in the setup method):

```
dfs = new HashMap<>();
File dir = new File("./df");
for (File file : dir.listFiles()) {
    try (FileInputStream is = FileUtils.openInputStream(file)) {
        LineIterator lines = IOUtils.lineIterator(is,
StandardCharsets.UTF_8);
        while (lines.hasNext()) {
            String line = lines.next();
            String[] split = line.split("t");
            dfs.put(split[0], Integer.parseInt(split[1]));
        }
    }
}
```

Here, `df` is the alias we assigned to the results file, and it is actually a folder, not a file. So to get the result, we need to go over each file in the folder, read them line by line, and put the results into a map. Then we can use this dictionary with counts in the map method for applying the IDF weight:

```
String doc = value.toString();
```

```
String[] split = doc.split("t");
String url = split[0];
List<String> tokens = Arrays.asList(split[1].split(" "));
Multiset<String> counts = HashMultiset.create(tokens);
String tfIdfTokens = counts.entrySet().stream()
        .map(e -> toTfIdf(dfs, e))
        .collect(Collectors.joining(" "));
Text output = new Text(url + "t" + tfIdfTokens);
context.write(output, NullWritable.get());
```

Here we take the tokens and use `Multiset` for calculating the Term Frequency. Next, we multiply the TF by IDF inside the `toTfIdf` function:

```
String token = e.getElement();
int tf = e.getCount();
int df = dfs.getOrDefault(token, 100);
double idf = LOG_N - Math.log(df);
String result = String.format("%s:%.5f", token, tf * idf);
```

Here, we get the DF (Document Frequency) for each input entry of `Multiset`, and if the token is not in our dictionary, we assume that it is quite rare, so we assign it the default DF of 100. Next, we calculate IDF and finally `tf*idf`. For calculating IDF, we use `LOG_N`, which is a constant we set to `Math.log(1_000_000)`.

For this example, 1 million was chosen as the number of documents. Even though the real number of documents is smaller (around 5k), it is the same for all the tokens. What is more, if we decide to add more documents to our index, we can still use the same *N* and not worry about re-calculating everything.

This produces the output that looks like this:

```
http://url.com/        flavors:9.21034 gluten:9.21034 specialty:14.28197
salad:18.36156 ...
```

As you must have noticed, the output of each job is saved to the disk. If we have multiple jobs, like we did, we need to read the data, process it, and save it back. I/O is quite costly, so some of these steps are intermediate and we do not need to save the results. In Hadoop there is no way to avoid this, which is why it is pretty slow sometimes and gives a lot of I/O overhead.

Luckily, there is another library that solves this problem: Apache Spark.

Apache Spark

Apache Spark is a framework for scalable data processing. It was designed to be better than Hadoop: it tries to process data in memory and not to save intermediate results on disk. Additionally, it has more operations, not just map and reduce, and thus richer APIs.

The main unit of abstraction in Apache Spark is **Resilient Distributed Dataset** (**RDD**), which is a distributed collection of elements. The key difference from usual collections or streams is that RDDs can be processed in parallel across multiple machines, in the same way, Hadoop jobs are processed.

There are two types of operations we can apply to RDDs: transformations and actions.

- **Transformations**: As the name suggests, it only changes data from one form to another. As input, they receive an RDD, and they also output an RDD. Operations such as map, flatMap, or filter are examples of transformation operations.
- **Actions**: These take in an RDD and produce something else, for example, a value, a list, or a map, or save the results. Examples of actions are count and reduce.

Like in the Java Steam API, transformations are lazy: they are not performed right away, but instead they are chained together and computed in one go, without the need to save the intermediate results to disk. In the Steam API, the chain is triggered by collecting the stream, and it is the same in Spark: when we perform an action, all transformation that is needed for this particular action are executed. If on the other hand some transformations are not required, then they will never be executed, which is why they are called *lazy*.

So let's start with Spark by first including its dependency to the `.pom` file:

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-core_2.11</artifactId>
  <version>2.1.0</version>
</dependency>
```

In this section, we will use it for computing TF-IDF: so we will try to reproduce the algorithm we just wrote for Hadoop.

The first step is to create the configuration and the context:

```
SparkConf conf = new SparkConf().setAppName("tfidf").setMaster("local[*]");
JavaSparkContext sc = new JavaSparkContext(conf);
```

Here, we specify the name of the Spark application and also the server URL to which it will connect. Since we are running Spark in a local mode, we put `local[*]`. This means that we set up a local server and create as many local workers as possible.

Spark relies on some Hadoop utilities, which makes it harder to run it on Windows. If you are running under Windows, you may have a problem with not being able to locate the `winutils.exe` file, which is needed by Spark and Hadoop. To solve this, do the following:

- Create a folder for Hadoop files, for example, `c:/tmp/hadoop` (make sure there are no spaces in the path)
- Download `winutils.exe` from `http://public-repo-1.hortonworks.com/hdp-win-alpha/winutils.exe`
- Put the file to the folder `c:/tmp/hadoop/bin`--note the `bin` subdirectory.
- Set the `HADOOP_HOME` environment variable to `c:/tmp/hadoop` and, alternatively, use the following code:

```
System.setProperty("hadoop.home.dir", "c:/tmp/hadoop");
```

This should solve the problem.

The next step is to read the text file, which we created with Hadoop after processing the Common Crawl files. Let's read it:

```
JavaRDD<String> textFile = sc.textFile("C:/tmp/warc");
```

To peek into the files, we can use the take function, which returns the top 10 elements from the RDD, and then we print each line to stdout:

```
textFile.take(10).forEach(System.out::println);
```

Now we can read the file line-by-line, and, for each document, output all the distinct tokens it has:

```
JavaPairRDD<String, Integer> dfRdd = textFile
    .flatMap(line -> distinctTokens(line))
   .mapToPair(t -> new Tuple2<>(t, 1))
   .reduceByKey((a, b) -> a + b)
   .filter(t -> t._2 >= 100);
```

Here, `distinctToken` is a function that splits the line and puts all the tokens into a set to keep only the distinct ones. Here is how it is implemented:

```
private static Iterator<String> distinctTokens(String line) {
    String[] split = line.split("t");
   Set<String> tokens = Sets.newHashSet(split[1].split(" "));
```

```
        return tokens.iterator();
    }
```

The `flatMap` function needs to return an iterator, so we invoke the iterator method on the set at the end.

Next, we convert each line to a tuple: this step is needed to tell Spark that we have key-value pairs, so functions such as `reduceByKey` and `groupByKey` are available. Finally, we call the `reduceByKey` method with the same implementation as we previously had in Hadoop. At the end of the transformation chain, we apply the filter to keep only frequent enough tokens.

You might have already noticed that this code is quite simple, compared to the Hadoop job we wrote previously.

Now we can put all the results into a `Map`: since we applied the filtering, we expect that the dictionary should easily fit into a memory even on modest hardware. We do it by using the `collectAsMap` function:

```
Map<String, Integer> dfs = dfRdd.collectAsMap();
```

Lastly, we go over all the documents again, and this time apply the TF-IDF weighting scheme to all the tokens:

```
JavaRDD<String> tfIdfRdd = textFile.map(line -> {
    String[] split = line.split("t");
    String url = split[0];
    List<String> tokens = Arrays.asList(split[1].split(" "));
    Multiset<String> counts = HashMultiset.create(tokens);

    String tfIdfTokens = counts.entrySet().stream()
           .map(e -> toTfIdf(dfs, e))
           .collect(Collectors.joining(" "));

    return url + "t" + tfIdfTokens;
});
```

We parse the input as we did previously and use `Multiset` from Guava for calculating TF. The `toTfIdf` function is exactly the same as before: it takes in an entry from `Multiset`, weights it by IDF, and outputs a string in the `token:weight` format.

To have a look at the results, we can take the first 10 tokens from the RDD and print them to stdout:

```
tfIdfRdd.take(10).forEach(System.out::println);
```

Finally, we save the results to a text file using the `saveAsTextFile` method:

```
tfIdfRdd.saveAsTextFile("c:/tmp/warc-tfidf");
```

As we see, we can do the same in Spark with significantly less code. What is more, it is also more efficient: it does not need to save the results to disk after each step and applies all the required transformations on-the-fly. This makes Spark a lot faster than Hadoop for many applications.

However, there are cases when Hadoop is better than Spark. Spark tries to keep everything in memory, and sometimes it fails with `OutOfMemoryException` because of this. Hadoop is a lot simpler: all it does is it writes to files, and then performa big distributed merge sort over the data. That being said, in general, you should prefer Apache Spark over Hadoop MapReduce, because Hadoop is slower and quite verbose.

In the next section, we will see how we can use Apache Spark and its Graph Processing and Machine Learning libraries for the Link Prediction problem.

Link prediction

Link Prediction is the problem of predicting which links will appear in a network. For example, we can have a friendship graph in Facebook or another social network, and functionality *like people you may know* is an application of Link Prediction. So, we can see Link Prediction is a recommendation system for social networks.

For this problem, we need to find a dataset that contains a graph evolving over time. Then, we can consider such a graph at some point in its evolution, calculate some characteristics between the existing links, and, based on that, predict which links are likely to appear next. Since for such graphs we know the future, we can use this knowledge for evaluating the performance of our models.

There are a number of interesting datasets available, but unfortunately, most of them do not have a time associated to the edges, so it is not possible to see how these graphs developed over time. This makes it harder to test the methods, but, of course, it is possible to do it without the time dimension.

Luckily, there are some datasets with timestamped edges. For this chapter, we will use the coauthorship graph based on the data from DBLP (`http://dblp.uni-trier.de/`) - a search engine that indexes computer science papers. The dataset is available from `http://projects.csail.mit.edu/dnd/DBLP/` (`dblp_coauthorship.json.gz` file) and it includes the papers from 1938 until 2015. It is already in the graph form: each edge is a pair of authors who published a paper together and each edge also contains the year when they did this.

This is what the first few lines of the file look like:

```
[
["Alin Deutsch", "Mary F. Fernandez", 1998],
["Alin Deutsch", "Daniela Florescu", 1998],
["Alin Deutsch", "Alon Y. Levy", 1998],
["Alin Deutsch", "Dan Suciu", 1998],
["Mary F. Fernandez", "Daniela Florescu", 1998],
```

Let's use this dataset to build a model that will predict who is very likely to become a coauthor in the future. One of the applications of such a model can be a recommender system: for each author, it may suggest the possible coauthors to cooperate with.

Reading the DBLP graph

To start with this project, we first need to read the graph data, and for this, we will use Apache Spark and a few of its libraries. The first library is Spark Data frames, it is similar to R data frames, pandas or joinery, except that they are distributed and based on RDDs.

Let's read this dataset. The first step is to create a special class `Edge` for storing the data:

```
public class Edge implements Serializable {
    private final String node1;
    private final String node2;
    private final int year;
    // constructor and setters omitted
}
```

Now, let's read the data:

```
SparkConf conf = new SparkConf().setAppName("graph").setMaster("local[*]");
JavaSparkContext sc = new JavaSparkContext(conf);
JavaRDD<String> edgeFile =
sc.textFile("/data/dblp/dblp_coauthorship.json.gz");

JavaRDD<Edge> edges = edgeFile.filter(s -> s.length() > 1).map(s -> {
    Object[] array = JSON.std.arrayFrom(s);

    String node1 = (String) array[0];
    String node2 = (String) array[1];
    Integer year = (Integer) array[2];

    if (year == null) {
        return new Edge(node1, node2, -1);
    }
```

```
        return new Edge(node1, node2, year);
    });
```

After setting up the context, we read the data from a text file, and then apply a map function to each line to convert it to `Edge`. For parsing JSON, we use the Jackson-Jr library as previously, so make sure you add this to the pom file.

Note that we also include a `filter` here: the first and the last line contain `[` and `]` respectively, so we need to skip them.

To check whether we managed to parse the data successfully, we can use the take method: it gets the head of the RDD and puts it into a `List`, which we can print to the console:

```
edges.take(5).forEach(System.out::println);
```

This should produce the following output:

```
Edge [node1=Alin Deutsch, node2=Mary F. Fernandez, year=1998]
Edge [node1=Alin Deutsch, node2=Daniela Florescu, year=1998]
Edge [node1=Alin Deutsch, node2=Alon Y. Levy, year=1998]
Edge [node1=Alin Deutsch, node2=Dan Suciu, year=1998]
Edge [node1=Mary F. Fernandez, node2=Daniela Florescu, year=1998]
```

After successfully converting the data, we will put it into a Data Frame. For that, we will use Spark DataFrame, which is a part of the Spark-SQL package. We can include it with the following dependency:

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.1.0</version>
</dependency>
```

To create a `DataFrame` from our `RDD`, we first create a SQL session, and then use its `createDataFrame` method:

```
SparkSession sql = new SparkSession(sc.sc());
Dataset<Row> df = sql.createDataFrame(edges, Edge.class);
```

There are quite a lot of papers in the dataset. We can make it smaller by restricting it to papers that were published only in `1990`. For this we can use the `filter` method:

```
df = df.filter("year >= 1990");
```

Next, many authors can have multiple papers together, and we are interested in the earliest one. We can get it using the `min` function:

```
df = df.groupBy("node1", "node2")
       .min("year")
       .withColumnRenamed("min(year)", "year");
```

When we apply the min function, the column gets renamed to `min(year)`, so we fix it by renaming the column back to `year` with the `withColumnRenamed` function.

For building any machine learning model, we always need to specify a train/test split. This case is no exception, so let's take all the data before 2013 as the training part, and all the papers after as testing:

```
Dataset<Row> train = df.filter("year <= 2013");
Dataset<Row> test = df.filter("year >= 2014");
```

Now, we can start extracting some features that we will use for creating a model.

Extracting features from the graph

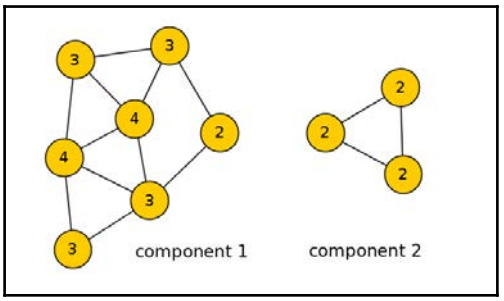
We need to extract some features that we will put to the Machine Learning model for training. For this dataset, all the information we have is the graph itself and nothing more: we do not have any external information such as author's affiliation. Of course, if we had it, it would be no problem to add it to the model. So let's discuss which features we can extract from the graph alone.

For graph models, there can be two kinds of features: node features (authors) and edge features (the coauthorship relation).

There are many possible features we can extract from graph nodes. For example, among others, we can consider the following:

- **Degree**: This is the number of coauthors this author has.
- **Page Rank**: This is the importance of a node.

Let's look at the following diagram:



Here we have two connected components, and the number on the node specifies the degree of the node, or how many connections it has.

In some way, the degree measures the importance (or centrality) of a node: the more connections it has, the higher the importance. Page Rank (also called Eigenvector Centrality) is another measure of importance. You have probably heard about Page Rank - it is used by Google as one of the components of its ranking formula. The main idea behind Page Rank is that, if a page is linked to other important pages, it also must be important. Sometimes it also makes sense to use Chei Rank, the reverse of Page Rank-- which instead of looking at incoming edges, looks at outcoming ones.

However, our graph is not directed: if *A* and *B* are coauthors, then *B* and *A* are also coauthors. So, in our case Page Rank and Chei Rank are exactly the same.

There are other important measures such as Closeness Centrality or Betweenness Centrality, but we will not consider them in this chapter.

Also, we can look at the connected component of a node: if two nodes are from different connected components, it is often difficult to predict whether there is going to be a link between them. (Of course, it becomes possible if we include other features, not only the ones we can extract from the graph.)

Graphs also have edges, and we can extract a lot of information about them for building our models. For example, we may consider the following features:

- **Common Friends**: This is the number of common coauthors
- **Total Friends**: This is the total number of distinct coauthors both authors have
- **Jaccard Similarity**: This is the Jaccard similarity of the coauthor's sets
- **Node-based features** This is the difference in the Page Rank of each node, min and max degree, and so on

Of course there are a lot of other features we can include, for example, the length of the shortest path between two authors should be quite a strong predictor, but calculating it typically requires a lot of time.

Node features

Let's first concentrate on the features with which we can compute nodes of a graph. For that, we will need a graph library which lets us compute graph features such as degree or Page Rank easily. For Apache Spark, such a library is GraphX. However, at the moment, this library only supports Scala: it uses a lot of Scala-specific features, which makes it very hard (and often impossible) to use it from Java.

However, there is another library called GraphFrames, which tries to combine GraphX with DataFrames. Luckily for us, it supports Java. This package is not available on Maven Central, and to use it, we first need to add the following repository to our `pom.xml`:

```
<repository>
  <id>bintray-spark</id>
  <url>https://dl.bintray.com/spark-packages/maven/</url>
</repository>
```

Next, let's include the library:

```
<dependency>
  <groupId>graphframes</groupId>
  <artifactId>graphframes</artifactId>
  <version>0.3.0-spark2.0-s_2.11</version>
</dependency>
```

We also need to add the GraphX dependency, because GraphFrames relies on it:

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-graphx_2.11</artifactId>
  <version>2.1.0</version>
</dependency>
```



The GraphFrames library is under active development at the moment, and it is quite likely that some of the method names will change in the future. Refer to the official documentation from `http://graphframes.github.io/`, if, the examples from this chapter stop working.

However, before we can actually use GraphFrames, we need to prepare our data, as it expects the Data Frames to follow a specific convention. First, it assumes that the graph is directed, which is not the case for our example. To overcome this problem, we need to add the reverse links. That is, we have a record `(A, B)` in our dataset, we need to add its reverse `(B, A)`.

We can do this by renaming the columns of a copy of our DataFrame and then use the union function with the original DataFrame:

```
Dataset<Row> dfReversed = df
        .withColumnRenamed("node1", "tmp")
        .withColumnRenamed("node2", "node1")
        .withColumnRenamed("tmp", "node2")
        .select("node1", "node2", "year")
Dataset<Row> edges = df.union(dfReversed);
```

Next, we need to create a special `DataFrame` for the nodes. We can do this by just selecting the `node1` column of the edges dataset and then calling the distinct function:

```
Dataset<Row> nodes = edges.select("node1")
        .withColumnRenamed("node1", "node")
        .distinct();
```

GraphFrame, when taking in a `DataFrame` with nodes, expects it to have the `id` column, which we have to manually create. The first option is to just rename the `node` column to `id` and pass this to GraphFrame. Another option is to create surrogate IDs, which we will do here:

```
nodes = nodes.withColumn("id", functions.monotonicallyIncreasingId());
```

For the preceding code, we need to add the following import:

```
import org.apache.spark.sql.functions;
```

This `functions` is a utility class with a lot of useful DataFrame functions.

To see what our DataFrame looks like after all these transformations, we can use the `show` method:

```
nodes.show();
```

It will produce the following output (truncated to six first rows here and in all examples):

```
+--------------------+---+
|                node| id|
+--------------------+---+
|         Dan Olteanu|  0|
```

```
|        Manjit Borah|  1|
|    Christoph Elsner|  2|
|         Sagnika Sen|  3|
|          Jerome Yen|  4|
|        Anand Kudari|  5|
|              M. Pan|  6|
+--------------------+---+
```

Now, let's prepare the edges. GraphFrames requires the data frame to have two special columns: `src` and `dst` (*source* and *destination*, respectively). To get the values for these columns, let's join them with the nodes DataFrame and get the numerical IDs:

```
edges = edges.join(nodes, edges.col("node2").equalTo(nodes.col("node")));
edges = edges.drop("node").withColumnRenamed("id", "dst");
edges = edges.join(nodes, edges.col("node1").equalTo(nodes.col("node")));
edges = edges.drop("node").withColumnRenamed("id", "src");
```

It will create a DataFrame with the following content:

```
+-------------+--------------------+----+-------------+----+
|        node1|               node2|year|          dst| src|
+-------------+--------------------+----+-------------+----+
|A. A. Davydov| Eugene V. Shilnikov|2013|  51539612101|2471|
|A. A. Davydov|      S. V. Sinitsyn|2011| 326417520647|2471|
|A. A. Davydov|      N. Yu. Nalutin|2011| 335007452466|2471|
|A. A. Davydov|        A. V. Bataev|2011| 429496733302|2471|
|A. A. Davydov|Boris N. Chetveru...|2013|1486058685923|2471|
| A. A. Sawant|          M. K. Shah|2011| 231928238662|4514|
| A. A. Sawant|      A. V. Shingala|2011| 644245100670|4514|
+-------------+--------------------+----+-------------+----+
```

Finally, we can create `GraphFrame` from the nodes and edges dataframes:

```
GraphFrame gf = GraphFrame.apply(nodes, edges);
```

The `GraphFrame` class allows us to use a lot of graph algorithms. For example, computing Page Rank is as easy as follows:

```
GraphFrame pageRank = gf.pageRank().resetProbability(0.1).maxIter(7).run();
Dataset<Row> pageRankNodes = pageRank.vertices();
pageRankNodes.show();
```

It will create a `DataFrame` with the following columns:

```
+----+-------------------+
|  id|           pagerank|
+----+-------------------+
|  26| 1.4394843416065657|
```

```
|  29|  1.012233852957335|
| 474| 0.7774103396731716|
| 964| 0.4443614094552203|
|1677|  0.274044687604839|
|1697|  0.493174385163372|
+----+-------------------+
```

Calculating degrees is even simpler:

```
Dataset<Row> degrees = gf.degrees();
```

This line of code will create a `DataFrame` with the degree of each node:

```
+-------------+------+
|           id|degree|
+-------------+------+
| 901943134694|    86|
| 171798692537|     4|
|1589137900148|   114|
|   8589935298|    86|
| 901943133299|    74|
| 292057778121|    14|
+-------------+------+
```

The same is true for calculating connected components:

```
Dataset<Row> cc = gf.connectedComponents().run();
```

It will create a `DataFrame` where for each node we will have the ID of the connected component it belongs to:

```
+----+---------+
|  id|component|
+----+---------+
|  26|        0|
|  29|        0|
| 474|        0|
| 964|      964|
|1677|        0|
|1697|        0|
+----+---------+
```

As we see here 5 out of 6 first components are the 0th component. Let's look at the sizes of these components--maybe the 0th is the largest and includes almost everything?

To do it, we can count how many times each component occurs:

```
Dataset<Row> cc = connectedComponents.groupBy("component").count();
cc.orderBy(functions.desc("count")).show();
```

We use the `groupBy` function and then invoke the `count` method. After that, we order the DataFrame by the values of the count column in the decreasing order. Let's look at the output:

```
+------------+-------+
|   component|  count|
+------------+-------+
|           0|1173137|
| 60129546561|     32|
| 60129543093|     30|
|         722|     29|
| 77309412270|     28|
| 34359740786|     28|
+------------+-------+
```

As we see, most of the nodes are indeed from the same component. Thus, in this case, the information about the component is not very useful: almost always the component is `0`.

Now, after computing the node features, we need to proceed to edge features. But before we can do that, we first need to sample edges for which we are going to compute these features.

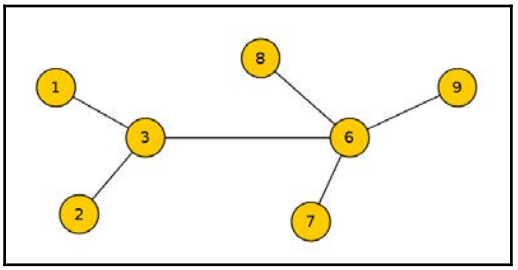
Negative sampling

Before we compute another set of features, the edge features, we need to first specify which edges we would like to take for that. So we need to select a set of candidate edges, and then we will train a model on them for predicting whether an edge should belong to the graph or not. In other words, we first need to prepare a dataset where existent edges are treated as positive examples, and nonexistent ones as negative.

Getting positive examples is simple: we just take all the edges and assign them the label `1`.

For negative examples, it is more complex: in any real-life graph, the number of positive examples is a lot smaller than the number of negative examples. So we need to find a way to sample the negative examples so that training a model becomes manageable.

Often, for the Link Prediction problems, we consider two types of negative candidates: simple and hard ones. The simple ones are just sampled from the same connected components, but the hard ones are one or two hops away. Since for our problem most of the authors are from the same component, we can relax it and sample the simple negatives from the entire graph, and not restrict ourselves to the same connected component:



If we consider the preceding graph, positive examples are easy: we just get the existent edges `(1, 3)`, `(2, 3)`, `(3, 6)`, and so on. For negative ones, there are two types: `simple` and `hard`. The simple ones are just sampled from the set of all possible nonexistent edges. Here, it can be `(1, 8)`, `(2, 7)`, or `(1, 9)`. The hard negative edges are only one hop away: `(1, 2)`, `(1, 6)`, or `(7, 9)` are possible examples of hard negatives.

In our dataset, we have about 6 million positive examples. To keep the training data more or less balanced, we can sample about 12 million simple negatives and about 6 million hard ones. Then the proportion of positive to negative examples will be 1/4.

Creating positive examples is straightforward: we just take the edges from the graph and assign them the `1.0` target:

```
Dataset<Row> pos = df.drop("year");
pos = pos.join(nodes, pos.col("node1").equalTo(nodes.col("node")));
pos = pos.drop("node", "node1").withColumnRenamed("id", "node1");

pos = pos.join(nodes, pos.col("node2").equalTo(nodes.col("node")));
pos = pos.drop("node", "node2").withColumnRenamed("id", "node2");

pos = pos.withColumn("target", functions.lit(1.0));
```

Here we do a few joins to replace the author names with their IDs. The result is the following:

```
+-------------+-----+------+
|        node1|node2|target|
+-------------+-----+------+
|  51539612101| 2471|   1.0|
```

```
| 429496733302| 2471|   1.0|
|1486058685923| 2471|   1.0|
|1254130450702| 4514|   1.0|
|  94489280742|  913|   1.0|
|1176821039357|  913|   1.0|
+-------------+-----+------+
```

Next, we sample the easy negative ones. For that, we first sample with replacement from the node's DataFrame twice, once for `node1` and once - for `node2`. Then, put these columns together into one single data frame. We can take the samples this way:

```
Dataset<Row> nodeIds = nodes.select("id");
long nodesCount = nodeIds.count();
double fraction = 12_000_000.0 / nodesCount;

Dataset<Row> sample1 = nodeIds.sample(true, fraction, 1);
sample1 = sample1.withColumn("rnd", functions.rand(1))
                 .orderBy("rnd")
                 .drop("rnd");

Dataset<Row> sample2 = nodeIds.sample(true, fraction, 2);
sample2 = sample2.withColumn("rnd", functions.rand(2))
                 .orderBy("rnd")
                 .drop("rnd");
```

Here, the fraction parameter specifies what is the fraction of the `DataFrame` which the sample should contain. Since we want to get 12 million examples, we divide 12 million by the number of nodes we have. Then, we shuffle each sample by adding a column with random number to it and use it for ordering the `DataFrame`s. We don't need this column after sorting is done, so it can be dropped.

It is possible that two samples have different sizes, so we need to select the minimal one, and then limit both samples to this size, so they become concatenable:

```
long sample1Count = sample1.count();
long sample2Count = sample2.count();

int minSize = (int) Math.min(sample1Count, sample2Count);

sample1 = sample1.limit(minSize);
sample2 = sample2.limit(minSize);
```

Next, we want to put these two samples together into one DataFrame. There is no easy way to do this with the DataFrame API, so we will need to use RDDs for that. To do it, we convert the `DataFrame`s into `JavaRDD`s, `zip` them together, and then convert the result back to a single `DataFrame`:

```
JavaRDD<Row> sample1Rdd = sample1.toJavaRDD();
JavaRDD<Row> sample2Rdd = sample2.toJavaRDD();
JavaRDD<Row> concat = sample1Rdd.zip(sample2Rdd).map(t -> {
    long id1 = t._1.getLong(0);
    long id2 = t._2.getLong(0);
    return RowFactory.create(id1, id2);
});

StructField node1Field = DataTypes.createStructField("node1",
DataTypes.LongType, false);
StructField node2Field = DataTypes.createStructField("node2",
DataTypes.LongType, false);
StructType schema = DataTypes.createStructType(Arrays.asList(node1Field,
node2Field));
Dataset<Row> negSimple = sql.createDataFrame(concat, schema);
```

For converting the `RDD` into a `DataFrame`, we need to specify a schema, and the preceding code shows how to do it.

Finally, we add the target column to this `DataFrame`:

```
negSimple = negSimple.withColumn("target", functions.lit(0.0));
```

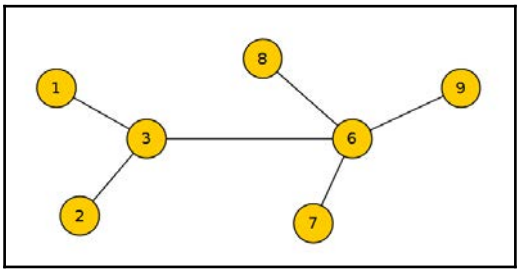
This will generate the following `DataFrame`:

```
+-------------+-------------+------+
|        node1|        node2|target|
+-------------+-------------+------+
| 652835034825|1056561960618|   0.0|
| 386547056678| 446676601330|   0.0|
| 824633725362|1477468756129|   0.0|
|1529008363870| 274877910417|   0.0|
| 395136992117| 944892811576|   0.0|
|1657857381212|1116691503444|   0.0|
+-------------+-------------+------+
```



It is possible that by sampling this way we accidentally generate pairs that happen to be among positive examples. However, the probability of this is quite low and it can be discarded.

We also need to create hard negative examples - these are examples that are just one jump away from each other:



In this figure, as we already discussed, the `(1, 2)`, `(1, 6)`, or `(7, 9)` pairs are examples of hard negative examples.

To get such pairs, let's first formulate the idea logically. We need to sample all the possible pairs `(B, C)` such that there exist some node `A` and edges `(A, B)` and `(A, C)` both exist, but there is no edge `(B, C)`.

When we phrase this sampling problem in such a way, it becomes easy to express this with SQL: all we need to do is a self-join and select edges with the same source, but a different destination. Consider the following example:

```
SELECT e1.dst as node1, e2.dst as node2
  FROM Edges e1, Edges e2
 WHERE e1.src = e2.src ANDe1.dst <> e2.dst;
```

Let's translate it to the Spark DataFrame API. To do a self-join, we first need to create two aliases of the edge `DataFrame` and rename the columns inside:

```
Dataset<Row> e1 = edges.drop("node1", "node2", "year")
        .withColumnRenamed("src", "e1_src")
        .withColumnRenamed("dst", "e1_dst")
        .as("e1");
Dataset<Row> e2 = edges.drop("node1", "node2", "year")
        .withColumnRenamed("src", "e2_src")
        .withColumnRenamed("dst", "e2_dst")
        .as("e2");
```

Now, we perform the join on condition that `dst` is different, but `src` is the same, and then rename the columns so they are consistent with the previous samples:

```
Column diffDest = e1.col("e1_dst").notEqual(e2.col("e2_dst"));
Column sameSrc = e1.col("e1_src").equalTo(e2.col("e2_src"));
Dataset<Row> hardNeg = e1.join(e2, diffDest.and(sameSrc));
```

```
hardNeg = hardNeg.select("e1_dst", "e2_dst")
        .withColumnRenamed("e1_dst", "node1")
        .withColumnRenamed("e2_dst", "node2");
```

Next, we need to take the first 6 million generated edges and call it the hard sample. However, Spark puts the values in this `DataFrame` in some particular order, which may introduce bias into our model. To make the bias less harmful, let's add some randomness to the sampling process: generate a column with random values and take only those edges where the value is greater than some number:

```
hardNeg = hardNeg.withColumn("rnd", functions.rand(0));
hardNeg = hardNeg.filter("rnd >= 0.95").drop("rnd");
hardNeg = hardNeg.limit(6_000_000);
hardNeg = hardNeg.withColumn("target", functions.lit(0.0));
```

After that, we just take the first 6m edges, and add the `target` column. The results follow the same schema as our previous samples:

```
+------------+-------------+------+
|       node1|        node2|target|
+------------+-------------+------+
| 34359740336| 970662610852|   0.0|
| 34359740336| 987842479409|   0.0|
| 34359740336|1494648621189|   0.0|
| 34359740336|1554778161775|   0.0|
| 42949673538| 326417515499|   0.0|
|266287973882| 781684049287|   0.0|
+------------+-------------+------+
```

Putting them together is done with the `union` function:

```
Dataset<Row> trainEdges = pos.union(negSimple).union(hardNeg);
```

Finally, let's associate a ID with every edge:

```
trainEdges = trainEdges.withColumn("id",
functions.monotonicallyIncreasingId());
```

With this, we have prepared the edges for which we can compute the edge features.

Edge features

There are a number of edge features that we can compute: the number of common friends, the total number of distinct friends both people have, and so on.

Let us start with the common friend's feature, which for our problem is the number of common coauthors both authors have. To get them we need to join the edges we selected with all the edges (two times) and then group by the ID and count how many elements are there per group. In SQL, it looks like this:

```
  SELECT train.id, COUNT(*)
    FROM Sample train, Edges e1, Edges e2
   WHERE train.node1 = e1.src AND
         train.node2 = e2.src AND
         e1.dst = e2.dst
GROUP BY train.id;
```

Let's translate this to DataFrame API. First, we use the joins:

```
Dataset<Row> join = train.join(e1,
        train.col("node1").equalTo(e1.col("e1_src")));
join = join.join(e2,
        join.col("node2").equalTo(e2.col("e2_src")).and(
        join.col("e1_dst").equalTo(e2.col("e2_dst"))));
```

Here, we reuse DataFrames `e1` and `e2` from the negative sampling subsection.

Then, we finally group by the `id` and count:

```
Dataset<Row> commonFriends = join.groupBy("id").count();
commonFriends = commonFriends.withColumnRenamed("count", "commonFriends");
```

The resulting `DataFrame` will contain the following:

```
+-------------+-------------+
|           id|commonFriends|
+-------------+-------------+
|1726578522049|          116|
|        15108|            1|
|1726581250424|          117|
|        17579|            4|
|         2669|           11|
|         3010|           73|
+-------------+-------------+
```

Now we calculate the Total Friends feature, which is the number of distinct coauthors both authors have. In SQL, it is a bit simpler than the previous feature:

```
  SELECT train.id, COUNT DISTINCT (e.dst)
    FROM Sample train, Edges e
   WHERE train.node1 = e.src
GROUP BY train.id
```

Let's now translate it to Spark:

```
Dataset<Row> e = edges.drop("node1", "node2", "year", "target");
Dataset<Row> join = train.join(e,
        train.col("node1").equalTo(edges.col("src")));
totalFriends = join.select("id", "dst")
        .groupBy("id")
.agg(functions.approxCountDistinct("dst").as("totalFriendsApprox"));
```

Here, we used the approximate count distinct because it is faster and typically gives quite accurate values. Of course, there is an option to use exact count distinct: for that, we need to use the `functions.countDistinct` function. The output of this step is the following table:

```
+-------------+------------------+
|           id|totalFriendsApprox|
+-------------+------------------+
|1726580872911|                 4|
| 601295447985|                 4|
|1726580879317|                 1|
| 858993461306|                11|
|1726578972367|               296|
|1726581766707|               296|
+-------------+------------------+
```

Next, we calculate the Jaccard Similarity between the two sets of coauthors. We first create a `DataFrame` with sets for each author, then join and calculate the jaccard. For this feature there's no direct way to express it in SQL, so we start from Spark API.

Creating the set of coauthors for each author is easy: we just use the `groupBy` function and then apply `functions.collect_set` to each group:

```
Dataset<Row> coAuthors = e.groupBy("src")
        .agg(functions.collect_set("dst").as("others"))
        .withColumnRenamed("src", "node");
```

Now we join it with our training data:

```
Dataset<Row> join = train.drop("target");

join = join.join(coAuthors,
join.col("node1").equalTo(coAuthors.col("node")));
join = join.drop("node").withColumnRenamed("others", "others1");

join = join.join(coAuthors,
join.col("node2").equalTo(coAuthors.col("node")));
join = join.drop("node").withColumnRenamed("others", "others2");
```

```
join = join.drop("node1", "node2");
```

At the end, the join column has the ID of the edge and the arrays of the coauthors for each edge. Next, we go over every record of this dataframe and compute the Jaccard similarity:

```
JavaRDD<Row> jaccardRdd = join.toJavaRDD().map(r -> {
    long id = r.getAs("id");
    WrappedArray<Long> others1 = r.getAs("others1");
    WrappedArray<Long> others2 = r.getAs("others2");

    Set<Long> set1 = Sets.newHashSet((Long[]) others1.array());
    Set<Long> set2 = Sets.newHashSet((Long[]) others2.array());

    int intersection = Sets.intersection(set1, set2).size();
    int union = Sets.union(set1, set2).size();
    double jaccard = intersection / (union + 1.0);
    return RowFactory.create(id, jaccard);
});
```

Here we use the **regularized Jaccard Similarity**: instead of just dividing intersection by union, we also add a small regularization factor to the denominator.

The reason for doing it is to give less score to very small sets: imagine that each set has the same element, then the Jaccard is 1.0. With regularization, the similarity of small lots is penalized, and for this example, it will be equal to 0.5.

Since we used RDDs here, we need to convert it back to a data frame:

```
StructField node1Field = DataTypes.createStructField("id",
DataTypes.LongType, false);
StructField node2Field = DataTypes.createStructField("jaccard",
DataTypes.DoubleType, false);
StructType schema = DataTypes.createStructType(Arrays.asList(node1Field,
node2Field));
Dataset<Row> jaccard = sql.createDataFrame(jaccardRdd, schema);
```

After executing it, we get a table like this:

```
+-------------+---------+
|           id|  jaccard|
+-------------+---------+
|1726581480054|    0.011|
|1726578955032|    0.058|
|1726581479913|    0.037|
|1726581479873|     0.05|
|1726581479976|      0.1|
|         1667|      0.1|
+-------------+---------+
```

Note that the preceding method is quite universal, and we could follow the same approach for calculating the *Common Friends* and *Total Friends* features: it would be the size of the intersection and union respectively. What is more, it could be computed in one pass along with Jaccard.

Our next step is to deprive edge features from the node features we already computed. Among others, we can include the following:

- Min degree, max degree
- Preferential attachment score: degree of node1 times degree of node2
- Product of page ranks
- Absolute difference of page ranks
- Same connected component

To do it we first join all the node features together:

```
Dataset<Row> nodeFeatures = pageRank.join(degrees, "id")
                                    .join(connectedComponents, "id");
nodeFeatures = nodeFeatures.withColumnRenamed("id", "node_id");
```

Next, we join the node features `DataFrame` we just created with the edges we prepared for training:

```
Dataset<Row> join = train.drop("target");

join = join.join(nodeFeatures,
                 join.col("node1").equalTo(nodeFeatures.col("node_id")));
join = join.drop("node_id")
        .withColumnRenamed("pagerank", "pagerank_1")
        .withColumnRenamed("degree", "degree_1")
        .withColumnRenamed("component", "component_1");

join = join.join(nodeFeatures,
                 join.col("node2").equalTo(nodeFeatures.col("node_id")));
join = join.drop("node_id")
        .withColumnRenamed("pagerank", "pagerank_2")
        .withColumnRenamed("degree", "degree_2")
        .withColumnRenamed("component", "component_2");

join = join.drop("node1", "node2");
```

Now, let's calculate the features:

```
join = join
    .withColumn("pagerank_mult",
join.col("pagerank_1").multiply(join.col("pagerank_2")))
    .withColumn("pagerank_max", functions.greatest("pagerank_1",
"pagerank_2"))
    .withColumn("pagerank_min", functions.least("pagerank_1",
"pagerank_2"))
    .withColumn("pref_attachm",
join.col("degree_1").multiply(join.col("degree_2")))
    .withColumn("degree_max", functions.greatest("degree_1", "degree_2"))
    .withColumn("degree_min", functions.least("degree_1", "degree_2"))
    .withColumn("same_comp",
join.col("component_1").equalTo(join.col("component_2")));
join = join.drop("pagerank_1", "pagerank_2");
join = join.drop("degree_1", "degree_2");
join = join.drop("component_1", "component_2");
```

This will create a `DataFrame` with edge ID and seven features: min and max Page Rank, a product of two Page Ranks, min and max degree, a product of two degrees (Preferential Attachment), and, finally, whether or not two nodes belong to the same component.

Now we have finished calculating all the features we wanted, so it's time we join them all together in a single `DataFrame`:

```
Dataset<Row> join = train.join(commonFriends, "id")
     .join(totalFriends, "id")
     .join(jaccard, "id")
     .join(nodeFeatures, "id");
```

So far we created the dataset with labels and computed a set of features. Now we are finally ready to train a machine learning model on it.

Link Prediction with MLlib and XGBoost

Now, when all the data is prepared and put into a suitable shape, we can train a model, which will predict whether two authors are likely to become coauthors or not. For that we will use a binary classifier model, which will be trained to predict what is the probability that this edge exists in a graph.

Apache Spark comes with a library which provides scalable implementation of several Machine Learning algorithms. This library is called MLlib. Let's add it to our `pom.xml`:

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-mllib_2.11</artifactId>
  <version>2.1.0</version>
</dependency>
```

There are a number of models we can use, including logistic regression, random forest, and Gradient Boosted Trees. But before we train any model, let's split the training dataset into train and validation sets:

```
features = features.withColumn("rnd", functions.rand(1));
Dataset<Row> trainFeatures = features.filter("rnd < 0.8").drop("rnd");
Dataset<Row> valFeatures = features.filter("rnd >= 0.8").drop("rnd");
```

To be able to use it for training machine learning models, we need to convert our data to RDD of `LabeledPoint` objects. For that, we first convert the DataFrame to RDD, and then convert each row to `DenseVector`:

```
List<String> columns = Arrays.asList("commonFriends", "totalFriendsApprox",
        "jaccard", "pagerank_mult", "pagerank_max", "pagerank_min",
        "pref_attachm", "degree_max", "degree_min", "same_comp");

JavaRDD<LabeledPoint> trainRdd = trainFeatures.toJavaRDD().map(r -> {
    Vector vec = toDenseVector(columns, r);
    double label = r.getAs("target");
    return new LabeledPoint(label, vec);
});
```

The `columns`stores all the column names we want to use as features in the order we want to put them into `DenseVector`. The `toDenseVector` function has the following implementation:

```
private static DenseVector toDenseVector(List<String> columns, Row r) {
    int featureVecLen = columns.size();
    double[] values = new double[featureVecLen];
    for (int i = 0; i < featureVecLen; i++) {
        Object o = r.getAs(columns.get(i));
        values[i] = castToDouble(o);
    }
    return new DenseVector(values);
}
```

Since, in our DataFrame we have data of multiple types, including `int`, `double`, and `boolean`, we need to be able to convert all of them to double. This is what the `castToDouble` function does:

```
private static double castToDouble(Object o) {
    if (o instanceof Number) {
        Number number = (Number) o;
        return number.doubleValue();
    }

    if (o instanceof Boolean) {
        Boolean bool = (Boolean) o;
        if (bool) {
            return 1.0;
        } else {
            return 0.0;
        }
    }

    throw new IllegalArgumentException();
}
```

Now we finally can train the logistic regression model:

```
LogisticRegressionModel logreg = new LogisticRegressionWithLBFGS()
            .run(JavaRDD.toRDD(trainRdd));
```

After it finishes, we can evaluate how good the model is.

Let's go through the entire validation dataset and make a prediction for each element there:

```
logreg.clearThreshold();

JavaRDD<Pair<Double, Double>> predRdd = valFeatures.toJavaRDD().map(r -> {
    Vector v = toDenseVector(columns, r);
    double label = r.getAs("target");
    double predict = logreg.predict(v);
    return ImmutablePair.of(label, predict);
});
```

Note that we first need to invoke the `clearThreshold` method - if we don't do this, then the model will output hard predictions (only 0.0 and 1.0), which will make the evaluation more difficult.

Now we can put the predictions and the true labels into separate double arrays and use any of the binary classification evaluation functions, which we covered in `Chapter 4`, *Supervised Learning - Classification and Regression*. For example, we can use `logLoss`:

```
List<Pair<Double, Double>> pred = predRdd.collect();
double[] actual = pred.stream().mapToDouble(Pair::getLeft).toArray();
double[] predicted = pred.stream().mapToDouble(Pair::getRight).toArray();
double logLoss = Metrics.logLoss(actual, predicted);
System.out.printf("log loss: %.4f%n", logLoss);
```

This produces the following output:

```
log loss: 0.6528
```

This is not a very good performance: if we always output `0.5` as the prediction, the `logLoss` would be `0.7`, so our model is just a bit better than that. We can try other models available in MLlib such as linear SVM or random forest to see whether they give better performance.

But there is another option: if you remember from `Chapter 7`, *Extreme Gradient Boosting* XGBoost, can also run in parallel mode, and it can use Apache Spark for doing it. So let's try to use it for this problem. To see how to build XGBoost, refer to `Chapter 7`, *Extreme Gradient Boosting*.

To include the Spark version to our project, we add the following dependency declaration to the project:

```
<dependency>
  <groupId>ml.dmlc</groupId>
  <artifactId>xgboost4j-spark</artifactId>
  <version>0.7</version>
</dependency>
```

As input, XGBoost also takes `RDD`s of `Vector` objects. Apart from that, it takes the same parameters as XGBoost running on a single machine: model parameters, number of trees to build, and so on. This is how it looks in the code:

```
Map<String, Object> params = xgbParams();
int nRounds = 20;
int numWorkers = 4;
ObjectiveTrait objective = null;
EvalTrait eval = null;
boolean externalMemoryCache = false;
float nanValue = Float.NaN;
RDD<LabeledPoint> trainData = JavaRDD.toRDD(trainRdd);
```

```
XGBoostModel model = XGBoost.train(trainData, params,
        nRounds, numWorkers, objective, eval, externalMemoryCache,
        nanValue);
```

Here, the `xgbParams` function returns a `Map` of XGBoost model parameters we use for training.

Note that XGBoost Spark wrapper is written in Scala, not in Java, so the `Map` object is actually `scala.collection.immutable.Map`, not `java.util.Map`. Thus, we also need to convert a usual `HashMap` into the Scala `Map`:

```
HashMap<String, Object> params = new HashMap<String, Object>();
params.put("eta", 0.3);
params.put("gamma", 0);
params.put("max_depth", 6);
// ... other parameters
Map<String, Object> res = toScala(params);
```

Here, the `toScala` utility method is implemented in this way:

```
private static <K, V> Map<K, V> toScala(HashMap<K, V> params) {
    return JavaConversions.mapAsScalaMap(params)
            .toMap(Predef.<Tuple2<K, V>>conforms());
}
```

It looks a bit strange because it uses some Scala-specific features. But we don't need to go into details and can use it as is.

With this, we will be able to train a distributed XGBoost. However, for evaluation, we cannot follow the same approach as for logistic regression. That is, we cannot convert each row to a vector, and then run the model against this vector, and if you do so, XGBoost will throw an exception. The reason for this is that such an operation is quite costly, as it will try to build `DMatrix` for each vector, and it will result in a significant slowdown.

Since our validation dataset is not very large, we can just convert the entire `RDD` to a `DMatrix`:

```
JavaRDD<LabeledPoint> valRdd = valFeatures.toJavaRDD().map(r -> {
    float[] vec = rowToFloatArray(columns, r);
    double label = r.getAs("target");
    return LabeledPoint.fromDenseVector((float) label, vec);
});

List<LabeledPoint> valPoints = valRdd.collect();
DMatrix data = new DMatrix(valPoints.iterator(), null);
```

Here, we go through the rows of the `DataFrame`, and convert each row to a `LabeledPoint` class (from the `ml.dmlc.xgboost4j` package - not to be confused with `org.apache.spark.ml.feature.LabeledPoint`). Then, we collect everything to a list and create a `DMatrix` from it.

Next, we get the trained model and apply it to this `DMatrix`:

```
Booster xgb = model._booster();
float[][] xgbPred = xgb.predict(new ml.dmlc.xgboost4j.scala.DMatrix(data),
false, 20);
```

Then, our method for `logLoss` calculations expects to get doubles as input, so let us convert the results to arrays of doubles:

```
double[] actual = floatToDouble(data.getLabel());
double[] predicted = unwrapToDouble(xgbPred);
```

We have already used these functions in `Chapter 7`, *Extreme Gradient Boosting*, and they are pretty straightforward: `floatToDouble` just converts a float array to a double array, and `unwrapToDouble` converts two-dimensional float arrays with one column to 1-dimensional double array.

Finally, we can calculate the score:

```
double logLoss = Metrics.logLoss(actual, predicted);
System.out.printf("log loss: %.4f%n", logLoss);
```

It says that the score is 0.497, which is a huge improvement over what we had previously. Here, we used the model with default parameters, which are often not the most optimal. We can tune the model further, and you can find strategies on how to tune XGBoost in `Chapter 7`, *Extreme Gradient Boosting*.

`logLoss` was chosen here only for its simplicity and often it is hard to interpret when it comes to recommender systems. Choosing an evaluation metric is usually quite case-specific, and we can use scores such as F1 score, **MAP** (**Mean Average Precision**), **NDCG** (**Normalized Discounted Cumulative Gain**), and many others. In addition to that, we can use on-line evaluation metrics such as how many suggested links are accepted by the users.

Next, we will see how this model can be used for suggesting links and how we can evaluate it better.

Link suggestion

So far we have discussed in a lot of details about building features for Link Prediction models and training these models. Now we need to be able to use such models for making suggestions. Again, think about *people you may know* banner on Facebook - here we would like to have something similar, like *authors you should write a paper with*. In addition to that, we will see how to evaluate the model so the evaluation result is more intuitive and clear in this context.

The first step is to re-train the XGBoost model on the entire training set, without the train-validation split. This is easy to do: we just fit the model again on the data before making the split.

Next, we need to process the test dataset. If you remember, the test dataset contains all the papers published in 2014 and 2015. To make things simpler, we will select a subset of test users, and make recommendations to them only. This is how we can do it:

```
Dataset<Row> fullTest = df.filter("year >= 2014");
Dataset<Row> testNodes = fullTest.sample(true, 0.05, 1)
                                 .select("node1")
                                 .dropDuplicates();
Dataset<Row> testEdges = fullTest.join(testNodes, "node1");
```

Here we first select the test set, and then sample nodes from it - and this gives us a list of authors we selected for testing. In other words, only these authors will receive recommendations. Next, we perform the join of the full test set with the selected nodes to get all the actual links that were made during the testing period. We will use these links later for evaluation.

Next, we replace the names of the authors with IDs. We do it in the same way we did previously - by joining it with the nodes `DataFrame`:

```
Dataset<Row> join = testEdges.drop("year");
join = join.join(nodes, join.col("node1").equalTo(nodes.col("node")));
join = join.drop("node", "node1").withColumnRenamed("id", "node1");
join = join.join(nodes, join.col("node2").equalTo(nodes.col("node")));
join = join.drop("node", "node2").withColumnRenamed("id", "node2");
Dataset<Row> selected = join;
```

Next, we need to select candidates on who we will apply the model. The list of candidates should contain a list of authors who are most likely to become a potential coauthor (that is, form a link in the network). The most obvious way to select such candidates is to take the authors that are one jump away from each other - in the same way, we sampled hard negative links:

```
Dataset<Row> e1 = selected.select("node1").dropDuplicates();
Dataset<Row> e2 = edges.drop("node1", "node2", "year")
        .withColumnRenamed("src", "e2_src")
        .withColumnRenamed("dst", "e2_dst")
        .as("e2");
Column diffDest = e1.col("node1").notEqual(e2.col("e2_dst"));
Column sameSrc = e1.col("node1").equalTo(e2.col("e2_src"));
Dataset<Row> candidates = e1.join(e2, diffDest.and(sameSrc));
candidates = candidates.select("node1", "e2_dst")
                       .withColumnRenamed("e2_dst", "node2");
```

The code is almost the same except that we do not consider all possible hard negatives, but only those that are related to the nodes we have preselected.

We assume that these candidates did not become coauthors during testing period, so we add the target column with `0.0` in it:

```
candidates = candidates.withColumn("target", functions.lit(0.0));
```

While in general this assumption will not hold, because we look at the link obtained only from the training period, and it might be quite possible that some of these links were actually formed during the testing period.

Let us fix it by manually adding the positive candidates, and then removing duplicates:

```
selected = selected.withColumn("target", functions.lit(1.0));
candidates = selected.union(candidates).dropDuplicates("node1", "node2");
```

Now, as we did previously, we assign an ID to each candidate edge:

```
candidates = candidates.withColumn("id",
functions.monotonicallyIncreasingId());
```

To apply the model to these candidates, we need to calculate features. For that, we just reuse the code we previously wrote. First, we calculate the node features:

```
Dataset<Row> nodeFeatures = nodeFeatures(sql, pageRank,
connectedComponents, degrees, candidates);
```

Here, the `nodeFeatures` method takes in the `pageRank`, `connectedComponents` and `degree` `DataFrame`s, which we computed on the train data, and calculates all the node-based features for the edges we pass in the `candidates` `DataFrame`.

Next, we compute the edge-based features for our candidates:

```
Dataset<Row> commonFriends = calculateCommonFriends(sql, edges,
candidates);
Dataset<Row> totalFriends = calculateTotalFriends(sql, edges, candidates);
Dataset<Row> jaccard = calculateJaccard(sql, edges, candidates);
```

We put the actual calculation of these features in utility methods, and just invoke them using a different set of edges.

Finally, we just join everything together:

```
Dataset<Row> features = candidates.join(commonFriends, "id")
        .join(totalFriends, "id")
        .join(jaccard, "id")
        .join(nodeFeatures, "id");
```

Now we are ready to apply the XGBoost model to these features. For that we will use the `mapPartition` function: it is similar to usual `map`, but instead of taking in just one item, it gets multiple at the same time. This way, we will build a `DMatrix` for multiple objects at the same time, and it will save time.

This is how we do it. First, we create a special class `ScoredEdge` for keeping the information about the nodes of the edge, the score it was assigned by the model as well as the actual label:

```
public class ScoredEdge implements Serializable {
    private long node1;
    private long node2;
    private double score;
    private double target;
    // constructor, getter and setters are omitted
}
```

Now we score the candidate edges:

```
JavaRDD<ScoredEdge> scoredRdd = features.toJavaRDD().mapPartitions(rows ->
{
    List<ScoredEdge> scoredEdges = new ArrayList<>();
    List<LabeledPoint> labeled = new ArrayList<>();

    while (rows.hasNext()) {
        Row r = rows.next();
```

```
            long node1 = r.getAs("node1");
            long node2 = r.getAs("node2");
            double target = r.getAs("target");
            scoredEdges.add(new ScoredEdge(node1, node2, target));

            float[] vec = rowToFloatArray(columns, r);
            labeled.add(LabeledPoint.fromDenseVector(0.0f, vec));
        }

        DMatrix data = new DMatrix(labeled.iterator(), null);
        float[][] xgbPred =
            xgb.predict(new ml.dmlc.xgboost4j.scala.DMatrix(data), false, 20);

        for (int i = 0; i < scoredEdges.size(); i++) {
            double pred = xgbPred[i][0];
            ScoredEdge edge = scoredEdges.get(i);
            edge.setScore(pred);
        }

        return scoredEdges.iterator();
    });
```

The code inside the `mapPartition` function does the following: first, we go over all input rows, and create a `ScoredEdge` class for each row. In addition to that, we also extract features from each row (in exactly the same way we did it previously). Then we put everything into `DMatrix` and use the XGBoost model to score each row of this matrix. Finally, we put the score to the `ScoredEdge` objects. As a result of this step, we get an `RDD`, where each candidate edge is scored by the model.

Next, for each user we would like to recommend 10 potential coauthors. To do it we group by `node1` from our `ScoredEdge` class, then sort by score within each group and keep only the first 10:

```
JavaPairRDD<Long, List<ScoredEdge>> topSuggestions = scoredRdd
        .keyBy(s -> s.getNode1())
        .groupByKey()
        .mapValues(es -> takeFirst10(es));
```

Here, `takeFirst10` may be implemented this way:

```
private static List<ScoredEdge> takeFirst10(Iterable<ScoredEdge> es) {
    Ordering<ScoredEdge> byScore =
            Ordering.natural().onResultOf(ScoredEdge::getScore).reverse();
    return byScore.leastOf(es, 10);
}
```

If you remember, `Ordering` here is a class from Google Guava which does the sorting by score and then takes the first 10 edges.

Finally, let's see how good the suggestions are. For that we go over all the suggestions and count how many of these links were actually formed during the test period. And then we take the mean across all groups:

```
double mp10 = topSuggestions.mapToDouble(es -> {
    List<ScoredEdge> es2 = es._2();
    double correct = es2.stream().filter(e -> e.getTarget() ==
1.0).count();
    return correct / es2.size();
}).mean();

System.out.println(mp10);
```

What it does is it calculates `Precision@10` (the faction of correctly classified edges among the first 10) for each group and then takes a mean of it.

When we run it, we see that the score is about 30%. This is not a very bad result: it means that 30% of recommended edges were actually formed.

Still, this score is far from ideal, and there are a lot of ways to improve it. In real-life Social Networks, there often is additional information that we can use for building a model. For example, in case of the coauthors graph, we could use affiliation, titles, and texts from abstracts of the papers, conferences, and journals where the papers were published, and many other things. If the social graph comes from a web Social Network such as Facebook, we could use geographical information, groups and communities, and, finally, likes. By including this information, we should be able to achieve far better performance.

Summary

In this chapter, we looked at ways to handle very large amounts of data and special tools for doing this such as Apache Hadoop MapReduce and Apache Spark. We saw how to use them to process Common Crawl - the copy of the Internet, and calculate some useful statistics from it. Finally, we created a Link Prediction model for recommending coauthors and trained an XGBoost model in a distributed way.

In the next chapter, we will look at how Data Science models can be deployed to production systems.

10
Deploying Data Science Models

So far we have covered a lot of data science models, we talked about many supervised and unsupervised learning methods, including deep learning and XGBoost, and discussed how we can apply these models to text and graph data.

In terms of the CRISP-DM methodology, we mostly covered the *modeling* part so far. But there are other important parts we have not yet discussed: *evaluation* and *deployment*. These steps are quite important in the application lifecycle, because the models we create should be useful for the business and bring value, and the only way to achieve that is integrate them into the application (the deployment part) and make sure they indeed are useful (the evaluation part).

In this last chapter of the book we will cover exactly these missing parts--we will see how we can deploy data science models so they can be used by other services of the application. In addition to that, we will also see how to perform an online evaluation of already deployed models.

In particular, we will cover the following:

- Microservices in Java with Spring Boot
- Model evaluation with A/B tests and multi-armed bandits

By the end of this chapter you will learn how to create simple web services with data science models and how to design them in a way that is easy to test.

Microservices

Java is a very common platform choice for running production code for many applications across many domains. When data scientists create a model for existing applications, Java is a natural choice, since it can be seamlessly integrated into the code. This case is straightforward, you create a separate package, implement your models there, and make sure other packages use it. Another possible option is packaging the code into a separate JAR file, and include it as a Maven dependency.

But there is a different architectural approach for combining multiple components of a large system--the microservices architecture. The main idea is that a system should be composed of small independent units with their own lifecycle--their development, testing, and deployment cycles are independent of all other components.

These microservices typically communicate via REST API, which is based on HTTP. It is based on four HTTP methods--`GET`, `POST`, `PUT` and `DELETE`. The first two are most commonly used:

- `GET`: Get some information from the service
- `POST`: Submit some information to the service

There are quite a few libraries that allow creating a web service with a REST API in Java, and one of them is Spring Boot, which is based on the Spring Framework. Next, we will look into how we can use it for serving data science models.

Spring Boot

Spring is a very old and powerful Java library. The Core Spring module implements the **Dependency Injection** (**DI**) pattern, which allows developing loosely coupled, testable, and reliable applications. Spring has other modules which are built around the core, and one of them is Spring MVC, which is a module for creating web applications.

In simple terms, the DI pattern says that you should put the application logic in so-called *services* and then *inject* these services into web modules.

Spring MVC, as we already mentioned, is used for developing web services. It runs on top of the Servlet API, which is the Java way of dealing with processing web requests and producing web responses. Servlet containers implement the Servlet API. Thus, to be able to use your application as a web service, you need to deploy it to a servlet container. The most popular ones are Apache Tomcat and Eclipse Jetty. However, the API is quite cumbersome to use. Spring MVC is built on top of the Servlet API but hides all its complexity.

Additionally, the Spring Boot library allows us to quickly start developing a Spring application without going into a lot of configuration details such as, setting up Apache Tomcat, the Spring application context and so on. It comes with a good set of pre-defined parameters which are expected to work fine, so we can just start using it and concentrate on the application logic rather than configuring the servlet container.

Now let us see how we can use Spring Boot and Spring MVC for serving machine learning models.

Search engine service

Let us finally come back to our running example--building a search engine. In `Chapter 7`, *Extreme Gradient Boosting*, we created a ranking model, which we can use for reordering search engine results so that the most relevant content gets higher positions.

In the previous chapter, `Chapter 9`, *Scaling Data Science*, we extracted a lot of text data from Common Crawl. What we can do now is to put it all together--use Apache Lucene to index the data from Common Crawl, and then search its content and get the best results with the XGBoost ranking model.

We already know how to use Hadoop MapReduce to extract text information from Common Crawl. However, if you remember, our ranking model needs more than just text--apart from just the body text, it needs to know the title and the headers. We can either modify existing MapReduce jobs to extract the parts we need, or process it without Hadoop and just index it with Lucene directly. Let us look at the second approach.

First, we will again use the `HtmlDocument` class, which has the following fields:

```
public class HtmlDocument implements Serializable {
    private final String url;
    private final String title;
    private final ArrayListMultimap<String, String> headers;
    private final String bodyText;
    // constructors and getters are omitted
}
```

Then we will also reuse the method for converting HTML to this `HtmlDocument` object, but will adapt it slightly so it can read the WARC record from Common Crawl:

```
private static HtmlDocument extractText(ArchiveRecord record) {
    String html = TextUtils.extractHtml(record);
    Document document = Jsoup.parse(html);
    String title = document.title();
    Element body = document.body();
```

```
        String bodyText = body.text();

        Elements headerElements = body.select("h1, h2, h3, h4, h5, h6");
        ArrayListMultimap<String, String> headers = ArrayListMultimap.create();
        for (Element htag : headerElements) {
            String tagName = htag.nodeName().toLowerCase();
            headers.put(tagName, htag.text());
        }

        return new HtmlDocument(url, title, headers, bodyText);
    }
```

Here `extractHtml` is a method from the previous chapter that extracts HTML content from a WARC record, and the rest is the same as used in `Chapter 6`, *Working with Text - Natural Language Processing and Information Retrieval*.

Next, we need to go over each record of a WARC archive and convert it to an object of the `HtmlDocument` class. Since the archives are large enough, we do not want to keep the content of all the `HtmlDocument` objects in memory all the time. Instead, we can do this lazily on the fly: read the next WARC record, convert it to `HtmlDocument`, an index with Lucene. and do it again for the next record.

To be able to do it lazily, we will use the `AbstractIterator` class from Google Guava:

```
    public static Iterator<HtmlDocument> iterator(File commonCrawlFile) {
        ArchiveReader archive = WARCReaderFactory.get(commonCrawlFile);
        Iterator<ArchiveRecord> records = archive.iterator();

        return new AbstractIterator<HtmlDocument>() {
            protected HtmlDocument computeNext() {
                while (records.hasNext()) {
                    ArchiveRecord record = records.next();
                    return extractText(record);
                }
                return endOfData();
            }
        };
    }
```

First, we open the WARC archive and pass it to the instance of our `AbstractIterator` class. Inside, while there are still records, we convert them using our `extractText` function. Once we are done with processing, we signal about it by invoking the `endOfData` method.

Now we can index all the WARC files with Lucene:

```
FSDirectory directory = FSDirectory.open("lucene-index");
WhitespaceAnalyzer analyzer = new WhitespaceAnalyzer();
IndexWriter writer = new IndexWriter(directory, new
IndexWriterConfig(analyzer));

for (File warc : warcFolder.listFiles()) {
    Iterator<HtmlDocument> iterator = CommonCrawlReader.iterator(warc);

    while (iterator.hasNext()) {
        HtmlDocument htmlDoc = iterator.next();
        Document doc = toLuceneDocument(htmlDoc);
        writer.addDocument(doc);
    }
}
```

In this code, first we create a filesystem Lucene index, and then go over all WARC files from the `warcFolder` directory. For each such file we get the iterator using the method we just wrote, and then index each record of this WARC file with Lucene. The `toLuceneDocument` method should already be familiar to us from the `Chapter 6`, *Working with Text - Natural Language Processing and Information Retrieval;* it converts `HtmlDocument` to a Lucene document, and contains the following code:

```
String url = htmlDoc.getUrl();
String title = htmlDoc.getTitle();
String bodyText = htmlDoc.getBodyText();
ArrayListMultimap<String, String> headers = htmlDoc.getHeaders();

String allHeaders = String.join(" ", headers.values());
String h1 = String.join(" ", headers.get("h1"));
String h2 = String.join(" ", headers.get("h2"));
String h3 = String.join(" ", headers.get("h3"));

Document doc = new Document();
doc.add(new Field("url", url, URL_FIELD));
doc.add(new Field("title", title, TEXT_FIELD));
doc.add(new Field("bodyText", bodyText, TEXT_FIELD));
doc.add(new Field("allHeaders", allHeaders, TEXT_FIELD));
doc.add(new Field("h1", h1, TEXT_FIELD));
doc.add(new Field("h2", h2, TEXT_FIELD));
doc.add(new Field("h3", h3, TEXT_FIELD));
```

You can refer to `Chapter 6`, *Working with Text - Natural Language Processing and Information Retrieval*or more details.

With this code, we can quite quickly index a part of Common Crawl. For our experiment, we took only 3 WARC archives from December 2016, which contain about 0.5 million documents.

Now, after we indexed the data, we need to get our ranker. Let us reuse the models we created in `Chapter 6`, *Working with Text - Natural Language Processing and Information Retrieval* and `Chapter 7`, *Extreme Gradient Boosting:* the feature extractor and the XGBoost model.

If you remember, the feature extractor performs the following: tokenizes the query and the body, the title, and the headers of each document; puts them all into the TF-IDF vector space and computes the similarity between the query and all text features. In addition to that, we also looked at the similarity in the LSA space (the space reduced with SVD) and also a similarity between the query and the title in the GloVe space. Please refer to `Chapter 6`, *Working with Text - Natural Language Processing and Information Retrieval*, for more details about it.

So let us use these classes to implement our ranker. But first, we need to have a proper abstraction for all ranking functions, and for that, we can create the `Ranker` interface:

```
public interface Ranker {
    SearchResults rank(List<QueryDocumentPair> inputList);
}
```



While creating the interface may seem redundant at this step, it will ensure the services we create are easily extensible and replaceable, and this is very important for being able for model evaluation.

Its only method rank takes in a list of `QueryDocumentPair` objects and produces a `SearchResults` object. We created the `QueryDocumentPair` class in `Chapter 6`, *Working with Text - Natural Language Processing and Information Retrieval*, and it contains the query along with the text features of the document:

```
public static class QueryDocumentPair {
    private final String query;
    private final String url;
    private final String title;
    private final String bodyText;
    private final String allHeaders;
    private final String h1;
    private final String h2;
    private final String h3;
    // constructor and getters are omitted
}
```

The `SearchResults` object just contains a reordered list of `SearchResult` objects:

```
public class SearchResults {
    private List<SearchResult> list;
    // constructor and getters are omitted
}
```

The `SearchResult` is another object that just holds the title and the URL of a page:

```
public class SearchResult {
    private String url;
    private String title;
    // constructor and getters are omitted
}
```

Now let us create an implementation of this interface and call it `XgbRanker`. First, we specify the constructor, which takes in the `FeatureExtractor` object and the path to the saved XGBoost model:

```
public XgbRanker(FeatureExtractor featureExtractor, String pathToModel) {
    this.featureExtractor = featureExtractor;
    this.booster = XGBoost.loadModel(pathToModel);
}
```

And the `rank` method is implemented the following way:

```
@Override
public SearchResults rank(List<QueryDocumentPair> inputList) {
    DataFrame<Double> featuresDf = featureExtractor.transform(inputList);
    double[][] matrix = featuresDf.toModelMatrix(0.0);

    double[] probs = XgbUtils.predict(booster, matrix);
    List<ScoredIndex> scored = ScoredIndex.wrap(probs);
    List<SearchResult> result = new ArrayList<>(inputList.size());
    for (ScoredIndex idx : scored) {
        QueryDocumentPair doc = inputList.get(idx.getIndex());
        result.add(new SearchResult(doc.getUrl(), doc.getTitle());
    }

    return new SearchResutls(result);
}
```

Here we just took the code we wrote in `Chapter 6`, *Working with Text - Natural Language Processing and Information Retrieval* and `Chapter 7`, *Extreme Gradient Boosting* and put it inside, feature extractor creates a `DataFrame` with features. Then we use the utility class `XgbUtils` for applying the XGBoost model to the data from the `DataFrame`, and finally, we use the score from the model for reordering the input list. At the end, it just converts the `QueryDocumentPair` objects into `SearchResult` objects and returns it.

To create an instance of this class, we can first load the feature extracted we *trained* and saved previously as well as the model:

```
FeatureExtractor fe = FeatureExtractor.load("project/feature-
extractor.bin");
Ranker ranker = new XgbRanker(fe, "project/xgb_model.bin");
```

Here the `load` method is just a wrapper around `SerializationUtils` from Commons Lang.

Now we have the ranker, and we can use it to create a search engine service. Inside, it should take in Lucene's `IndexSearcher` for the Common Crawl index, and our ranker.

When we have a ranker, let us create a search service. It should take in Lucene's `IndexSearcher` and our `Ranker`.

Then we create the `search` method with the user query; it parses the query, gets the top 100 documents from the Lucene index, and reorders them with the ranker:

```
public SearchResults search(String userQuery) {
    Query query = parser.parse(userQuery);
    TopDocs result = searcher.search(query, 100);
    List<QueryDocumentPair> data = wrapResultsToObject(userQuery, searcher,
result)
    return ranker.rank(data);
}
```

Here we again reuse a function from `Chapter 6`, *Working with Text - Natural Language Processing and Information Retrieval:* the `wrapResultsToObject` which converts the Lucene results to `QueryDocumentPair` objects:

```
private static List<QueryDocumentPair> wrapResultsToObject(String
userQuery,
              IndexSearcher searcher, TopDocs result) throws IOException {
    List<QueryDocumentPair> data = new ArrayList<>();

    for (ScoreDoc scored : result.scoreDocs) {
        int docId = scored.doc;
        Document doc = searcher.doc(docId);
```

```
            String url = doc.get("url");
            String title = doc.get("title");
            String bodyText = doc.get("bodyText");
            String allHeaders = doc.get("allHeaders");
            String h1 = doc.get("h1");
            String h2 = doc.get("h2");
            String h3 = doc.get("h3");

            data.add(new QueryDocumentPair(userQuery, url, title,
                  bodyText, allHeaders, h1, h2, h3));
        }

        return data;
    }
```

Our search engine service is ready, so we can finally put it into a microservice. As discussed previously, a simple way to do it is via Spring Boot.

For that, the first step is including Spring Boot into our project. It is a bit unusual: instead of just specifying a dependency, we use the following snippet, which you need to put after your dependency section:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-dependencies</artifactId>
      <version>1.3.0.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

And then the following dependency in the usual place:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Note that the version part is missing here: Maven takes it from the dependency management section we just added. Our web service will respond with JSON objects, so we also need to add a JSON library. We will use Jackson, because Spring Boot already provides a built-in JSON handler that works with Jackson. Let us include it to our `pom.xml`:

```
<dependency>
  <artifactId>jackson-databind</artifactId>
  <groupId>com.fasterxml.jackson.core</groupId>
</dependency>
```

Now all the dependencies are added, so we can create a web service. In Spring terms, they are called `Controller` (or `RestController`). Let us create a `SearchController` class:

```
@RestController
@RequestMapping("/")
public class SearchController {
private final SearchEngineService service;

    @Autowired
    public SearchController(SearchEngineService service) {
        this.service = service;
    }

    @RequestMapping("q/{query}")
    public SearchResults contentOpt(@PathVariable("query") String query) {
        return service.search(query);
    }

}
```

Here we use a few of Spring's annotations:

- `@RestController` to tell Spring that this class is a REST controller
- `@Autowired` to tell Spring that it should inject the instance of the `SearchEngineService` into the controller
- `@RequestMapping("q/{query}")` to specify the URL for the service

Note that here we used the `@Autowired` annotation for injecting `SearchEngineService`. But Spring does not know how such a service should be instantiated, so we need to create a container where we do it ourselves. Let us do it:

```
@Configuration
public class Container {

    @Bean
    public XgbRanker xgbRanker() throws Exception {
```

```
        FeatureExtractor fe = load("project/feature-extractor.bin");
        return new XgbRanker(fe, "project/xgb_model.bin");
    }

    @Bean
    public SearchEngineService searchEngineService(XgbRanker ranker)
             throws IOException {
        File index = new File("project/lucene-rerank");
        FSDirectory directory = FSDirectory.open(index.toPath());
        DirectoryReader reader = DirectoryReader.open(directory);
        IndexSearcher searcher = new IndexSearcher(reader);
        return new SearchEngineService(searcher, ranker);
    }

    private static <E> E load(String filepath) throws IOException {
        Path path = Paths.get(filepath);
        try (InputStream is = Files.newInputStream(path)) {
            try (BufferedInputStream bis = new BufferedInputStream(is)) {
                return SerializationUtils.deserialize(bis);
            }
        }
    }
}
```

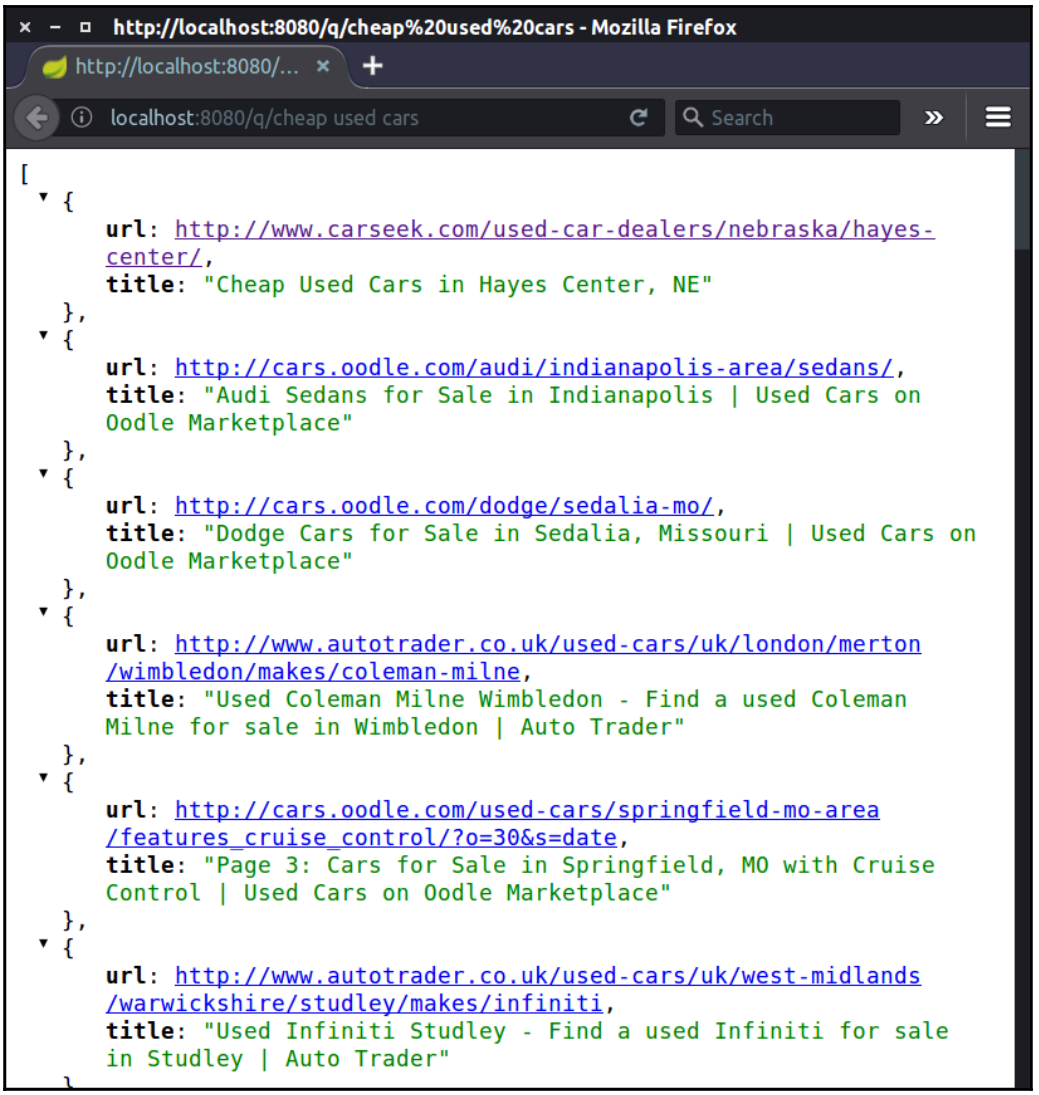
Here we first create an object of the `XgbRanker` class, and by using the `@Bean` annotation we tell Spring to put this class into the container. Next, we create the `SearchEngineService` which depends on `XgbRanker`, so the method where we initialize it takes it as a parameter. Spring treats this as a dependency and passes the `XgbRanker` object there so the dependency can be satisfied.

The final step is creating the application, which will listen to the `8080` port for incoming requests and respond with JSON:

```
@SpringBootApplication
public class SearchRestApp {
    public static void main(String[] args) {
        SpringApplication.run(SearchRestApp.class, args);
    }
}
```

Once we run this class, we can query our service by sending a `GET` request to `http://localhost:8080/q/query`, where `query` can be anything.

For example, if we want to find all the pages about *cheap used cars*, then we send a `GET` request to `http://localhost:8080/q/cheap%20used%20cars`. If we do this in a web browser, we should be able to see the JSON response:



As we see, it is possible to create a simple microservice serving data science models with a few easy steps. Next, we will see how the performance of our models can be evaluated online that is, after a model is deployed and users have started using it.

Online evaluation

When we do cross-validation, we perform offline evaluation of our model, we train the model on the past data, and then hold out some of it and use it only for testing. It is very important, but often not enough, to know if the model will perform well on actual users. This is why we need to constantly monitor the performance of our models online--when the users actually use it. It can happen that a model, which is very good during offline testing, does not actually perform very well during online evaluation. There could be many reasons for that--overfitting, poor cross-validation, using the test set too often for checking the performance, and so on.

Thus, when we come up with a new model, we cannot just assume it will be better because its offline performance is better, so we need to test it on real users.

For testing models online we usually need to come up with a sensible way of measuring performance. There are a lot of metrics we can capture, including simple ones such as the number of clicks, the time spent on the website and many others. These metrics are often called **Key Performance Indicators** (**KPIs**). Once we have decided which metrics to monitor, we can split all the users into two groups, and see where the metrics are better. This approach is called **A/B testing**, which is a popular approach to online model evaluation.

A/B testing

A/B testing is a way of performing a controlled experiment on users of your system. Typically, we have two systems--the original version of the system (the *control* system) and the new improved version (the *treatment* system).

A/B test is a way of performing controlled experiments on online users of the system. In these experiments, we have two systems--the original version (the *control*) and the new version (the *treatment*). To test whether the new version is better than the original one, we split the users of the system into two groups (*control* and *treatment*) and each group gets the output of its respective system. While the users interact with the system, we capture the KPI of our interest, and when the experiment is finished, we see if the KPI across the treatment group is significantly different from the control. If it is not (or it is worse), then the test suggests that the new version is not actually better than the existent one.

The comparison is typically performed using the ***t*-test**, we look at the mean of each group and perform a two-sided (or, sometimes, one-sided) test, which tells us whether the mean of one group is significantly better than the other one, or the difference can be attributed only to random fluctuations in the data.

Suppose we already have a search engine that uses the Lucene ranking formula and does not perform any re-ordering. Then we come up with the XGBoost model and would like to see if it is better or not. For that, we have decided to measure the number of clicks users made



This KPI was chosen because it is quite simple to implement and serves as a good illustration. But it is not a very good KPI for evaluating search engines: for example, if one algorithm gets more clicks than other, it may mean that the users weren't able to find what they were looking for. So, in reality, you should choose other evaluation metrics. For a good overview of existent options, you can consult the paper *Online Evaluation for Information Retrieval* by K. Hoffman.

Let us implement it for our example. First, we create a special class `ABRanker`, which implements the `Ranker` interface. In the constructor it takes two rankers and the random `seed` (for reproducibility):

```
public ABRanker(Ranker aRanker, Ranker bRanker, long seed) {
    this.aRanker = aRanker;
    this.bRanker = bRanker;
    this.random = new Random(seed);
}
```

Next, we implement the `rank` method, which should be quite straightforward; we just randomly select whether to use `aRanker` or the `bRanker`:

```
public SearchResults rank(List<QueryDocumentPair> inputList) {
    if (random.nextBoolean()) {
        return aRanker.rank(inputList);
    } else {
        return bRanker.rank(inputList);
    }
}
```

Let us also modify the `SearchResults` class and include two extra fields there, the ID of the result as well as the ID of the algorithm that generated it:

```
public class SearchResults {
    private String uuid = UUID.randomUUID().toString();
    private String generatedBy = "na";
    private List<SearchResult> list;
}
```

We will need that for tracking purposes. Next, we modify `XGBRanker` so it sets the `generatedBy` field to `xgb`--this change is trivial, so we will omit it here. Additionally, we need to create an implementation of the Lucene ranker. It is also trivial-- all this implementation needs to do is returning the given list as is without reordering it, and setting the `generatedBy` field to `lucene`.

Next, we modify our container. We need to create two rankers, assign each of them a name (by using the `name` parameter of the `@Bean` annotation), and then finally create the `ABRanker`:

```
@Bean(name = "luceneRanker")
public DefaultRanker luceneRanker() throws Exception {
    return new DefaultRanker();
}

@Bean(name = "xgbRanker")
public XgbRanker xgbRanker() throws Exception {
    FeatureExtractor fe = load("project/feature-extractor.bin");
    return new XgbRanker(fe, "project/xgb_model.bin");
}

@Bean(name = "abRanker")
public ABRanker abRanker(@Qualifier("luceneRanker") DefaultRanker lucene,
        @Qualifier("xgbRanker") XgbRanker xgb) {
    return new ABRanker(lucene, xgb, 0L);
}

@Bean
public SearchEngineService searchEngineService(@Qualifier("abRanker")
Ranker ranker)
        throws IOException {
    // content of this method stays the same
}
```

When we create `ABRanker` and `SearchEngineService`, in the parameters we provide the `@Qualifier` - which is the name of the bean. Since we now have quite a few rankers, we need to be able to distinguish between them, so they need to have names.

Once we have done it, we can restart our web service. From now on, half of the requests will be handled by the Lucene default ranker with no reordering, and half--by the XGBoost ranker with reordering by our model's score.

The next step is getting the user's feedback and storing it. In our case the feedback is the clicks, so we can create the following HTTP endpoint in `SearchController` for capturing this information:

```
@RequestMapping("click/{algorithm}/{uuid}")
public void click(@PathVariable("algorithm") String algorithm,
        @PathVariable("uuid") String uuid) throws Exception {
    service.registerClick(algorithm, uuid);
}
```

This method will be invoked when we receive a `GET` request to the `click/{algorithm}/{uuid}` path, where both `{algorithm}` and `{uuid}` are placeholders. Inside this method, we forward the call to the `SearchEngineService` class.

Now let us re-organize our abstractions a bit and create another interface `FeedbackRanker`, which extends the `Ranker` interface and provides the `registerClick` method:

```
public interface FeedbackRanker extends Ranker {
    void registerClick(String algorithm, String uuid);
}
```

We can make `SearchEngineService` dependent on it instead of a simple `Ranker`, so we can collect the feedback. In addition to that, we can also forward the call to the actual ranker:

```
public class SearchEngineService {
    private final FeedbackRanker ranker;

    public SearchEngineService(IndexSearcher searcher, FeedbackRanker
ranker) {
        this.searcher = searcher;
        this.ranker = ranker;
    }

    public void registerClick(String algorithm, String uuid) {
        ranker.registerClick(algorithm, uuid);
    }

    // other fields and methods are omitted
}
```

Finally, we make our `ABRanker` implement this interface, and put the capturing logic in the `registerClick` method.

For example, we can make the following modifications:

```
public class ABRanker implements FeedbackRanker {
    private final List<String> aResults = new ArrayList<>();
    private final List<String> bResults = new ArrayList<>();
    private final Multiset<String> clicksCount =
ConcurrentHashMultiset.create();

    @Override
    public SearchResults rank(List<QueryDocumentPair> inputList)
                    throws Exception {
        if (random.nextBoolean()) {
            SearchResults results = aRanker.rank(inputList);
            aResults.add(results.getUuid());
            return results;
        } else {
            SearchResults results = bRanker.rank(inputList);
            bResults.add(results.getUuid());
            return results;
        }
    }

    @Override
    public void registerClick(String algorithm, String uuid) {
        clicksCount.add(uuid);
    }

    // constructor and other fields are omitted
}
```

Here we create two array lists, which we populate with `UUID`s of created results and one `Multiset` from Guava, which counts how many clicks each of the algorithms received. We use collections here only for illustration purposes, and in reality, you should write the results to a database or some log.

Finally, let us imagine that the system was running for a while and we were able to collect some feedback from the users. Now it's time to check if the new algorithm is better than the old one. This is done with the *t*-test, which we can take from Apache Commons Math.

The simplest way of implementing it is the following:

```
public void tTest() {
    double[] sampleA = aResults.stream().mapToDouble(u ->
clicksCount.count(u)).toArray();
    double[] sampleB = bResults.stream().mapToDouble(u ->
clicksCount.count(u)).toArray();

    TTest tTest = new TTest();
    double p = tTest.tTest(sampleA, sampleB);

    System.out.printf("P(sample means are same) = %.3f%n", p);
}
```

After executing it, this will report the ***p*-value** of the *t*-test, or, the probability of rejecting the null hypothesis that two samples have the same mean. If this number is very small, then the difference is significant, or, in other words, there is strong evidence that one algorithm is better than another.

With this simple idea, we can perform online evaluation of our machine learning algorithm and make sure that the offline improvements indeed led to online improvements. In the next section, we will talk about a similar idea, multi-armed bandits, which allow us to select the best performing algorithm at runtime.

Multi-armed bandits

A/B testing is a great tool for evaluating some ideas. But sometimes there is no better model, for one particular case sometimes one is better, and sometimes another is better. To select the one which is better at this particular moment we can use on-line learning.

We can formulate this problem as a Reinforcement Learning problem--we have the **agents** (our search engine and the rankers), they interact with the **environment** (the users of the search engine), and get some **reward** (clicks). Then our systems learn from the interaction by taking **actions** (selecting the ranker), observing the feedback and selecting the best strategy based on it.

If we try to formulate A/B tests in this framework, then the action of the A/B test is choosing the ranker at random, and the reward is clicks. But for A/B tests, when we set up the experiment, we wait till it finishes. In online learning settings, however, we do not need to wait till the end and can already select the best ranker based on the feedback we received so far.

This problem is called the **bandit problem** and the algorithm called multi-armed bandit helps us solve it--it can select the best model while performing the experiment. The main idea is to have two kinds of actions--exploration, where you try to take actions of unknown performance, and exploitation, where you use the best performing model.

The way it is implemented is following: we pre-define some probability *e* (epsilon), with which we choose between exploration and exploitation. With probability *e* we randomly select any available action, and with probability *1 - e* we exploit the empirically best action. For our problem, it means that if we have several rankers, we use the best one with probability *1 - e*, and with probability e we use a randomly selected ranker for re-ordering the results. During the runtime, we monitor the KPIs to know which ranker is currently the best one, and update the statistics as we get more feedback.

This idea has a small drawback, when we just start running the bandit, we do not have enough data to choose which algorithm is the best one. This can be solved with a series of warm-ups, for example, the first 1000 results may be obtained exclusively in the exploration mode. That is, for the first 1000 results we just choose the ranker at random. After that we should collect enough data, and then select between exploitation and exploration with probability *e* as discussed above.

So let us create a new class for this, which we will call `BanditRanker`, which will implement the `FeedbackRanker` interface we defined for our `ABRanker`.

The constructor will take a map of `Ranker` with names associated to each ranker, the `epsilon` parameter, and the random `seed`:

```
public BanditRanker(Map<String, Ranker> rankers, double epsilon, long seed)
{
    this.rankers = rankers;
    this.rankerNames = new ArrayList<>(rankers.keySet());
    this.epsilon = epsilon;
    this.random = new Random(seed);
}
```

Inside, we will also keep a list of ranker names for internal use.

Next, we implement the `rank` function:

```
@Override
public SearchResults rank(List<QueryDocumentPair> inputList) throws
Exception {
    if (count.getAndIncrement() < WARM_UP_ROUNDS) {
        return rankByRandomRanker(inputList);
    }
```

```
        double rnd = random.nextDouble();
        if (rnd > epsilon) {
            return rankByBestRanker(inputList);
        }

        return rankByRandomRanker(inputList);
    }
```

Here we always select the ranker at random at first, and then either explore (select the ranker at random via the `rankByRandomRanker` method) or exploit (select the best ranker via the `rankByBestRanker` method).

Now let us see how to implement these methods, First, the `rankByRandomRanker` method is implemented in the following way:

```
private SearchResults rankByRandomRanker(List<QueryDocumentPair> inputList)
{
    int idx = random.nextInt(rankerNames.size());
    String rankerName = rankerNames.get(idx);
    Ranker ranker = rankers.get(rankerName);
    SearchResults results = ranker.rank(inputList);
    explorationResults.add(results.getUuid().hashCode());
    return results;
}
```

This is pretty simple: we randomly select a name from the `rankerName` list, then get the ranker by the name and use it for re-arranging the results. Finally, we also have the `UUID` of the generated result to a `HashSet` (or, rather, its hash to save the RAM).

The `rankByBestRanker` method has the following implementation:

```
private SearchResults rankByBestRanker(List<QueryDocumentPair> inputList)
{
    String rankerName = bestRanker();
    Ranker ranker = rankers.get(rankerName);
    return ranker.rank(inputList);
}

private String bestRanker() {
    Comparator<Multiset.Entry<String>> cnp =
            (e1, e2) -> Integer.compare(e1.getCount(), e2.getCount());
    Multiset.Entry<String> entry =
counts.entrySet().stream().max(cnp).get();
    return entry.getElement();
}
```

Here we keep `Multiset<String>`, which stores the number of clicks each algorithm has received. Then we select the algorithm based on this number and use it for re-arranging the results.

Finally, this is how we can implement the `registerClick` function:

```
@Override
public void registerClick(String algorithm, String uuid) {
    if (explorationResults.contains(uuid.hashCode())) {
        counts.add(algorithm);
    }
}
```

Instead of just counting the number of clicks, we first filter out the clicks for results generated at the exploitation phase, so they do not skew the statistics.

With this, we implemented the simplest possible version of multi-armed bandits, and you can use this for selecting the best-deployed model. To include this to our working web service, we need to modify the `container` class, but the modification is trivial, so we omit it here.

Summary

In this book we have covered a lot of material, starting from data science libraries available in data, then exploring supervised and unsupervised learning models, and discussing text, images, and graphs. In this last chapter we spoke about a very important step: how these models can be deployed to production and evaluated on real users.

25
Bibliography

This course is a blend of text and quizzes, all packaged up keeping your journey in mind. It includes content from the following Packt products:

- *Java for Data Science*, *Richard M. Reese*, and *Jennifer L. Reese*
- *Mastering Java for Data Science*, *Alexey Grigorev*

Index

A

A/B testing 675, 680
activation function
  about 20, 204, 205
  references 235
  sigmoid 205
  step function 205
Alex Black
  URL 273
AltaVista 8
AOL Cyclops React
  about 385
  reference 386
Apache Common Collections
  reference 365
Apache Commons IO
  reference 366
Apache Commons Math API
  URL 27, 321
  using 321
Apache Commons Math
  reference 367
Apache Commons
  Apache Commons Lang 378
  common modules 380
  Commons Collections 380
  Commons IO 379
  reference 379, 381
Apache Flink
  reference 367
Apache Hadoop
  about 619, 622
  Common Crawl 620, 625, 628
  Hadoop MapReduce 619
  reference 367
  used, for performing map-reduce 315
  YARN 619
Apache HttpComponents
  reference 393
Apache Lucene
  about 515
  customizing 519, 520
  reference 368
  using 511, 513
Apache Mahout
  reference 367, 368
Apache ODF Toolkit
  URL 70
Apache OpenNLP
  reference 368
Apache POI
  about 35
  URL 71
Apache Spark DataFrames
  reference 366
Apache Spark
  about 630
  reference 367
  URL 180
Apache Tika
  reference 388
Apache UIMA
  URL 24
Apache's Hadoop
  used, for performing map-reduce 316
Aparapi
  about 310
  application, creating 325, 326, 327, 328
  URL 325
  using 325
  using, for matrix multiplication 328, 330
apriori algorithm
  about 197

confidence 197
conviction 197
leverage 197
lift 197
support 197
Artificial Neural Network (ANN) 19, 203
associated probability distribution 178
association rule learning
about 177, 196
URL 197
using 198
attribute selection 123
audio coding formats
URL 31
autoencoder
about 22, 248
building, in DL4J 246
axet/wget
URL 65

B

backpropagation algorithms 228
Bag of Words 502
bandit problem 681
bar charts
country, using as category 128, 130
creating 126, 127
decade, using as category 130, 132
bar graph
about 117, 121
visual analysis 122, 123
Basic Linear Algebra Subprograms (BLAS)
URL 320
Bayesian Belief Network (BBN) 193
Bayesian Information Criterion (BIC) 490
Bayesian networks
about 192
using 193
bias unit 206
big data 7
binarization 231
binary classification models 419
Bliki
URL 55
blurring 109
Body Mass Index (BMI) 163
bottom-up approach 481
Brown-UMBC Reinforcement Learning and Planning (BURPLAP)
about 201
URL 201
bubble charts
creating 145, 146, 147

C

C4.5 algorithm 180
Canova
URL 231
Cascading Neural Networks (CNNs) 227
cats, versus dogs
data augmentation 609, 612
data, reading 595, 598
deep learning, URL 595
DeepLearning4J, running on GPU 613, 617
model, creating 598, 602
performance, monitoring 602, 608
Central Processing Unit (CPU) 324
class 178
Classification and Regression Tree (CART) analysis 179
classification models
about 420, 430
Encog 431
Java Statistical Analysis Tool (JSAT) 422
LIBLINEAR 424
LIBSVM 424
Statistical Machine Intelligence and Learning Engine (Smile) 420
classification problem
binary 358
multi-class 358
classification
about 419
binary classification 419
binary classification models 419
evaluation 432
multi-class classification 419
classifying text
about 260
by labels 260

by similarity 263
Doc2Vec 260
Word2Vec 260
cluster analysis
about 359, 480
additional set of features, adding 492
as dimensionality reduction 492, 494
density-based 480
graph-based 480
hierarchical methods 480, 482, 483, 485
K-means 487
partitioning 480
using, in supervised learning 491
versus supervised learning 495
clustering 123, 359
CMU Sphinx
about 25
used, for converting speech to text 289
Comma Separated Values (CSV)
about 11
data, handling 68
overview 32
Common Crawl
about 620, 621, 622, 627
download link 620
compressed row storage (CRS) 479
concatenation synthesis 281
conditional probability tables (CPTs) 194
console-based application
architecture 338
enhancements 353
purpose, defining 338
scope, defining 338
continuous restricted Boltzmann machine 242
Convolutional Neural Networks (CNNs)
about 229, 249, 593
model, building 249
model, evaluating 252
using 588, 594
corpus 255
costs function 234
crawler4j web crawler
URL 45
using 45
Cross Industry Standard Process for Data Mining (CRISP-DM)
about 355, 361
business understanding 362
data preparation 362
data understanding 362
evaluation 363
feature engineering 363
model development 363
modeling 363
CUDA
URL 28
curve of dimensionality 475
customer segmentation 359

D

data accessing
about 386
Comma Separated Values (CSV) 386
databases, using 393
DataFrames 396
HTML, using 388
JSON, using 391
text data, reading 386
web, using 388
data acquisition techniques
about 39
API calls, using to access common social media sites 52
HttpUrlConnection class, using 39
web crawlers 41
web scraping 48
data acquisition
about 11
using, Twitter 342
data augmentation 609
data classification 122
data cleaning
about 66
importance 12
process 12
data formats
audio, overview in Java 38
CSV data, handling 68
CSV data, overview 32
databases, overview 33

handling 67
images, overview in Java 38
JSON, handling 74
JSON, overview 36
PDF files, handling 72
PDF files, overview 35
spreadsheets, handling 70
spreadsheets, overview 32
streaming data, overview 38
used, for data science applications 31
video, overview in Java 38
XML, overview 36
data imputation 92
data mining 356
data quality
about 66
accuracy 66
consistency 66
uniformity 66
validity 66
data science applications
data formats, using 31
data science libraries
about 365
categories 365
data mining 367
data processing libraries 365
machine learning 367
math and stats libraries 367
text processing 368
data science
about 356, 364
Java, using 10
libraries 365
machine learning 17, 357
neural network, using 19
problem solving approach 9
process models 361
statistical method, using 15
URL 8
used, for problem solving 8
uses 356
data validation
about 100
dates, validating 102
e-mail addresses, validating 104
names, validating 106
types, validating 101
zip codes, validating 106
data visualization 14
data wrangling 12, 66
Database Management Systems (DBMS)
overview 33
Datavec
URL 231
decision tree
about 178
libraries 179
testing 184
types 179
using, with book dataset 180
deep autoencoders
about 245
autoencoder, building in DL4J 246
network, building 248
network, configuring 246
network, learning 248
network, retrieving 248
network, saving 248
specialized autoencoders 248
URL 246
Deep Belief Networks (DBNs) 242
Deep Learning for Java (DL4J)
URL 230
Deep Learning Networks (DLNs) 229
deep learning
about 237
approaches 22
Deeplearning4j
architecture 230
autoencoder, building 246
CSV file, reading 231
data, acquiring 231
data, manipulating 231
GPUs, using 313
hyperparameters, using 233
loss functions 234
model, building 232
model, configuring 232
model, testing 236

model, training 235
network model, instantiating 235
neural network 578, 581, 585
reference 368
references 261, 263
URL 230, 314
demographic statistics
URL 68
dendrites 203
denoising autoencoder 248
density-based spatial clustering of applications with noise (DBSCAN) 359, 490
Dependency Injection (DI) 664
dermatology
URL 216
Digital Signal Processor (DSP) 324
dimensionality reduction
shortcomings 462
supervised dimensionality reduction 462
truncated SVD 470
unsupervised dimensionality 462
unsupervised dimensionality reduction 463
diphones 288
distortion 487
Doc2Vec 256, 260
Document Object Model (DOM) 37
donut charts
creating 143, 144
Dual Coordinate Descent (DCD) 424
dummy-coding 471
dying ReLU 584
dynamic neural networks
about 215
Learning Vector Quantization (LVQ) 220
multilayer perceptron networks 216
Self-Organizing Maps (SOM) 221

E

Eigenvalue Decomposition (EVD) 464
elbow method 489
Encog
reference 368
URL 230
ensemble methods
about 179
bagging decision trees 179
boosted trees 179
random forest classifier 179
rotation forest 179
epoch 217, 581
evaluation metrics, classification
about 432
accuracy 433
F1 434
k-fold cross-validation 442
precision 434
recall 434
result validation 439, 440
ROC 436, 439
ROC (AUC) 436, 439
testing 444
training 441, 444
validation 444
evaluation metrics, regression
Mean Absolute Error (MAE) 454
Mean Squared Error (MSE) 453
evaluation, cluster analysis
about 496
manual evaluation 496
supervised evaluation 497
unsupervised evaluation 500
Exploratory Data Analysis (EDA)
about 362, 403
Apache Commons Math 405, 407
in Java 403
joinery 408, 409
search engine dataset 404
Extensible Markup Language (XML)
overview 36
URL, for rules 36
extensions, to standard Java library
about 378
AOL Cyclops React 385
Apache Commons 378
Google Guava 381
Extract, Transform, and Load (ETL) 231
Extreme Gradient Boosting (XGBoost) 550, 574

F

F1 score 435
faces
  identifying 293
  OpenCV, used for detecting 294, 295
feature extraction algorithms, dimensionality reduction
  Locally Linear Embedding (LLE) 463
  Non-Negative Matrix Factorization (NNMF) 463
  Principal Component Analysis (PCA) 463
  random projection 463, 476, 478, 479
  Singular Value Decomposition (SVD) 463
  t-SNE 463
  truncated SVD 467, 468, 469
  truncated SVD, for categorical and sparse data 470, 474, 475
feature selection 123
features, extracting from graph
  node feature 639
feedforward neural network (FNN) 215
Festival
  URL 25, 280
FFmpeg
  URL 289
Field-Programmable Gate Array (FPGA) 324
file type data
  URL 31
Flickr Developer Guide
  URL 58
Flickr4Java
  URL 59
Flickr
  handling 58
  URL 58
flow
  reference 398
fluent programing 331
formant synthesis 281
FreeTTS
  about 25
  URL 280, 281, 282
  using 283, 284
FScore 563

G

GATE
  reference 368
Generalized Least Squared (GLS) regression 172
geometric plane 18
Google account
  URL 62
Google Developer Console
  URL 62
Google Guava
  about 381, 382, 384
  reference 365
Gradient Boosted Classification Trees 550
Gradient Boosted Regression Trees 550
Gradient Boosted Trees 550
Gradient Boosting Machines (GBM) 549
gradient descent 22, 229
GRAL
  URL 15
Graphics Processing Unit (GPU)
  about 324, 613
  using, with Deeplearning4j 313
GRAphing Library (GRAL)
  about 115
  URL 115
ground truth 234

H

H2O
  reference 368
Han Solo
  URL, for aliases 342
hardware performance project 454, 456, 458, 459
hashtag 340
HBC application
  reference link 53
Hidden Layer 20, 206, 243
histograms
  about 121
  creating 140, 141, 142, 143
hold out technique 439
Holographic Associative Memory (HAM) 227
HttpUrlConnection class
  using 39

Hyperbolic Tangent kernel 187
hyperparameters 230
hypothesis testing 162

I

images
  brightening 111
  cleaning 107
  contrast, modifying 108
  converting, in different formats 113
  resizing 112
  smoothing 109
index charts
  about 119
  creating 123, 125, 126
indexing 511
information retrieval (IR)
  about 502
  ranking 540, 542, 544
  reranking, with Apache Lucene 545, 548
Input Layer 206, 243
Instantaneously Trained Neural Networks (ITNNs) 226
Interactive Exploratory Analysis Data
  about 410
  joinery shell 412, 417
  JVM languages 410
issues, text classification
  about 538
  language detection 537
  sentiment analysis 537
  spam detection 537
iteration 581

J

Jackson Project
  URL 12, 75
Java 8 streams
  about 331
  lambda expression 331
  used, for performing map-reduce 333, 335
  used, for performing matrix multiplication 332
  using 330
Java 9
  download link 411
Java API for XML Processing (JAXP) 37
Java bindings for CUDA (JCuda)
  URL 311
Java bindings for OpenCL (JOCL)
  URL 324
Java Database Connectivity (JDBC) 366, 393
Java HTML parser
  reference link 51
Java Native Interface (JNI) 550
Java OpenCL
  URL 324
Java Speech API (JSAPI)
  URL 25, 280
Java Statistical Analysis Tool (JSAT)
  reference 422
Java tokenizers
  about 83
  libraries 83
  third-party tokenizers 83
  used, for extracting words 82
Java Virtual Machine (JVM) 410
Java
  data frame libraries 366
  data science 364
  Exploratory Data Analysis (EDA) 403
  used, for data science 10
JavaCL
  URL 324
JavaML
  reference 368
JavaScript Object Notation (JSON)
  about 11
  input, processing 348
  overview 36
  URL 11, 36
JBayes
  URL 193
jblas API
  URL 320
  using 320, 321
JBlas
  reference 367
JBoost
  URL 180
JDBC API

URL 35
joinery
reference 366, 398, 412
JSAT
reference 368
JSON
handling 74
reference 392
streaming API, using 74
tree API, using 79
jsoup
URL 42, 49, 51
JVM languages
about 410
interactive Java 411

K

k-fold cross-validation 442
K-means
DBSCAN 490
implementing 487
K, selecting 488
k-Nearest Neighbor (k-NN) algorithm 203, 226
Kaggle
customer complaints, URL 470
kernel trick 187
kernels
linear 187
polynomial 187
Radial Basis Function (RBF) 187
sigmoid 187
URL 187
Key Performance Indicators (KPIs) 675

L

Latent Semantic Indexing (LSI) 521
Launch Instance button 615
Learning Vector Quantization (LVQ) 220
LIBLINEAR
reference 424, 428
LIBSVM
reference 424
Lightweight Java Game Library (LWJGL)
URL 324
line charts 119
Linear Algebra Package (LAPACK)
URL 320
linear regression
references 163
LingPipe
about 14
URL 24, 350, 368
Link Prediction
about 633
DBLP graph, reading 634, 636
edge feature 650, 652
edge features 647
features, extracting from graph 636, 637
link suggestions 658, 659, 661
MLlib, using 652, 654
negative sampling 642, 643, 646
node feature 638, 642
XGBoost, using 653, 655
linkages
about 481
average linkages (UPGMA linkage) 481
complete linkages 481
single linkages 481
linking 481
local minimums 209
Long Short-Term Memory (LSTM) 253
loss function 234

M

MAchine Learning for LanguagE Toolkit (MALLET)
URL 180
machine learning libraries, for regression
JSAT 452
LIBSVM 453
Smile 451
machine learning
Bayesian networks 18
decision trees 18
for texts 521
support vector machines 18
with data science 17
MALLET
reference 368
map-reduce
about 7

Hadoop job, creating 318, 319, 320
Hadoop job, executing 318, 319, 320
map method, implementing 316, 317
performing, with Apache's Hadoop 315, 316
performing, with Java 8 333, 335
reduce method, writing 317
reference link 315
using 315
market basket analysis 196
massaging 66
mathematical libraries
about 320
Apache Commons math API, using 321
jblas API, using 320, 321
ND4J API, using 322
references 320
Matrix Java Toolkit (MTJ) 474, 509
matrix operations
GPUs, using with DeepLearning4j 313
implementing 311
Maven Central repository
reference 366
MBROLA Project's
references 284
Mean Absolute Error (MAE) 454
Mean Average Precision (MAP) 545, 657
Mean Squared Error (MSE) 164, 453
mean
calculating 150
calculating, with Apache Commons 152
calculating, with Google Guava 152
calculating, with Java 8 techniques 151
calculating, with Java techniques 150
working, with 150
median
about 153
calculating 153
calculating, with Apache Commons 155
calculating, with simple Java techniques 153
working, with 150
MediaWiki API
URL 55
membrane potential 227
microservices
about 664
search engine service 665
Spring Boot, using 664
Mnist
URL 246
mode
about 155
calculating 155
calculating, with Apache Commons 159
calculating, with ArrayLists 157
calculating, with HashMap 158
working, with 150
Multi Layer Perceptron (MLP) network
about 20, 203, 216, 296
model, building 216
model, evaluating 218
model, retrieving 220
model, saving 220
values, predicting 219
multi-armed bandits 680
multimap 383
Multiple Additive Regression Trees
(LambdaMART) 569
multiple regression
references 169
using 168
multisets 382
munging 12, 66
MySQL Workbench 33

N

N-dimensional arrays for Java (ND4J) 575, 578
N-Dimensional Arrays for Java (ND4J)
URL 230
named entity recognition (NER)
implementing 256
location entities, identifying 258
OpenNLP, using 257
Natural Language Processing (NLP) tools
lemmatization 515
Named Entity Recognition (NER) 515
Part-of-Speech tagging (POS) Tagging 515
sentence splitting 515
Stanford CoreNLP 516, 519
tokenization 515
Natural Language Processing (NLP)

about 23, 255, 360, 502
name entity recognition (NER) 255
parts of speech (POS) 255
relationships 255
stop words 255
tokenization 255
tools 515
ND4J API
URL 322
using 322
neural network
about 19, 575
architectures 206
in DeepLearning4J 578, 581
reference link 203, 207
training 205
using, in data science 19
neuron 203
Neuroph Studio
creating, for classifying visual images 297
references 296
NLP APIs
DL4J 256
LingPipe 256
OpenNLP 256
Standford 256
UIMA 256
nonlinear java support
references 164
normalization 231
Normalized Discounted Cumulative Gain (NDCG) 545, 657

O

OAuth 2.0 provider
URL 52
objective 234
Obtain, Scrub, Explore, Model, and iNterpret (OSEMN) 361
online evaluation
A/B testing 675
about 675
Open Computing Language (OpenCL)
URL 324
using 324
Open Document Format (ODF)
URL 70
OpenCSV
URL 69
OpenCV 3.1
URL 293
OpenCV Javadoc
URL 108
OpenCV
URL 25, 108, 280, 294
using, to detect faces 294, 295
OpenNLP
URL 24, 257, 266
used, for extracting relationships 270
used, for performing NER 257
OpenRefine
URL 67
Optical Character Recognition (OCR)
about 107, 249, 292
Tess4j, using to extract text 292, 293
URL 26, 292
Ordinary Least Squares (OLS) 451, 462
outliers 116
Output Layer 206

P

package manager
reference link 225
page prediction 446, 447, 448, 450
parallel techniques
used, for improving application 27
parent nodes 193
patterns 208
PDFBox
about 35
URL 72
Penn TreeBankPOS tags
URL 268
perceptrons 204
phones 288
pie charts
about 118
creating 135, 137
Platform for Implementing Q-Learning Experiments (Piqle)

about 201
URL 201
plots 116
Pointwise Mutual Information (PMI) 528
population data
URL 123, 127
Portable Document Format (PDF)
files, handling 72
overview 35
POS tagging
about 266, 268
OpenNLP, using for identification 266
predictive analytics 356
Principal Component Analysis (PCA) 179, 360, 463, 466
problem solution categories, with machine learning
Natural Language Processing (NLP) 360
reinforcement learning 357
semi-supervised learning 357
supervised learning 357, 358
unsupervised learning 357, 359
process models, data science
about 361
Cross Industry Standard Process for Data Mining (CRISP-DM) 361
example 363
proximity matrix 482
Pulse Code Modulation (PCM) 289

Q

Q function 200

R

Radial Basis Function (RBF) 425
rank 313
ranking classifier 436
Read-Evaluate-Print Loop (REPL) 410
records 32
Recurrent Neural Networks (RNN) 229, 253
recursive partitioning 179
regression analysis
about 123, 163, 237
class, setting up 238
data, preparing 237, 239
data, reading 238
model, building 239
model, evaluating 240
multiple regression, using 168
simple linear regression, using 165
regression problem 358
regression tree 178
regression
about 450
evaluation 453
machine learning libraries 451
regularized Jaccard Similarity 650
reinforcement learning 200
relationship
extracting, from sentences 269
extracting, with OpenNLP 270
reshaping 12, 66
Residual Sum of Squares (RSS) 164
Resilient Distributed Dataset (RDD)
about 630
actions 630
transformations 630
Restricted Boltzmann Machines (RBM)
about 22, 242
configuring 244
reconstruction 243
tb 229
Rhino 412
Root Mean Squared Error (RMSE) 454

S

saddle points 209
sample size determination
about 162
reference link 162
sbt tool
reference 421
scatter charts
creating 138, 139, 140
scatter plot 122
screen scraping 11
Search engine service
multi-armed bandits 680, 683
search engine
building 398, 665
Self-Organizing Maps (SOM)

about 207
reference link 221
results, displaying 222
using 221
sentiment analysis
about 272
data, extracting 346
model, building 276
performing 351
reference link 273, 346
text, classifying 276
Word2Vec model, downloading 273
Word2Vec model, extracting 273
sequential minimal optimization algorithm
URL 187
Simple API for XML (SAX) 37
simple linear regression
using 165
Singular Value Decomposition (SVD) 459, 464, 522
social media sites
accessing, with API calls 52
OAuth, using to authenticate users 52
twitter, handling 52
wikipedia, handling 55
YouTube, handling 61
speech recognition
about 288
CMUPhinx, used for speech to text conversion 289
words 290
Spiking Neural Network (SNN) 227
spreadsheets
Excel spreadsheets, handling 71
handling 70
overview 32
Spring Batch
reference 367
Spring Boot 664
Stacked Denoising Autoencoder (SdA) 248
stacked graphs
creating 133, 134, 135
standard deviation 159
standard Java library
about 370, 371
collections 371, 372
extensions 378
input data, reading 372
input/output 372
output data, writing 374
streaming API 375
Standford NLP
URL 24
Stanford CoreNLP
reference 368
State-Action-Reward-State-Action (SARSA) 200
static neural networks
about 207
Java example 207
Statistical Machine Intelligence and Learning Engine (Smile)
reference 368, 420, 451, 452
stop words
removing 350
URL 351
Streaming API for XML (StAX) 37
streaming data
overview 38
Sum of Squared Errors (SSE) 164
Sum of Squared Residuals (SSR) 164
supervised dimensionality reduction
feature selection 462
supervised learning techniques
about 177
decision trees 178
Support Vector Machine (SVM) 185
supervised learning
about 358
for texts 536
Support Vector Machine (SVM) 358
about 18, 185
individual instances, testing 191
used, for camping data 188
Support Vector Regression (SVR) 358

T

Tab-Separated Values (TSV) files 32
Tablesaw
reference 366
tensor 575

Tess4j
  URL 292, 293
  used, for extracting text 292, 293
text categorization 359
text cleaning
  about 80, 85
  data imputation 92
  data validation 100
  data, subsetting 95
  data, transforming into usable form 85
  Java tokenizers, using to extract words 82
  stop words, removing 87
  text, replacing 90
  text, searching 90
  text, sorting 96
  words, searching 89
Text-To-Speech (TTS)
  about 280, 281
  FreeTTS, using 283
  voice information, gathering 287
  voices 285, 287
threshold 210
time series graphs 118
tokens 502
Tonal 108
top-down clustering 480
Top-Down Induction of Decision Trees (TDIDT) 179
tree 178
TweetHandler class
  about 344
  data, cleaning 349
  data, extracting for sentiment analysis model 346
  JSON input, processing 348
  results, analyzing 351
  sentiment analysis, performing 351
  sentiment model, building 347
  stop words, removing 350
tweets 52
Twitter4J
  URL 348
Twitter
  with data acquisition 342

U

UnitConcatenator 283
unstructured data 502
unsupervised dimensionality reduction
  feature extraction 462
unsupervised learning, for texts
  Latent Semantic Analysis (LSA) 521, 522, 523
  text clustering 525
  word embeddings 527, 529, 530, 534, 536
unsupervised learning
  about 359, 461
  clustering 359
  dimensionality reduction 359
  for texts 521
unsupervised machine learning
  about 196
  association rule learning 196
Utterance 283, 288

V

vanishing activation problem 608
vanishing gradient problem 584, 608
variables, not using in cluster analysis
  company response to customer 498
  consumer disputed 498
  timely response 498
vector quantization 221
Vector Space Model
  about 502, 503
  implementing 505, 507, 509, 511
vectorization 231
vectors 231
visual data
  classifying 296
  model, training 304, 305, 307
  Neuroph Studio, creating for classification 297

W

Waikato Environment for Knowledge Analysis (Weka)
  about 180
  classification algorithm, references 221
  URL 180
Watson Cloud 280

web crawler
  about 30, 41
  crawler4j, using 45
  custom web crawler, creating 42
  references 42
web scraping 30, 48
Weka
  reference 367
wikipedia
  handling 55
  references 47, 55
winutils.exe file
  reference 631
word embeddings
  about 535
  download link 534
Word2Vec 256, 260, 527
  URL 273

X

XGBFI
  reference 564
XGBoost4j 550
XGBoost
  features, importance 563
  for Learning to Rank 568, 570, 572
  for regression 565, 567
  installing 551
  parameter tuning 560
  parameter tuning, algorithm 561
  text features 562
  used, for classification 553, 557, 560
  using 553

Y

Youtube
  handling 61
  references 61
  searching, by keyword 62



Thank you for buying

Java: Data Science Made Easy

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.