
Md. Rezaul Karim
Md. Mahedi Kaysar
Large Scale Machine Learning with Spark
Discover everything you need to build robust machine learning applications with Spark 2.0
Packt>

Table of Contents

Large Scale Machine Learning with Spark
Credits
About the Authors
About the Reviewer
www.Packtpub.com
    Why subscribe?
Preface
    What this book covers
    What you need for this book
    Who this book is for
    Conventions
    Reader feedback
    Customer support
        Downloading the example code
        Errata
        Piracy
        Questions
1. Introduction to Data Analytics with Spark
    Spark overview
        Spark basics
        Beauties of Spark
    New computing paradigm with Spark
        Traditional distributed computing
        Moving code to the data
        RDD – a new computing paradigm
    Spark ecosystem
        Spark core engine
        Spark SQL
        DataFrames and datasets unification
        Spark streaming
        Graph computation – GraphX
        Machine learning and Spark ML pipelines
        Statistical computation – SparkR
    Spark machine learning libraries
        Machine learning with Spark
        Spark MLlib
            Data types
            Basic statistics
            Classification and regression
            Recommender system development
            Clustering
            Dimensionality reduction
            Feature extraction and transformation
            Frequent pattern mining

Spark ML
Installing and getting started with Spark
Packaging your application with dependencies
Running a sample machine learning application
Running a Spark application from the Spark shell
Running a Spark application on the local cluster
Running a Spark application on the EC2 cluster
References
Summary
2. Machine Learning Best Practices
What is machine learning?
Machine learning in modern literature
Machine learning and computer science
Machine learning in statistics and data analytics
Typical machine learning workflow
Machine learning tasks
Supervised learning
Unsupervised learning
Reinforcement learning
Recommender system
Semi-supervised learning
Practical machine learning problems
Machine learning classes
Classification and clustering
Rule extraction and regression
Most widely used machine learning problems
Large scale machine learning APIs in Spark
Spark machine learning libraries
Spark MLlib
Spark ML
Important notes for practitioners
Practical machine learning best practices
Best practice before developing an ML application
Good machine learning and data science worth huge
Best practice – feature engineering and algorithmic performance
Beware of overfitting and underfitting
Stay tuned and combining Spark MLlib with Spark ML
Making ML applications modular and simplifying pipeline synthesis
Thinking of an innovative ML system
Thinking and becoming smarter about Big Data complexities
Applying machine learning to dynamic data
Best practice after developing an ML application
How to enable real-time ML visualization
Do some error analysis
Keeping your ML application tuned
Keeping your ML application adaptive and scale-up
Choosing the right algorithm for your application
Considerations when choosing an algorithm

Accuracy
Training time
Linearity
Talking to your data when choosing an algorithm
Number of parameters
How large is your training set?
Number of features
Special notes on widely used ML algorithms
Logistic regression and linear regression
Recommendation systems
Decision trees
Random forests
Decision forests, decision jungles, and variants
Bayesian methods
Summary
3. Understanding the Problem by Understanding the Data
Analyzing and preparing your data
Data preparation process
Data selection
Data pre–processing
Data transformation
Resilient Distributed Dataset basics
Reading the Datasets
Reading from files
Reading from a text file
Reading multiple text files from a directory
Reading from existing collections
Pre–processing with RDD
Getting insight from the SMSSpamCollection dataset
Working with the key/value pair
mapToPair()
More about transformation
map and flatMap
groupByKey, reduceByKey, and aggregateByKey
sortByKey and sortBy
Dataset basics
Reading datasets to create the Dataset
Reading from the files
Reading from the Hive
Pre-processing with Dataset
More about Dataset manipulation
Running SQL queries on Dataset
Creating Dataset from the Java Bean
Dataset from string and typed class
Comparison between RDD, DataFrame and Dataset
Spark and data scientists workflow
Deeper into Spark
Shared variables

Broadcast variables
Accumulators
Summary
4. Extracting Knowledge through Feature Engineering
The state of the art of feature engineering
Feature extraction versus feature selection
Importance of feature engineering
Feature engineering and data exploration
Feature extraction – creating features out of data
Feature selection – filtering features from data
Importance of feature selection
Feature selection versus dimensionality reduction
Best practices in feature engineering
Understanding the data
Innovative way of feature extraction
Feature engineering with Spark
Machine learning pipeline – an overview
Pipeline – an example with Spark ML
Feature transformation, extraction, and selection
Transformation – RegexTokenizer
Transformation – StringIndexer
Transformation – StopWordsRemover
Extraction – TF
Extraction – IDF
Selection – ChiSqSelector
Advanced feature engineering
Feature construction
Feature learning
Iterative process of feature engineering
Deep learning
Summary
5. Supervised and Unsupervised Learning by Examples
Machine learning classes
Supervised learning
Supervised learning example
Supervised learning with Spark - an example
Air-flight delay analysis using Spark
Loading and parsing the Dataset
Feature extraction
Preparing the training and testing set
Training the model
Testing the model
Unsupervised learning
Unsupervised learning example
Unsupervised learning with Spark - an example
K-means clustering of the neighborhood
Recommender system
Collaborative filtering in Spark

- Advanced learning and generalizations
  - Generalizations of supervised learning
- Summary

6. Building Scalable Machine Learning Pipelines
- Spark machine learning pipeline APIs
  - Dataset abstraction
  - Pipeline
- Cancer-diagnosis pipeline with Spark
  - Breast-cancer-diagnosis pipeline with Spark
    - Background study
    - Dataset collection
    - Dataset description and preparation
    - Problem formalization
    - Developing a cancer-diagnosis pipeline with Spark ML
- Cancer-prognosis pipeline with Spark
  - Dataset exploration
  - Breast-cancer-prognosis pipeline with Spark ML/MLlib
- Market basket analysis with Spark Core
  - Background
  - Motivations
  - Exploring the dataset
  - Problem statements
  - Large-scale market basket analysis using Spark
  - The algorithm solution using Spark Core
  - Tuning and setting the correct parameters in SAMBA
- OCR pipeline with Spark
  - Exploring and preparing the data
  - OCR pipeline with Spark ML and Spark MLlib
- Topic modeling using Spark MLlib and ML
  - Topic modeling with Spark MLlib
  - Scalability
- Credit risk analysis pipeline with Spark
  - What is credit risk analysis? Why is it important?
  - Developing a credit risk analysis pipeline with Spark ML
    - The dataset exploration
  - Credit risk pipeline with Spark ML
    - Performance tuning and suggestions
- Scaling the ML pipelines
  - Size matters
  - Size versus skewness considerations
  - Cost and infrastructure
- Tips and performance considerations
- Summary

7. Tuning Machine Learning Models
- Details about machine learning model tuning
- Typical challenges in model tuning
- Evaluating machine learning models
  - Evaluating a regression model

Evaluating a binary classification model
Evaluating a multiclass classification model
Evaluating a clustering model
Validation and evaluation techniques
Parameter tuning for machine learning models
Hyperparameter tuning
Grid search parameter tuning
Random search parameter tuning
Cross-validation
Hypothesis testing
Hypothesis testing using ChiSqTestResult of Spark MLlib
Hypothesis testing using the Kolmogorov–Smirnov test from Spark MLlib
Streaming significance testing of Spark MLlib
Machine learning model selection
Model selection via the cross-validation technique
Cross-validation and Spark
Cross-validation using Spark ML for SPAM filtering a dataset
Model selection via training validation split
Linear regression–based model selection for an OCR dataset
Logistic regression-based model selection for the cancer dataset
Summary
8. Adapting Your Machine Learning Models
Adapting machine learning models
Technical overview
The generalization of ML models
Generalized linear regression
Generalized linear regression with Spark
Adapting through incremental algorithms
Incremental support vector machine
Adapting SVMs for new data with Spark
Incremental neural networks
Multilayer perceptron classification with Spark
Incremental Bayesian networks
Classification using Naive Bayes with Spark
Adapting through reusing ML models
Problem statements and objectives
Data exploration
Developing a heart diseases predictive model
Machine learning in dynamic environments
Online learning
Statistical learning model
Adversarial model
Summary
9. Advanced Machine Learning with Streaming and Graph Data
Developing real-time ML pipelines
Streaming data collection as unstructured text data

Labeling the data towards making the supervised machine learning
Creating and building the model
Real-time predictive analytics
Tuning the ML model for improvement and model evaluation
Model adaptability and deployment
Time series and social network analysis
Time series analysis
Social network analysis
Movie recommendation using Spark
Model-based movie recommendation using Spark MLlib
Data exploration
Movie recommendation using Spark MLlib
Developing a real-time ML pipeline from streaming
Real-time tweet data collection from Twitter
Tweet collection using TwitterUtils API of Spark
Topic modeling using Spark
ML pipeline on graph data and semi-supervised graph-based learning
Introduction to GraphX
Getting and parsing graph data using the GraphX API
Finding the connected components
Summary
10. Configuring and Working with External Libraries
Third-party ML libraries with Spark
Using external libraries with Spark Core
Time series analysis using the Cloudera Spark-TS package
Time series data
Configuring Spark-TS
TimeSeriesRDD
Configuring SparkR with RStudio
Configuring Hadoop run-time on Windows
Summary

Large Scale Machine Learning with Spark

Large Scale Machine Learning with Spark

Copyright © 2016 Packt Publishing

All rights reserved. No part of this book 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 book 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 book.

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.

First published: October 2016

Production reference: 1201016

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78588-874-8

www.packtpub.com

Credits

Authors Md. Rezaul Karim Md. Mahedi Kaysar	**Copy Editor** Safis Editing
Reviewer Muthukumar Subramanian	**Project Coordinator** Shweta H Birwatkar
Commissioning Editor Akram Hussain	**Proofreader** Safis Editing
Acquisition Editor Lester Frias	**Indexer** Aishwarya Gangawane
Content Development Editor Amrita Noronha	**Graphics** Disha Haria
Technical Editor Akash Patel	**Production Coordinator** Arvindkumar Gupta

About the Authors

Md. Rezaul Karim has more than 8 years of experience in the area of research and development with a solid knowledge of algorithms and data structures, focusing C, C++, Java, R, and Python and big data technologies such as Spark, Kafka, DC/OS, Docker, Mesos, Hadoop, and MapReduce.

He was first enchanted by machine learning while studying an Advanced Artificial Intelligence post-graduate course by applying the combined technique of Hadoop-based MapReduce and machine learning together for market basket analysis on large-scale business-oriented transactional databases in back 2010. Consequently, his research interests include machine learning, data mining, Semantic Web, big data, and bioinformatics. He has published more than 30 research papers in renowned peer-reviewed international journals and conferences focusing on the areas of data mining, machine learning, and bioinformatics, with good citations.

He is a Software Engineer and Researcher currently working at the Insight Centre for Data Analytics, Ireland (the largest data analytics center in Ireland and the largest Semantic Web research institute in the world) as a PhD Researcher. He is also a PhD candidate at the National University of Ireland, Galway. He also holds an ME (Master of Engineering) degree in Computer Engineering from the Kyung Hee University, Korea, majoring in data mining and knowledge discovery. And he has a BS (Bachelor of Science) degree in Computer Science from the University of Dhaka, Bangladesh.

Before joining the Insight Center for Data Analytics, he had been working as a Lead Software Engineer with Samsung Electronics, where he worked with the distributed Samsung R&D centers across the world, including Korea, India, Vietnam, Turkey, UAE, Brazil, and Bangladesh. Before that, he worked as a Graduate Research Assistant in the Database Lab at Kyung Hee University, Korea, while working towards his Master's degree. He also worked as an R&D Engineer with BMTech21 Worldwide, Korea. Even before that, he worked as a Software Engineer with i2SoftTechnology, Dhaka, Bangladesh.

*This book could not have been written without the support of my family and friends. In particular, my wife **Saroar** and my son **Shadman** deserve many thanks for their patience and encouragement throughout the past year.*

*I would like to give special thanks to **Md. Mahedi Kaysar** for co-authoring this book, and without his contributions, the writing would have been impossible. I would also like to thank **Jaynal Abedin** for his valuable suggestions towards different statistical and machine algorithms. Overall, I would like to dedicate this book to my respected teacher and research Guru **Prof. Dr. Chowdhury Farhan Ahmed** (Dept. of Computer Science &*

Engineering, University of Dhaka, Bangladesh) for his endless contributions to my life.

Further, I would like to thank the acquisition, content development and technical editors of Packt Publishing (and others who were involved to this book title) for their sincere cooperation and coordination. Additionally, without the work of numerous researchers who shared their expertise in publications, lectures, and source code, this book might not exist at all! Finally, I appreciate the efforts of the Apache Spark community and all those who have contributed to Spark APIs, whose work ultimately brought the machine learning to the masses.

Md. Mahedi Kaysar is a Software Engineer and Researcher at the Insight Center for Data Analytics (the largest data analytics center across the Ireland and the largest semantic web research institute in the world), Dublin City University (DCU), Ireland. Before joining the Insight Center at DCU, he worked as a Software Engineer at the Insight Center for Data Analytics, National University of Ireland, Galway and Samsung Electronics, Bangladesh.

He has more than 5 years of experience in research and development with a strong background in algorithms and data structures concentrating on C, Java, Scala, and Python. He has lots of experience in enterprise application development and big data analytics.

He obtained a BSc in Computer Science and Engineering from the Chittagong University of Engineering and Technology, Bangladesh. Now, he has started his postgraduate research in Distributed and Parallel Computing at the Dublin City University, Ireland.

His research interests include Distributed Computing, Semantic Web, Linked Data, big data, Internet of Everything, and machine learning. Moreover, he was involved in a research project in collaboration with CISCO Systems Inc. in the area of Internet of Everything and Semantic Web Technologies. His duties were to develop an IoT-enabled meeting management system, a scalable system for stream processing, designing, and showcasing the use cases of a project.

*This book could not have been written without the support of my family and friends. In particular, my beloved wife **Irin Akhtar** deserves many thanks for her patience and encouragement throughout the past year.*

*I would like to give special thanks to **Md. Rezaul Karim** for co-authoring this book and without his contributions, the writing would have been impossible. I would also like thank Packt Publishing and the group members who provides us a lot of support with the writing of this book, which helped to complete the book in time.*

About the Reviewer

Muthukumar Subramanian (https://www.linkedin.com/in/muthu4all) is an iSMAC (IoT, Social, Mobility, Analytics, Cloud) technologist, passionate about creating solutions that will yield better results. He is a new technology advisor and a startup incubator implementing innovative approaches, products, and solutions by embracing the latest technologies such as Spark Streaming, Spark Graph Processing, Lambda Architecture in big data Ecosystem, NoSQL, Apache Flink, and Amazon Web Services. He has vast training and consulting experience with Apache Spark and MLlib for various project implementation for many professionals and organisations from various domains and verticals.

www.Packtpub.com

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



https://www.packtpub.com/mapt

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Preface

Machine learning, at its core, is concerned with algorithms that transform raw data into information into actionable intelligence. This fact makes machine learning well suited to the predictive analytics of Big Data. Without machine learning, therefore, it would be nearly impossible to keep up with these massive streams of information altogether. Spark, which is relatively a new and emerging technology, provides big data engineers and data scientists a powerful response and a unified engine that is both faster and easy to use.

This allows learners from numerous areas to solve their machine learning problems interactively and at much greater scale. The book is designed to enable data scientists, engineers, and researchers to develop and deploy their machine learning applications at scale so that they can learn how to handle large data clusters in data intensive environments to build powerful machine learning models.

The contents of the books have been written in a bottom-up approach from Spark and ML basics, exploring data with feature engineering, building scalable ML pipelines, tuning and adapting them through for the new data and problem types, and finally, model building to deployment. To clarify more, we have provided the chapters outline in such a way that a new reader with a minimum of knowledge of machine learning and programming with Spark will be able to follow the examples and move towards some real-life machine learning problems and their solutions.

What this book covers

Chapter 1, *Introduction to Data Analytics with Spark*, this chapter covers Spark's overview, its computing paradigm, installation, and help us get started with Spark. It will briefly describe the main components of Spark and focus on its new computing advancements with the Resilient Distributed Datasets (RDD) and Dataset. It will then focus on the Spark’s ecosystem of machine learning libraries. Installing, configuring, and packaging a simple machine learning application with Spark and Maven will be demonstrated before scaling up on Amazon EC2.

Chapter 2, *Machine Learning Best Practices*, provides a conceptual introduction to statistical machine learning (ML) techniques aiming to take a newcomer from a minimal knowledge of machine learning all the way to being a knowledgeable practitioner in a few steps. The second part of the chapter is focused on providing some recommendations for choosing the right machine learning algorithms depending on its application types and

requirements. It will then go through some best practices when applying large-scale machine learning pipelines.

Chapter 3, *Understanding the Problem by Understanding the Data*, covers in detail the Dataset and Resilient Distributed Dataset (RDD) APIs for working with structured data, aiming to provide a basic understanding of machine learning problems with the available data. By the end, you will be able to deal with basic and complex data manipulation with ease. Some comparisons will be made available with basic abstractions in Spark using RDD and Dataset-based data manipulation to show gains both in terms of programming and performance. In addition, we will guide you on the right track so that you will be able to use Spark to persist an RDD or data object in memory, allowing it to be reused efficiently across the parallel operations in the later stage.

Chapter 4, *Extracting Knowledge through Feature Engineering*, explains that knowing the features that should be used to create a predictive model is not only vital but also a difficult question that may require deep knowledge of the problem domain to be examined. It is possible to automatically select those features in data that are most useful or most relevant for the problem someone is working on. Considering these questions, this chapter covers feature engineering in detail, explaining the reasons to apply it along with some best practices in feature engineering.

In addition to this, theoretical descriptions and examples of feature extraction, transformations, and selection applied to large-scale machine learning technique using both Spark MLlib and Spark ML APIs will be discussed.

Chapter 5, *Supervised and Unsupervised Learning by Examples*, will provide the practical knowledge surrounding how to apply supervised and unsupervised techniques on the available data to new problems quickly and powerfully through some widely used examples based on the previous chapters. These examples will be demonstrated from the Spark perspective.

Chapter 6, *Building Scalable Machine Learning Pipelines*, explains that the ultimate goal of machine learning is to make a machine that can automatically build models from data without requiring tedious and time-consuming human involvement and interaction. Therefore, this chapter guides the readers through creating some practical and widely used machine learning pipelines and applications using Spark MLlib and Spark ML. Both APIs will be described in detail, and a baseline use case will also be covered for both. Then we will focus towards scaling up the ML application so that it can cope up with increasing data loads.

Chapter 7, *Tuning Machine Learning Models*, shows that tuning an algorithm or machine learning application can be simply thought of as a

process by which one goes through and optimizes the parameters that impact the model in order to enable the algorithm to perform to its best. This chapter aims at guiding the reader through model tuning. It will cover the main techniques used to optimize an ML algorithm's performance. Techniques will be explained both from the MLlib and Spark ML perspective. We will also show how to improve the performance of the ML models by tuning several parameters, such as hyperparameters, grid search parameters with MLlib and Spark ML, hypothesis testing, Random search parameter tuning, and Cross-validation.

Chapter 8, *Adapting Your Machine Learning Models*, covers advanced machine learning techniques that will make algorithms adaptable to new data and problem types. It will mainly focus on batch/streaming architectures and on online learning algorithms using Spark streaming. The ultimate target is to bring dynamism to static machine learning models. Readers will also see how the machine learning algorithms learn incrementally from the data, that is, the models are updated each time the algorithm sees a new training instance.

Chapter 9, *Advanced Machine Learning with Streaming and Graph Data*, explains reader how to apply machine learning techniques, with the help of Spark MLlib and Spark ML, on streaming and graph data, for example, in topic modeling. The readers will be able to use available APIs to build real-time and predictive applications from streaming data sources such as Twitter. Through the Twitter data analysis, we will show how to perform large-scale social sentiment analysis. We will also show how to develop a large-scale movie recommendation system using Spark MLlib, which is an implicit part of social network analysis.

Chapter 10, *Configuring and Working with External Libraries*, guides the reader on using external libraries to expand their data analysis. Examples will be given for deploying third-party packages or libraries for machine learning applications with Spark core and ML/MLlib. We will also discuss how to compile and use external libraries with the core libraries of Spark for time series. As promised, we will also discuss how to configure SparkR to improve exploratory data manipulation and operations.

What you need for this book

Software requirements:

Following software is required for chapters 1-8 and 10: Spark 2.0.0 (or higher), Hadoop 2.7 (or higher), Java (JDK and JRE) 1.7+/1.8+, Scala 2.11.x (or higher), Python 2.6+/3.4+, R 3.1+, and RStudio 0.99.879 (or higher) installed. Eclipse Mars or Luna (latest) can be used. Moreover, Maven Eclipse plugin (2.9 or higher), Maven compiler plugin for Eclipse (2.3.2 or higher) and Maven assembly plugin for Eclipse (2.4.1 or higher) are required. Most importantly, re-use the provided `pom.xml` file with Packt's supplements and change the previously-mentioned version and APIs accordingly and everything will be sorted out.

For Chapter 9, *Advanced Machine Learning with Streaming and Graph Data,* almost all the software required, mentioned previously, except for the Twitter data collection example, which will be shown in Spark 1.6.1. Therefore, Spark 1.6.1 or 1.6.2 is required, along with the Maven-friendly `pom.xml` file.

Operating system requirements:

Spark can be run on a number of operating systems including Windows, Mac OS, and LINUX. However, Linux distributions are preferable (including Debian, Ubuntu, Fedora, RHEL, CentOS and so on). To be more specific, for example, for Ubuntu it is recommended to have a 14.04/15.04 (LTS) 64-bit complete installation or VMWare player 12 or Virtual Box. For Windows, Windows (XP/7/8/10) and for Mac OS X (10.4.7+) is recommended.

Hardware requirements:

To work with Spark smoothly, a machine with at least a core i3 or core i5 processor is recommended. However, to get the best results, core i7 would achieve faster data processing and scalability with at least 8 GB RAM (recommended) for a standalone mode and at least 32 GB RAM for a single VM, or higher for a cluster. Besides, enough storage to run heavy jobs (depending upon the data size you will be handling), and preferably at least 50 GB of free disk storage (for stand-alone and for SQL warehouse).

Who this book is for

Python and R are two popular languages for data scientists due to the large number of modules or packages that are readily available to help them solve their data analytics problems. However, traditional uses of these tools are often limiting, as they process data on either a single machine or with main memory-based approaches where the movement of data becomes time-consuming, the analysis requires sampling, and moving from development to production environments requires extensive re-engineering. To address these issues, Spark provides data engineers and data scientists a powerful and unified engine that is both faster and easy to use. This allows you to solve their machine learning problems interactively and at much greater scale.

Therefore, if you are an academic, researcher, data science engineer, or even a big data engineer working with large and complex data sets. Furthermore, if you want to board your data processing pipelines and machine learning applications to scale up more quickly, this book would be a suitable companion to this journey. Moreover, Spark provides many language choices, including Scala, Java, and Python. This facility will definitely help you to lift your machine learning applications on top of Spark and reshape using any one of these programming languages with Spark.

You should be familiar with the basics of machine learning concepts at least. Knowledge of open source tools and frameworks such as Apache Spark and Hadoop-based MapReduce would be good, but is not essential. A solid background in statistics and computational mathematics is expected. In addition, knowledge of Scala, Python, and Java is advisable. However, if you are experienced with intermediate programming languages, this will help you to understand the discussions and examples demonstrated in this book.

Conventions

In this book, you will find a number of styles of text 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: "We can include other contexts through the use of the `include` directive."

A block of code for creating the Spark session in a Windows environment is set as follows:

```
[default]
SparkSession spark = SparkSession
                  .builder()
                  .appName("JavaFPGrowthExample")
                  .master("local[*]")

 .config("spark.sql.warehouse.dir",
"E:/Exp/")                  .getOrCreate();
```

Or creating simple RDD from the input dataset is set as follows:

```
[default]
            String filename = "input/dataset.txt";
            RDD<String> data =
spark.sparkContext().textFile(fileName, 1);
```

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

```
$ scp -i /usr/local/key/my-key-pair.pem
 /usr/local/code/FPGrowth-0.0.1-SNAPSHOT-jar-with-
dependencies.jar
ec2-user@ec2-52-18-252-59.eu-west-
1.compute.amazonaws.com:/home/ec2-user/
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialogue boxes, for example, appear in the text like this: "Clicking the **Next** button moves you to the next screen".

> **Note**

Warnings or important notes appear in a box like this.

Tip

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 may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via 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 on https://www.packtpub.com/books/info/packt/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 all Packt books you have purchased 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/Large-Scale-Machine-Learning-with-Spark. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

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 would 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 on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from http://www.packtpub.com/support.

Piracy

Piracy of copyright 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

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

Chapter 1. Introduction to Data Analytics with Spark

This chapter covers an overview of Apache Spark, its computing paradigm, and installation to getting started. It will briefly describe the main components of Spark and focus on its new computing advancements. A description of the **Resilient Distributed Datasets** (**RDD**) and Dataset will be discussed as a base knowledge for the rest of this book. It will then focus on the Spark machine learning libraries. Installing and packaging a simple machine learning application with Spark and Maven will be demonstrated then before getting on board. In a nutshell, the following topics will be covered in this chapter:

- Spark overview
- New computing paradigm with Spark
- Spark ecosystem
- Spark machine learning libraries
- Installing and getting started with Spark
- Packaging your application with dependencies
- Running a simple machine learning application

Spark overview

This section describes Spark (https://spark.apache.org/) basics followed by the issues with the traditional parallel and distributed computing, then how Spark was evolved, and it then brings a new computing paradigm across the big data processing and analytics on top of that. In addition, we also presented some exciting features of Spark that easily attract the big data engineers, data scientists, and researchers, including:

- Simplicity of data processing and computation
- Speed of computation
- Scalability and throughput across large-scale datasets
- Sophistication across diverse data types
- Ease of cluster computing and deployment with different cluster managers
- Working capabilities and supports with various big data storage and sources
- Diverse APIs are written in widely used and emerging programming languages

Spark basics

Before praising Spark and its many virtues, a short overview is in the mandate. Apache Spark is a fast, in-memory, big data processing, and general-purpose cluster computing framework with a bunch of sophisticated APIs for advanced data analytics. Unlike the Hadoop-based MapReduce, which is only suited for batch jobs in speed and ease of use, Spark could be considered as a general execution engine that is suitable for applying advanced analytics on both static (batch) as well as real-time data:

- Spark was originally developed at the University of California, Berkeley's AMPLab based on **Resilient Distributed Datasets** (**RDDs**), which provides a fault-tolerant abstraction for in-memory cluster computing facilities. However, later on Spark's code base was bequeathed to the Apache Software Foundation making it open source, since then open source communities are taking care of it. Spark provides an interface to perform data analytics on entire clusters at scale with implicit data parallelism and fault-tolerance through its high-level APIs written in Java, Scala, Python, and R.

In Spark 2.0.0, elevated libraries (most widely used data analysis algorithms) are implemented, including:

- Spark SQL for querying and processing large-scale structured data
- SparkR for statistical computing that provides distributed computing using programming language R at scale
- MLlib for machine learning (ML) applications, which is internally divided into two parts; MLlib for RDD-based machine learning application development and Spark ML for a high-level abstraction to develop complete computational data science and machine learning workflows
- GraphX for large-scale graph data processing
- Spark Streaming for handling large-scale real-time streaming data to provide a dynamic working environment to static machine learning

Since its first stable release, Spark has already experienced dramatic and rapid development as well as wide adoptions through active initiatives from a wide range of IT solution providers, open source communities, and researchers around the world. Recently it has emerged as one of the most active, and the largest open source project in the area of big data processing and cluster computing, not only for its comprehensive adoptions, but also deployments and surveys by IT peoples, data scientists, and big data engineers worldwide. As quoted by *Matei Zaharia*, founder of Spark and the CTO of *Databricks* on the *Big Data analytics* news website at: http://bigdataanalyticsnews.com/apache-spark-3-real-world-use-cases/:

It's an interesting thing. There hasn't been as much noise about it commercially, but the actual developer community votes with its feet and people are actually getting things done and working with the project.

Even though many Tech Giants such as Yahoo, Baidu, Conviva, ClearStory, Hortonworks, Gartner, and Tencent are already using Spark in production - on the other hand, IBM, DataStax, Cloudera, and BlueData provide the commercialized Spark distribution for the enterprise. These companies have enthusiastically deployed Spark applications at a massive scale collectively for processing multiple petabytes of data on clusters of 8,000 nodes, which is the largest known cluster of Spark.

Beauties of Spark

Are you planning to develop a machine learning (ML) application? If so, you probably already have some data to perform preprocessing before you train a model on that data, and finally, you will be using the trained model to make predictions on new data to see the adaptability. That's all you need? We guess no, since you have to consider other parameters as well. Obviously, you will desire your ML models to be working perfectly in terms of accuracy, execution time, memory usage, throughput, tuning, and adaptability. Wait! Still not done yet; what happens if you would like to make your application handle large training and new datasets at scale? Or as a data scientist, what if you could build your ML models to overcome these issues as a multi-step journey from data incorporation through train and error to production by running the same machine learning code on the big cluster and the personal computer without breaking down further? You can simply rely on Spark and close your eyes.

Spark has several advantages over other big data technologies such as MapReduce (you can refer to https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html for MapReduce tutorials and the research paper *MapReduce: Simplified Data Processing on Large Clusters, Jeffrey Dean et al, In proc of OSDI, 2004* to get to know more) and Storm, which is a free and open source distributed real-time computation system (please refer to http://storm.apache.org/ for more on Storm-based distributed computing). First of all, Spark gives a comprehensive, unified engine to manage big data processing requirements with a variety of datasets such as text and tabular to graph data as well as the source of data (batch and real-time streaming data) that are diverse in nature. As a user (data science engineers, academicians, or developers), you can be likely benefited from Spark's rapid application development through simple and easy-to-understand APIs across batches, interactive, and real-time streaming applications.

Working and programming with Spark is easy and simple. Let us show you an example of that. Yahoo is one of the contributors and an early adopter of Spark, who implemented an ML algorithm with 120 lines of Scala code. With just 30 minutes of training on a large dataset with a hundred million records, the Scala ML algorithm was ready for business. Surprisingly, the same algorithm was written using C++ in 15,000 lines of code previously (please refer to the following URL for more at: https://www.datanami.com/2014/03/06/apache_spark_3_real-world_use_cases/). You can develop your applications using Java, Scala, R, or Python with a built-in set of over 100 high-level operators (mostly supported after Spark release 1.6.1) for transforming datasets and getting the familiarity with the data frame APIs for manipulating semi-structured, structured, and streaming data. In addition to the Map and Reduce operations, it supports SQL queries, streaming data, machine learning, and graph data processing. Moreover, Spark also provides an interactive shell written in Scala and Python for executing your codes sequentially (such as SQL or R style).

The main reason Spark adopts so quickly is because of two main factors: speed and sophistication. Spark provides order-of-magnitude performance for many applications using coarse-grained, immutable, and sophisticated data called Resilient Distributed Datasets that are spread across the cluster and that can be stored in memory or disks. An RDD offers fault-tolerance, which is resilient in a sense that it cannot be changed once created. Moreover, Spark's RDD has the property of recreating from its lineage if it is lost in the middle of computation. Furthermore, the RDD can be distributed automatically across the clusters by means of partitions and it holds your data. You can also keep it on your data on memory by the caching mechanism of Spark, and this mechanism enables big data applications in Hadoop-based MapReduce clusters to execute up to 100 times faster for in-memory if executed iteratively and even 10 times faster for disk-based operation.

Let's look at a surprising statistic about Spark and its computation powers. Recently, Spark took over Hadoop-based MapReduce by completing the 2014 Gray Sort Benchmark in the 100 TB category, which is an industry benchmark on how fast a system can sort 100 TB of data (1 trillion records) (please refer to http://spark.apache.org/news/spark-wins-daytona-gray-sort-100tb-benchmark.html and http://sortbenchmark.org/). Finally, it becomes the open source engine (please refer to the following URL for more information https://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html) for sorting at petabyte scale. In comparison, the previous world record set by Hadoop MapReduce had to use 2100 machines, taking 72 minutes of execution time, which implies Spark sorted the same data three times faster using 10 times fewer machines. Moreover, you can combine multiple libraries seamlessly to develop large-scale machine learning and data analytics pipelines to

execute the job on various cluster managers such as Hadoop YARN, Mesos, or in the cloud by accessing data storage and sources such as **HDFS**, **Cassandra**, **HBase**, **Amazon S3**, or even **RDBMs**. Moreover, the job can be executed as a standalone mode on a local PC or cluster, or even on **AWS EC2**. Therefore, deployment of a Spark application on the cluster is easy (we will show more on how to deploy a Spark application on the cluster later in this chapter).

The other beauties of Spark are: it is open source and platform independent. These two are also its greatest advantage, which is it's free to use, distribute, and modify and develop an application on any platform. An open source project is also more secure as the code is accessible to everyone and anyone can fix bugs as they are found. Consequently, Spark has evolved so rapidly that it has become the largest open source project concerning big data solutions with 750+ contributors from 200+ organizations.

New computing paradigm with Spark

In this section, we will show a chronology of Spark that will provide a concept of how it was evolved and emerged as a revolution for big data processing and cluster computing. In addition to this, we will also describe the Spark ecosystem in brief to understand the features and facilities of Spark in more details.

Traditional distributed computing

The traditional data processing paradigm is commonly referred to as a client-server model, which people used to move data to the code. The database server (or simply the server) was mainly responsible for performing data operations and then returning the results to the client-server (or simply the client) program. However, when the number of task to be computed is increased, a variety of operations and client devices also started to increase exponentially. As a result, a progressively complex array of computing endpoint in servers also started in the background. So to keep this type of computing model we need to increase the application (client) servers and database server in balance for storing and processing the increased number of operations. Consequently, the data propagation between nodes and data transfer back and forth across this network also increases drastically. Therefore, the network itself becomes a performance bottleneck. As a result, the performance (in terms of both the scalability and throughput) in this kind of computing paradigm also decreases undoubtedly. It is shown in the following diagram:

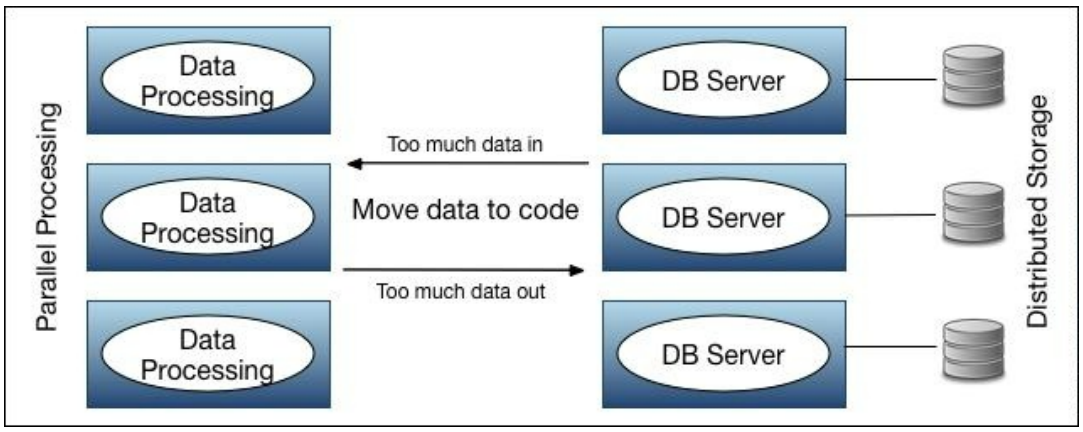


Figure 1: Traditional distributed processing in action.

After the successful completion of human genome projects in life sciences, real-time IOT data, sensor data, data from mobile devices, and data from the Web are creating the data-deluge and contributing for the big data, which has mostly evolved the data-intensive computing. The data-intensive

computing nowadays is now flattering increasingly in an emerging way, which requires an integrated infrastructure or computing paradigm, so that the computational resources and data could be brought in a common platform and perform the analytics on top of it. The reasons are diverse because big data is really huge in terms of complexity (**volume**, **variety**, and **velocity**), and from the operational perspective four ms (that is, **move**, **manage**, **merge**, and **munge**).

In addition, since we will be talking about large-scale machine learning applications in this book, we also need to consider some addition and critical assessing parameters such as validity, veracity, value, and visibility to grow the business. Visibility is important, because suppose you have a big dataset with a size of 1 PB; however, if there is no visibility, everything is a black hole. We will explain more on big data values in upcoming chapters.

It may not be feasible to store and process these large-scale and complex big datasets in a single system; therefore, they need to be partitioned and stored across multiple physical machines. Well, big datasets are partitioned or distributed, but to process and analyze these rigorously complex datasets, both the database servers as well as application servers might need to be increased to intensify the processing power at a large-scale. Again, the same performance bottleneck issues arrive at worst in multi-dimension that requires a new and more data-intensive big data processing and related computing paradigm.

Moving code to the data

To overcome the issues mentioned previously, a new computing paradigm is desperately needed so that instead of moving data to the code/application, we could move the code or application to the data and perform the data manipulation, processing, and associated computing at home (that is, where the data is stored). As you understand the motivation and objective, now the reverts programming model can be called **move code to data and do parallel processing on distributed system**, which can be visualized in the following diagram:

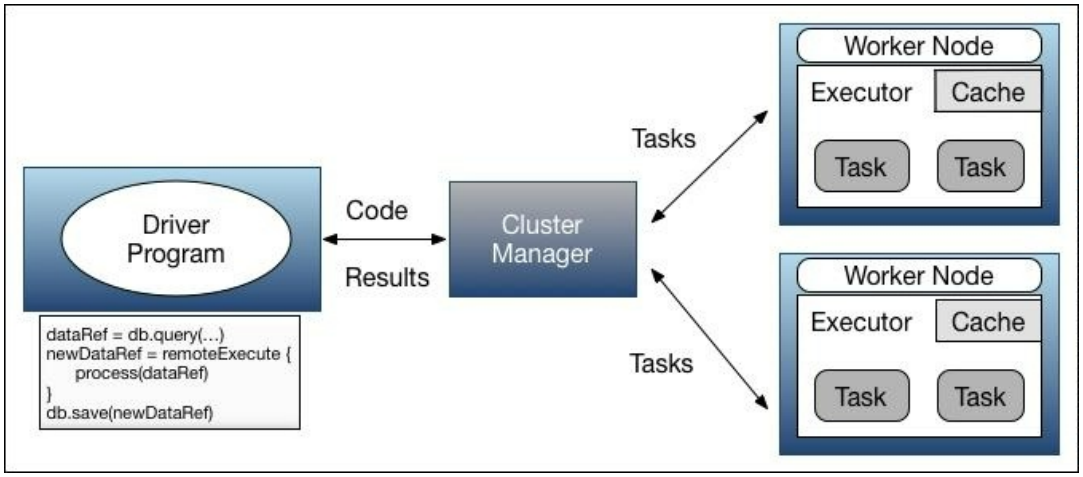


Figure 2: New computing (move code to data and do parallel processing on distributed system).

To understand the workflows illustrated in *Figure 2*, we can envisage a new programming model described as follows:

- Execution of a big data processing using your application initiated at your personal computer (let's name it **Driver Program**), which coordinates the execution in action remotely across multiple computing nodes within a cluster or grid, or a more openly speaking cloud.
- Now what you need is to transfer your developed application/algorithm/code segments (could be invoked or revoked using command-line or shell scripting as a simple programming language notation) to the computing/worker nodes (having large storage, main memory, and processing capability). We can simply imagine that the data to be computed or manipulated is already stored in those computing nodes as partitions or blocks.
- It is also understandable that the bulk data no longer needs to be transferred (upload/download) to your driver program because of the network or computing bottleneck, but it only holds the data reference in its variable instead, which is basically an address (hostname/IP address with a port) to locate the physical data stored in the computing nodes in a cluster, for example (of course bulk-upload could be performed using other solutions, such as scalable provisioning that is to be discussed in later chapters).
- So what do the remote computing nodes have? They have the data as well as code to perform the data computations and necessary processing to materialize the output or modified data without leaving their home (more technically, the computing nodes).
- Finally, upon your request, only the results could be transferred across the network to your driver program for validation or other analytics since there are many subsets of the original datasets.

It's worth noticing that by moving the code to the data, the computing structure has been changed drastically. Most interestingly, the volume of data transfer across the network has significantly reduced. The justification here is that you will be transferring only a small chunk of software code to the computing nodes and receiving a small subset of the original data as results in return. This was the most important paradigm shifting for big data processing that Spark brought to us with the concept of RDD, datasets, DataFrame, and other lucrative features that imply great revolution in the history of big data engineering and cluster computing. However, for brevity, in the next section we will only discuss the concepts of RDD and the other computing features will be discussed in upcoming sections

RDD – a new computing paradigm

To understand the new computing paradigm, we need to understand the concept of **Resilient Distributed Datasets** (**RDDs**), by which and how Spark has implemented the data reference concept. As a result, it has been able to shift the data processing easily to take it at scale. The basic thing about RDD is that it helps you to treat your input datasets almost like any other data objects. In other words, it brings the diversity of input data types, which you greatly missed in the Hadoop-based MapReduce framework.

An RDD provides the fault-tolerance capability in a resilient way in a sense that it cannot be changed once created and the Spark engine will try to iterate the operation once failed. It is distributed because once it has created performed partition operations, RDDs are automatically distributed across the clusters by means of partitions. RDDs let you play more with your input datasets since RDDs can also be transformed into other forms rapidly and robustly. In parallel, RDDs can also be dumped through an action and shared across your applications that are logically co-related or computationally homogeneous. This is achievable because it is a part of Spark's general-purpose execution engine to gain massive parallelism, so it can virtually be applied in any type of datasets.

However, to make the RDD and related operation on your inputs, Spark engines require you to make a distinguishing borderline between the data pointer (that is, the reference) and the input data itself. Basically, your driver program will not hold data, but only the reference of the data where the data is actually located on the remote computing nodes in a cluster.

To make the data processing faster and easier, Spark supports two types of operations, which can be performed on RDDs: transformations and actions (please refer to *Figure 3*). A transformation operation basically creates a new dataset from an existing one. An action, on the other hand, materializes a value to the driver program after a successful computation on input datasets on the remote server (computing nodes to be more exact).

The style of data execution initiated by the driver program builds up a graph as a **Directed Acyclic Graph** (**DAG**) style; where nodes represent RDDs and the transformation operations are represented by the edges. However, the execution itself does not start in the computing nodes in a Spark cluster until an action operation is performed. Nevertheless, before starting the operation, the driver program sends the execution graph (that signifies the style of operation for the data computation pipelining or workflows) and the code block (as a domain-specific script or file) to the cluster and each worker/computing node receives a copy from the cluster manager node:

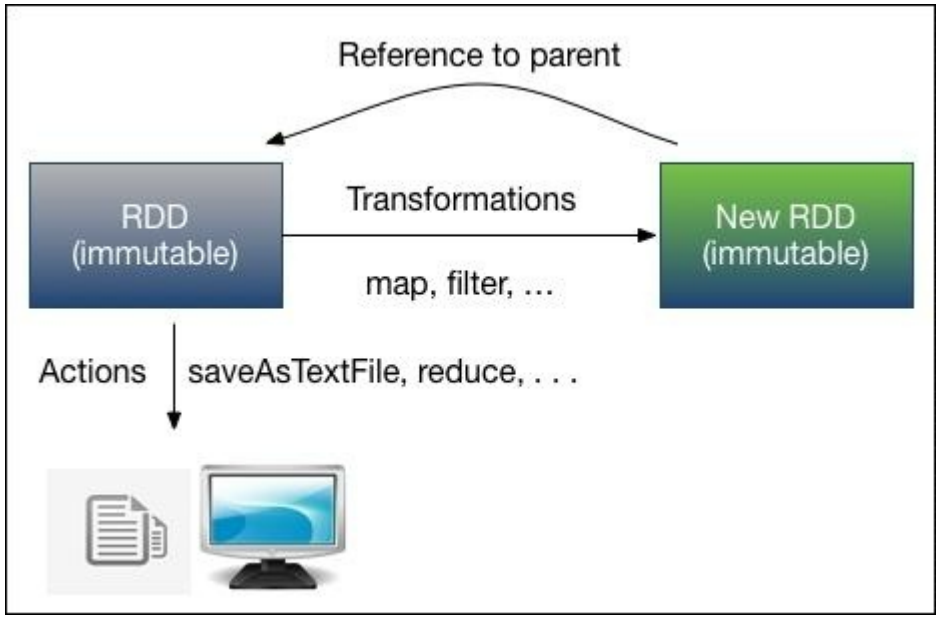


Figure 3: RDD in action (transformation and action operation).

Before proceeding to the next section, we argue you to learn about the action and transformation operation in more detail. Although we will discuss these two operations in Chapter 3, *Understanding the Problem by Understanding the Data* in detail. There are two types of transformation operations currently supported by Spark. The first one is the narrow transformation, where data mingle is unnecessary. Typical Spark narrow transformation operations are performed using the `filter()`, `sample()`, `map()`, `flatMap()`, `mapPartitions()` , and other methods. The wide transformation is essential to make a wider change to your input datasets so that the data could be brought in a common node out of multiple partitions of data shuffling. Wide transformation operations include `groupByKey()`, `reduceByKey()`, `union()`, `intersection()`, `join()`, and so on.

An action operation returns the final results of RDD computations from the transformation by triggering execution as a **Directed Acyclic Graph** (**DAG**) style to the Driver Program. But the materialized results are actually written on the storage, including the intermediate transformation results of the data objects and return the final results. Common actions include: `first()`,

`take()`, `reduce()`, `collect()`, `count()`, `saveAsTextFile()`, `saveAsSequenceFile()`, and so on. At this point we believe that you have gained the basic operation on top of RDDs, so we can now define an RDD and related programs in a natural way. A typical RDD programming model that Spark provides can be described as follows:

- From an environment variable, Spark Context (Spark shell or Python Pyspark provides you with a Spark Context or you can make your own, this will be described later in this chapter) creates an initial data reference RDD object.
- Transform the initial RDD to create more RDDs objects following the functional programming style (to be discussed later on).
- Send the code/algorithms/applications from the driver program to the cluster manager nodes. Then the cluster manager provides a copy to each computing node.
- Computing nodes hold a reference of the RDDs in its partition (again, the driver program also holds a data reference). However, computing nodes could have the input dataset provided by the cluster manager as well.
- After a transformation (via either narrow or wider transformation), the result to be generated is a brand new RDD, since the original one will not be mutated. Finally, the RDD object or more (specifically data reference) is materialized through an action to dump the RDD into the storage.
- The Driver Program can request the computing nodes for a chunk of results for the analysis or visualization of a program.

Wait! So far we have moved smoothly. We suppose you will ship your application code to the computing nodes in the cluster. Still you will have to upload or send the input datasets to the cluster to be distributed among the computing nodes. Even during the bulk-upload you will have to transfer the data across the network. We also argue that the size of the application code and results are negligible or trivial. Another obstacle is if you/we want Spark to process the data at scale computation, it might require data objects to be merged from multiple partitions first. That means we will need to shuffle data among the worker/computing nodes that are usually done by `partition()`, `intersection()`, and `join()` transformation operations.

So frankly speaking, the data transfer has not been eliminated completely. As we and you understand the overheads being contributed especially for the bulk upload/download of these operations, their corresponding outcomes are as follows:

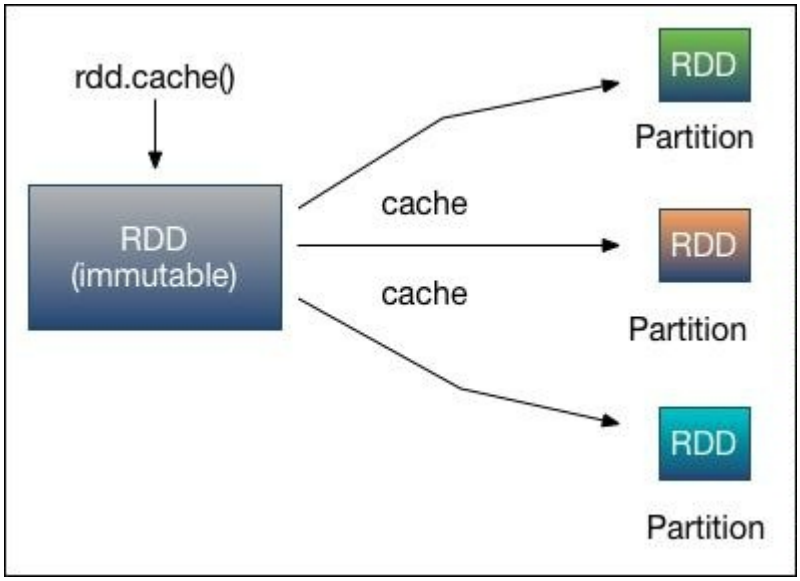


Figure 4: RDD in action (the caching mechanism).

Well, it's true that we have been compromised with these burdens. However, situations could be tackled or reduced significantly using the caching mechanism of Spark. Imagine you are going to perform an action multiple times on the same RDD objects, which have a long lineage; this will cause an increase in execution time as well as data movement inside a computing node. You can remove (or at least reduce) this redundancy with the caching mechanism of Spark (*Figure 4*) that stores the computed result of the RDD in the memory. This eliminates the recurrent computation every time. Because, when you cache on an RDD, its partitions are loaded into the main memory instead of a disk (however, if there is not enough space in the memory, the disk will be used instead) of the nodes that hold it. This technique enables big data applications on Spark clusters to outperform MapReduce significantly for each round of parallel processing. We will discuss more on Spark data manipulations and other techniques in Chapter 3, *Understanding the Problem by Understanding the Data* in detail.

Spark ecosystem

To provide more enhancements and additional big data processing capabilities, Spark can be configured and run on top of existing Hadoop-based clusters. As already stated, although Hadoop provides the **Hadoop Distributed File System** (**HDFS**) for efficient and operational storing of large-scale data cheaply; however, MapReduce provides the computation fully disk-based. Another limitation of MapReduce is that; only simple computations can be executed with a high-latency batch model, or static data to be more specific. The core APIs in Spark, on the other hand, are written in Java, Scala, Python, and R. Compared to MapReduce, with the more general and powerful programming model, Spark also provides several libraries that are part of the Spark ecosystems for redundant capabilities in big data analytics, processing, and machine learning areas. The Spark ecosystem consists of the following components, as shown in *Figure 5*:

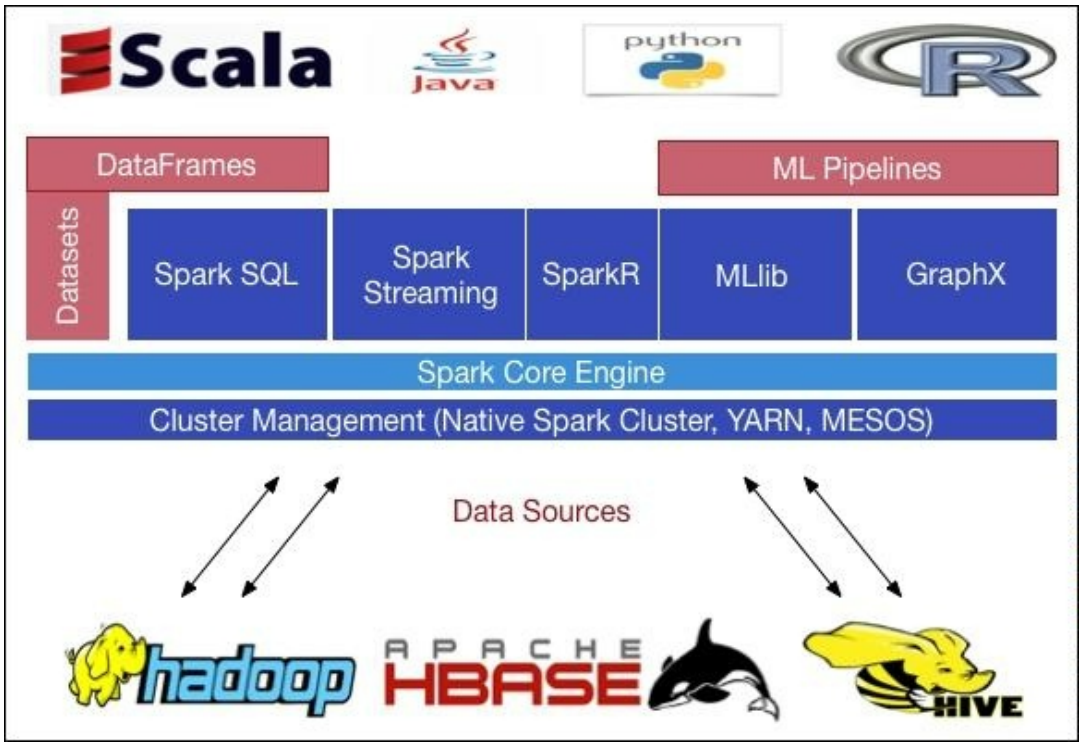


Figure 5: Spark ecosystem (till date up to Spark 1.6.1).

As we have already stated, it is very much possible to combine these APIs seamlessly to develop large-scale machine learning and data analytics applications. Moreover, the job can be executed on various cluster managers such as Hadoop YARN, Mesos, standalone, or in the cloud by accessing data storage and sources such as HDFS, Cassandra, HBase, Amazon S3, or even RDBMs.

Nevertheless, Spark is enriched with other features and APIs. For example, recently Cisco has announced to invest $150M in the Spark ecosystem towards Cisco Spark Hybrid Services (http://www.cisco.com/c/en/us/solutions/collaboration/cloud-collaboration/index.html). So Cisco Spark open APIs could boost its popularity with developers in higher cardinality (highly secure collaboration and connecting smartphone systems to the cloud). Beyond this, Spark has recently integrated Tachyon (http://ampcamp.berkeley.edu/5/exercises/tachyon.html), a distributed in-memory storage system that economically fits in memory to further improve Spark's performance.

Spark core engine

Spark itself is written in Scala, which is functional, as well as **Object Oriented Programming Language** (**OOPL**) which runs on top of JVM. Moreover, as mentioned in *Figure 5*, Spark's ecosystem is built on top of the general and core execution engine, which has some extensible API's implemented in different languages. The lower level layer or upper level layer also uses the Spark core engine as a general execution job performing engine and it provides all other functionality on top. The Spark Core is written in Scala as already mentioned, and it runs on **Java Virtual Machine** (**JVM**) and the high-level APIs (that is, Spark MLlib, SparkR, Spark SQL, Dataset, DataFrame, Spark Streaming, and GraphX) that use the core in the execution time.

Spark has brought the in-memory computing mode to a great visibility. This concept (in-memory computing) enables the Spark core engine to leverage speed through a generalized execution model to develop diverse applications.

The low-level implementation of general purpose data computing and machine learning algorithms written in Java, Scala, R, and Python are easy to use for big data application development. The Spark framework is built on Scala, so developing ML applications in Scala can provide access to the latest features that might not be available in other Spark languages initially. However, that is not a big problem, open source communities also take care of the necessity of developers around the globe. Therefore, if you do need a particular machine learning algorithm to be developed, and you want to add it to the Spark library, you can contribute it to the Spark community. The source code of Spark is openly available on GitHub at https://github.com/apache/spark as Apache Spark mirror. You can do a pull out request and the open source community will review your changes or algorithm before adding it to the master branch. For more information, please check the Spark Jira confluence site at https://cwiki.apache.org/confluence/display/SPARK/Contributing+to+Spark.

Python was a great arsenal for data scientists previously, and the contribution of Python in Spark is also not different. That means Python also has some excellent libraries for data analysis and processing; however, it is comparatively slower than Scala. R on the other hand, has a rich environment for data manipulation, data pre-processing, graphical analysis, machine learning, and statistical analysis, which can help to increase the developer's productivity. Java is definitely a good choice for developers who are coming from the Java and Hadoop background. However, Java also has the similar problem as Python, since Java is also slower than Scala.

A recent survey presented on the Databricks website at http://go.databricks.com/2015-spark-survey on Spark users (66% users evaluated the Spark languages where 41% were data engineers and 22% were data scientists) shows that 58% are using Python, 71% are using Scala, 31% are using Java, and 18% are using R for developing their Spark applications. However, in this book, we will try to provide the examples mostly in Java and a few in Scala if needed for the simplicity. The reason for this is that many of the readers are very familiar with Java-based MapReduce. Nevertheless, we will provide some hints of using the same examples in Python or R in the appendix at the end.

Spark SQL

Spark SQL is a Spark component for querying and structured data processing. The demand was obvious since many data science engineers and business intelligence analysts also rely on interactive SQL queries for exploring data from RDBMS. Previously, MS SQL server, Oracle, and DB2 were used frequently by the enterprise. However, these tools were not scalable or interactive. Therefore, to make it easier, Spark SQL provides a programming abstraction called DataFrames and datasets that work as distributed SQL query engines, which support unmodified Hadoop Hive queries to execute 100 times faster on existing deployments and data. Spark SQL is a powerful integration with the rest of the Spark ecosystem.

Recently, Spark has offered a new experimental interface, commonly referred to as datasets (to be discussed in more details in the next section), which provide the same benefits of RDDs to use the `lambda` functions strongly. Lambda evolves from the Lambda Calculus (http://en.wikipedia.org/wiki/Lambda_calculus) that refers to anonymous functions in computer programming. It is a flexible concept in modern programming language that allows you to write any function quickly without naming them. In addition, it also provides a nice way to write closures. For example, in Python:

```
def adder(x):
    return lambda y: x + y
```

```
add6 = adder(6)
add4(4)
```

It returns the result as `10`. On the other hand, in Java, it can be similarly written if an integer is odd or even:

```
Subject<Integer> sub = x -> x % 2 = 0; // Tests
if the parameter is even.
boolean result = sub.test(8);
```

The previous `lambda` function checks if the parameter is even and returns either `true` or `false`. For example, the preceding snippet would return `true` since `8` is divisible by `2`.

Please note that in Spark 2.0.0, the Spark SQL has substantially been improved with the SQL functionalities with SQL 2003 support. Therefore, Spark SQL can now be executed with all the 99 TPC-DS queries. More importantly, now a native SQL parser supports ANSI_SQL and Hive QL. Native DDL is a command that can also be executed, it also now supports sub querying of SQL and the view of canonicalization support.

DataFrames and datasets unification

In the latest Spark release 2.0.0, in Scala and Java, the DataFrame and dataset have been unified. In other words, the DataFrame is just a type alias for a dataset of rows. However, in Python and R, given the lack of type safety, DataFrame is the main programming interface. And for Java, DataFrame is no longer supported, but only the Dataset and RDD-based computations are supported, and DataFrame has become obsolete (note that it has become obsolete - not depreciated). Although SQLContext and HiveContext are kept for backward compatibility; however, in Spark 2.0.0 release, the new entry point that replaces the old SQLContext and HiveContext for DataFrame and dataset APIs is SparkSession.

Spark streaming

You might want your applications to have the ability to process and analyze not only static datasets, but also real-time streams data. To make your wish easier, Spark Streaming provides the facility to integrate your application with popular batch and streaming data sources. The most commonly used data sources include HDFS, Flume, Kafka, and Twitter, and they can be used through their public APIs. This integration allows users to develop powerful interactive and analytical applications on both streaming and historical data. In addition to this, the fault tolerance characteristics are achieved through Spark streaming.

Graph computation – GraphX

GraphX is a resilient distributed graph computation engine built on top of Spark. GraphX brought a revolution to the users who want to interactively build, transform, and reason graph structured data with millions of nodes and vertices at scale. As a developer you will enjoy the simplicity so that a large-scale graph (social network graph, normal network graph, or astrophysics) could be represented using a small chunk of code written in Scala, Java, or Python. GraphX enables the developers to take the advantages of both data-parallel and graph-parallel systems by efficiently expressing graph computation with ease and speed. Another beauty added in the cabinet of GraphX is that it can be used to build an end-to-end graph analytical pipeline on real-time streaming data, where the graph-space partitioning is used to handle large-scale directed multigraph with properties associated with each vertex and edge. Well, some fundamental graph operators are used to make this happen such as subgraph, joinVertices and aggregateMessages as well as an optimized variant of the Pregel API in particular.

Machine learning and Spark ML pipelines

Traditional machine learning applications were used to build using R or Matlab that lacks scalability issues. Spark has brought two emerging APIs, Spark MLlib and Spark ML. These APIs make the machine learning as an actionable insight for engineering big data to remove the scalability constraint. Built on top of Spark, MLlib is a scalable machine learning library that is enriched with numerous high-quality algorithms with a high-accuracy performance that mainly works for RDDs. Spark provides many language options for the developers that are functioning in Java, Scala, R, and Python to develop complete workflows. Spark ML, on the other hand, is an ALPHA component that enhances a new set of machine learning algorithms to let data scientists quickly assemble and configure practical machine learning pipelines on top of DataFrames.

Statistical computation – SparkR

SparkR is an R package specially designed for the data scientists who are familiar with R language and want to analyze large datasets and interactively run jobs from the R shell, which supports all the major Spark DataFrame operations such as aggregation, filtering, grouping, summary statistics, and much more. Similarly, users also can create SparkR DataFrames from local R data frames, or from any Spark supported data sources such as Hive, HDFS, Parquet, or JSON. Technically speaking, the concept of Spark DataFrame is a tabular data object akin to R's native DataFrame (https://cran.r-project.org/web/packages/dplyr/vignettes/data_frames.html), which on the other hand, is syntactically similar to `dplyr` (an R package, refer

to https://cran.rstudio.com/web/packages/dplyr/vignettes/introduction.html), but is stored in the cluster setting instead.

Spark machine learning libraries

In this section, we will describe two main machine learning libraries (Spark MLib and Spark ML) and the most widely used implemented algorithms. The ultimate target is to provide you with some familiarization about the machine learning treasures of Spark since many people still think that Spark is only a general-purpose in-memory big data processing or cluster computing framework. However, this is not like that, rather this information would help you to understand what could be done with the Spark machine learning APIs. In addition, this information would help you to explore and will increase the usability while deploying real-life machine learning pipelines using Spark MLlib and Spark ML.

Machine learning with Spark

In the pre-Spark era, big data modelers typically used to build their ML models. Where a model is prepared through a training process where it is required to make predictions and is corrected when those predictions are wrong. In short, an ML model is an object that takes an input, does some processing, and finally produces the output. Those models were commonly constructed using statistical languages such as R and SAS. Then the data engineers used to re-implement the same model in Java to deploy on Hadoop. However, this kind of workflow lacks efficiency, scalability, throughput, and accuracy with extended execution time. Using Spark, the same ML model can be built, adopted, and deployed, making the whole workflow much more efficient, robust, and faster and that allows you to provide hands-on insight to increase the performance. The main goal of Spark machine learning libraries is to make practical machine learning applications scalable, faster, and easy. It consists of common and widely used machine learning algorithms and their utilities, including classification, regression, clustering, collaborative filtering, and dimensionality reduction. It is divided into two packages: Spark MLlib (`spark.mllib`) and Spark ML (`spark.ml`).

Spark MLlib

MLlib is the machine learning library of Spark. It is a distributed, low-level library written with Scala, Java, and Python against Spark core runtime. MLlib mainly focus on learning algorithms and their proper utilities to not only provide machine learning analytical capabilities. The major learning utilities include classification, regression, clustering, recommender system, and dimensionality reduction. In addition, it also aids to optimize the general purpose primitives for developing large-scale machine learning pipelines. As stated earlier, MLlib comes with some exciting APIs written in Java,

Scala, R, and Python. The main components of Spark MLlib are described in the following sections.

Data types

Spark provides support of local vectors and matrix data types stored on a single machine, as well as distributed matrices backed by one or multiple RDDs. Local vectors and matrices are simple data models that serve as public interfaces. The vector and matrix operations are heavily dependent on the linear algebra operation, and you are recommended to gain some background knowledge before using these data types.

Basic statistics

Spark not only provides a column summary and basic statistics to be performed on RDDs, but it also supports calculating the correlation between two series of data or more complex correlation operations, such as pairwise correlations among many series of data, which are a common operation in statistics. However, currently Pearson's and Spearman's correlations are only supported and more are to be added in future Spark releases. Unlike the other statistical function, stratified sampling is also supported by Spark and can be performed on RDD's as key-value pairs; however, some functionalities are yet to be added to Python developers.

Spark provides only the Pearson's chi-squared test for hypothesis testing for its goodness of fit and independence of a claim hypothesis, which is a powerful technique in statistics that determines whether a result is statistically significant to satisfy the claim. Spark also provides online implementations of some tests to support use cases such as A/B testing as streaming significance testing typically performed on real-time streaming data. Another exciting feature of Spark is the factory methods to generate random double RDDs or vector RDDs that are useful for randomized algorithms, prototyping, performance, and hypothesis testing. Other functionality in the current Spark MLib provides computation facilities of kernel density estimation from sample RDDs, which is a useful technique for visualizing empirical probability distributions.

Classification and regression

Classification is a typical process that helps new data objects and components to be organized, differentiated, and understood or belong in a certain way on the basis of training data. In statistical computing, two types of classification exist, binary classification (also commonly referred to as binomial classification) and multiclass classification. Binary classification is the task of classifying data objects of a given observation into two groups. **Support Vector Machines** (**SVMs**), logistic regression, decision trees,

random forests, gradient-boosted trees, and Naive Bayes have been implemented up to the latest release of Spark.

Multiclass classification, on the other hand, is the task of classifying data objects of a given observation into more than two groups. The logistic regression, decision trees, random forests, and naive Bayes are implemented as multiclass classification. However, more complex classification algorithms such as multi-level classification and multiclass perceptron have not been implemented yet. The regression analysis is also a statistical process that estimates relationships among variables or observation. Other than the classification process, regression analysis involves several techniques for modeling and analyzing data objects. Currently, the following algorithms are supported by Spark MLlib library:

- Linear least squares
- Lasso
- Ridge regression
- Decision trees
- Random forests
- Gradient-boosted trees
- Isotonic regression

Recommender system development

An intelligent and scalable recommender system is an emerging application that is currently being developed by many enterprises to expand their business and cost towards automating recommendation for customers. The collaborative filtering approach is the most widely used algorithm in the recommender system, aiming to fill in the missing entries of a user-item association matrix. For example, Netflix is an example who could manage to reduce their movie recommendation by several million dollars. However, the current implementation of Spark MLlib provides only the model-based collaborative filtering technique.

The pros of a model-based collaborative filtering algorithm are users and products that can be described by a small set of latent factors to predict missing entries using the **Alternating Least Squares** (**ALS**) algorithm. The con is that user rating or feedback cannot be taken into consideration for predicting an interest. Interestingly, open source developers are also working to develop a memory-based collaborative filtering technique to be incorporated into Spark MLib in which user rating data could be used to compute the similarity between users or items making the ML model more versatile.

Clustering

Clustering is an unsupervised machine learning problem/technique. The aims are to group subsets of entities with one another based on some

notion of similarity that is often used for exploratory analysis and for developing hierarchical supervised learning pipelines. Spark MLib provides support for various clustering models such as K-means, Gaussian matrix, **Power Iteration Clustering** (**PIC**), **Latent Dirichlet Allocation** (**LDA**), Bisecting K-means, and Streaming K-means from real time streaming data. We will discuss more on supervised/unsupervised and reinforcement learning in upcoming chapters.

Dimensionality reduction

Working with high-dimensional data is cool and demanding to meet the big data related complexities. However, one of the problems with high-dimensional data is unwanted features or variables. Since all of the measured variables might not be important for building the model, to answer the questions of interest you might need to reduce the search space. Therefore, based on certain considerations or requirements, we need to reduce the dimension of the original data before creating any model without sacrificing the original structure.

The current implementation of MLib API supports two types of dimensionality reduction techniques: **Singular Value Decomposition** (**SVD**) and **Principal Component Analysis** (**PCA**) for tall-and-skinny matrices that are stored in row-oriented formats and for any vectors. The SVD technique has some performance issues; however, PCA is the most widely used technique in dimensionality reduction. These two techniques are very useful in large scale ML applications, but they require strong background knowledge of linear algebra.

Feature extraction and transformation

Spark provides different techniques for making the feature engineering easy to use through the **Term frequency-inverse document frequency** (**TF-IDF**), **Word2Vec**, **StandardScaler**, **ChiSqSelector**, and so on. If you are working or planning to work in the area of mining towards building a text mining ML application, TF-IDF would be an interesting option from Spark MLlib. TF-IDF provides a feature vectorization method to reflect the importance of a term to a document in the corpus that is very helpful to develop a text analytical pipeline.

In addition, you might be interested in using the Word2Vec computers distributed vector representation of the words or corpus on your ML application for text analysis. This feature of Word2Vec will eventually make your generalization and model estimation more robust in the area of the novel patterns. You also have the StandardScaler to normalize the extracted features by scaling to the unit variance or by removing the mean based on column summary statistics. It is needed in the pre-processing step while building a scalable ML application typically performed on the samples in the training dataset. Well, suppose you have extracted features

through this method, now you will need to select the features to be incorporated into your ML model. Therefore, you might also be fascinated in the ChiSqSelector algorithm of Spark MLlib for feature selection. ChiSqSelector tries to identify relevant features during the ML model building. The reason is obviously to reduce the size of the feature space as well as the search space in a tree-based approach and to improve both speed and statistical learning behavior in the reinforcement learning algorithms.

Frequent pattern mining

Mining frequent items, maximal frequent patterns/itemsets, contiguous frequent patterns or subsequences, or other substructures is usually among the first steps to analyze a large-scale dataset before starting to build your ML models. The current implementation of Spark MLib provides a parallel implementation of FP-growth for mining frequent patterns and the association rules. It also provides the implementation of another popular algorithm, PrefixSpan, for mining sequence patterns. However, you will have to customize the algorithm for mining maximal frequent patterns accordingly. We will provide a scalable ML application for mining privacy, and preserving maximal frequent patterns in upcoming chapters.

Spark ML

Spark ML is an ALPHA component that adds a new set of machine learning APIs to let users quickly assemble and configure practical machine learning pipelines on top of DataFrames. Before praising the features and advantages of Spark ML, we should know about the DataFrames machine learning techniques that can be applied and developed to a wide variety of data types, such as vectors, unstructured (that is, raw texts), images, and structured data. In order to support a variety of data types to make the application development easier, recently, Spark ML has adopted the DataFrame and Dataset from Spark SQL.

A DataFrame or Dataset can be created either implicitly or explicitly from an RDD of objects that supports the basic and structured types. The goal of Spark ML is to provide a uniform set of high-level APIs built on top of DataFrames and datasets rather than RDDs. It helps the users to create and tune practical machine learning pipelines. The Spark ML also provides for the feature estimators and transformers for developing scalable ML pipelines. Spark ML systematizes many ML algorithms and APIs to make it even easier to combine multiple algorithms into a single pipeline, or data workflow that uses the concept of DataFrame and datasets.

The three basic steps in feature engineering are feature extraction, feature transformation, and selection. Spark ML provides implementation of several algorithms to make these steps easier. Extraction provides the facility for extracting features from raw data, whereas transformation provides the

facility of scaling, converting, or modifying features that are found from the extraction step and the selection helps to select a subset from a larger set of features from the second step. Spark ML also provides several classifications (logistic regression, decision tree classifier, random forest classifier, and more), regression (liner regression, decision tree regression, random forest regression, survival regression, and gradient-boosted tree regression), decision tree and tree ensembles (random forest and gradient-boosted trees), as well as clustering (K-means and LDA) algorithms implemented for developing ML pipelines on top of DataFrames. We will discuss more on RDDs and DataFrames and their underlying operations in Chapter 3, *Understanding the Problem by Understanding the Data*.

Installing and getting started with Spark

Spark is Apache Hadoop's successor. Therefore, it would be better to install and work Spark into a Linux-based system even though you can also try on Windows and Mac OS. It is also very possible to configure your Eclipse environment to work with Spark as a Maven project on any OS and bundle your applications as a jar file with all the dependencies. Secondly, you can try running an application from the Spark shell (Scala shell to be more specific) following the same fashion as SQL or R programming:

The third way is from the command line (Windows)/Terminal (Linux/Mac OS). At first you need to write your ML application using Scala or Java and prepare the jar file with the required dependencies. Then the jar file can be submitted to a cluster to compute a Spark job.

We will show how to develop and deploy a Spark ML application in three ways. However, the very first perquisite is to prepare your Spark application development environment. You can install and configure Spark on a number of operating systems, including:

- Windows (XP/7/8/10)
- Mac OS X (10.4.7+)
- Linux distribution (including Debian, Ubuntu, Fedora, RHEL, CentOS, and so on)

> **Note**
>
> Please check, the Spark website at https://spark.apache.org/documentation.html for the Spark version and OS related documentation. The following steps show you how to install and configure Spark on Ubuntu 14.04 (64-bit). Please note that Spark 2.0.0 runs on Java 7+, Python 2.6+/3.4+, and R 3.1+. For the Scala API, Spark 2.0.0 uses Scala 2.11. Therefore, you will need to use a compatible Scala version (2.11.x).

Step 1: Java installation

Java installation should be considered as one of the mandatory requirements in installing Spark since Java and Scala-based APIs require having a Java virtual machine installed on the system. Try the following command to verify the Java version:

```
$ java -version
```

If Java is already installed on your system, you should see the following message:

```
java version "1.7.0_80"
Java(TM) SE Runtime Environment (build
1.7.0_80-b15)
Java HotSpot(TM) 64-Bit Server VM (build
24.80-b11, mixed mode)
```

In case you do not have Java installed on your system, make sure you install Java before proceeding to the next step. Please note that to avail and enjoy the lambda expression support it is recommended to install Java 8 on your system, preferably JDK and JRE both. Although for Spark 1.6.2 and prior releases Java 7 should be enough:

```
$ sudo apt-add-repository ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt-get install oracle-java8-installer
```

After installing, don't forget to set `JAVA_HOME`. Just apply the following commands (we assume Java is installed at `/usr/lib/jvm/java-8-oracle`):

```
$ echo "export
JAVA_HOME=/usr/lib/jvm/java-8-
oracle" >> ~/.bashrc
$ echo "export PATH=$PATH:$JAVA_HOME/bin" >>
~/.bashrc
```

You can add these environmental variables manually in the `.bashrc` file located in the home directory. If you cannot find the file, probably it is hidden so it needs to be explored. Just go to the **view** tab and enable the **Show hidden file**.

Step 2: Scala installation

Spark is written in Scala itself; therefore, you should have Scala installed on your system. Checking this is so straight forward by using the following command:

```
$ scala -version
```

If Scala is already installed on your system, you should get the following message on the terminal:

```
Scala code runner version 2.11.8 -- Copyright
2002-2016, LAMP/EPFL
```

Note that during the writing of this installation, we used the latest version of Scala, that is 2.11.8. In case you do not have Scala installed on your system, make sure you install it, so before proceeding to the next step, you can download the latest version of Scala from the Scala website at http://www.scala-lang.org/download/. After the download has finished, you should find the Scala `tar` file in the download folder:

1. Extract the Scala `tar` file by extracting, from its location or type the following command for extracting the Scala tar file from the terminal:

```
$ tar -xvzf scala-2.11.8.tgz
```

2. Now move the Scala distribution to the user's perspective (for example, `/usr/local/scala`) by the following command or do it manually:

```
$ cd /home/Downloads/
$ mv scala-2.11.8 /usr/local/scala
```

3. Set the Scala home:

```
$ echo "export
SCALA_HOME=/usr/local/scala/scala-2.11.8" >>
        ~/.bashrc
$ echo "export PATH=$PATH:$SCALA_HOME/bin" >>
~/.bashrc
```

4. After installation has been completed, you should verify it using the following command:

```
$ scala -version
```

5. If Scala has successfully been configured on your system, you should get the following message on your terminal:

```
Scala code runner version 2.11.8 -- Copyright
2002-2016, LAMP/EPFL
```

Step 3: Installing Spark

Download the latest version of Spark from the Apace Spark website at https://spark.apache.org/downloads.html. For this installation step, we used the latest Spark stable release 2.0.0 version pre-built for Hadoop 2.7 and later. After the download has finished, you will find the Spark `tar` file in the download folder:

1. Extract the Scala `tar` file by extracting it from its location or type the following command for extracting the Scala `tar` file from the terminal:

```
$ tar -xvzf spark-2.0.0-bin-hadoop2.7.tgz
```

2. Now move the Scala distribution to the user's perspective (for example, `/usr/local/spark`) by the following command or do it manually:

```
    $ cd /home/Downloads/
    $ mv
spark-2.0.0-bin-hadoop2.7 /usr/local/spark
```

3. To set after Spark installing, just apply the following commands:

```
$ echo "export
SPARK_HOME=/usr/local/spark/spark-2.0.0-bin-
hadoop2.7" >>
      ~/.bashrc
$ echo "export PATH=$PATH:$SPARK_HOME/bin" >>
~/.bashrc
```

Step 4: Making all the changes permanent

Source the `~/.bashrc` file using the following command to make the changes permanent:

```
$ source ~/.bashrc
```

If you execute the `$ vi ~/. bashrc` command, you will see the following entry in your `bashrc` file as follows:

```
export JAVA_HOME=/usr/lib/jvm/java-8-oracle
export
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/ga
8-oracle/bin:/usr/lib/jvm/java-8-
```

```
oracle/db/bin:/usr/lib/jvm/java-8-
oracle/jre/bin: /usr/lib/jvm/java-8-oracle/bin
export SCALA_HOME=/usr/local/scala/scala-2.11.8
export
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/ga
8-oracle/bin:/usr/lib/jvm/java-8-
oracle/db/bin:/usr/lib/jvm/java-8-
oracle/jre/bin: /bin
export
SPARK_HOME=/usr/local/spark/spark-2.0.0-bin-
hadoop2.7
export
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/ga
8-oracle/bin:/usr/lib/jvm/java-8-
oracle/db/bin:/usr/lib/jvm/java-8-
oracle/jre/bin: /bin
```

Step 5: Verifying the Spark installation

The verification of the Spark installation is shown in the following screenshot:

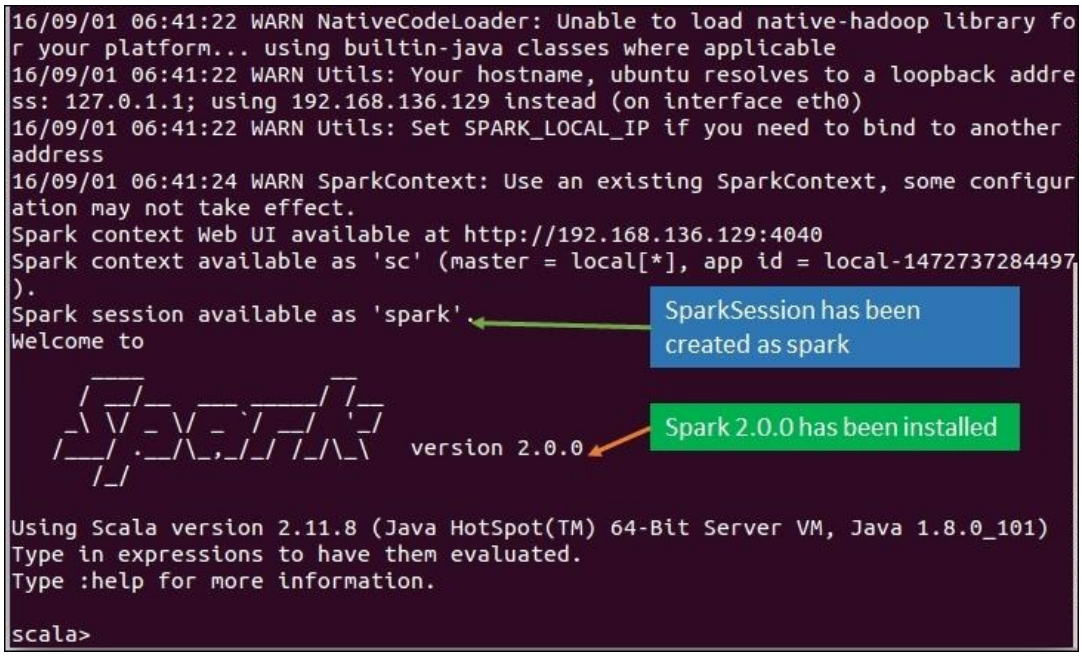


Figure 6: The Spark shell confirms the successful Spark installation.

Write the following command to open the Spark shell to verify if Spark has been configured successfully:

```
$ spark-shell
```

If Spark is installed successfully, you should see the following message (*Figure 6*).

The Spark server will start on localhost at port `4040`, more precisely at `http://localhost:4040/` (*Figure 7*). Just move there to make sure if it's really running:

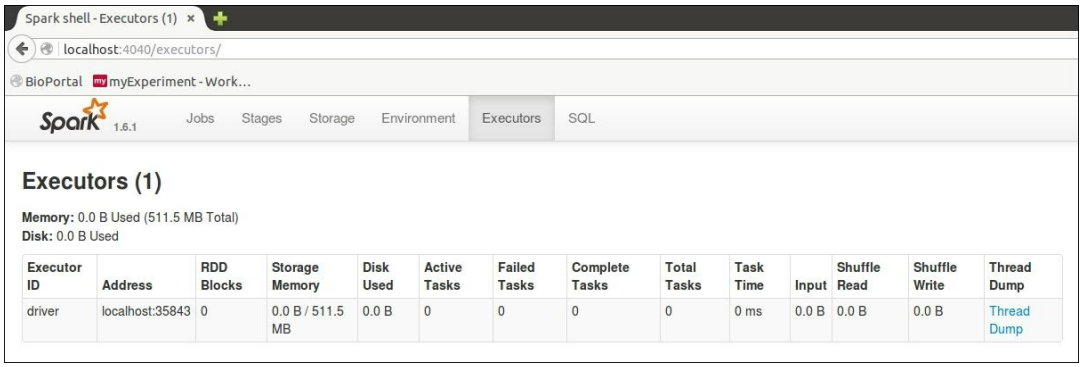


Figure 7: Spark is running as a local web server.

Well done! Now you are ready to start writing the Scala code on the Spark shell.

Packaging your application with dependencies

Now we will show you how to package the applications as a Java archive (JAR) file with all the required dependencies on Eclipse, which is an **Integrated Development Environment** (**IDE**) and an open source tool for Java development as an Apache Maven project (https://maven.apache.org/). Maven is a software project management and comprehension tool like Eclipse. Based on the concept of a **Project Object Model** (**POM**), Maven can manage a project's build, reporting and documenting from a central piece of information.

Note that it is possible to export an ML application written in Java or Scala as an archive/executable jar file using Command Prompt. However, for the simplicity and faster application development we will use the same as the Maven project using Eclipse so that readers can enjoy the same facility to submit the application to the master node for computation. Now let's move to the discussion of exporting frequent pattern mining applications as a jar file with all the dependencies.

Step 1: Creating a Maven project in Eclipse

On successful creation of a sample Maven project, you will see the following project structure in Eclipse shown in *Figure 8*:

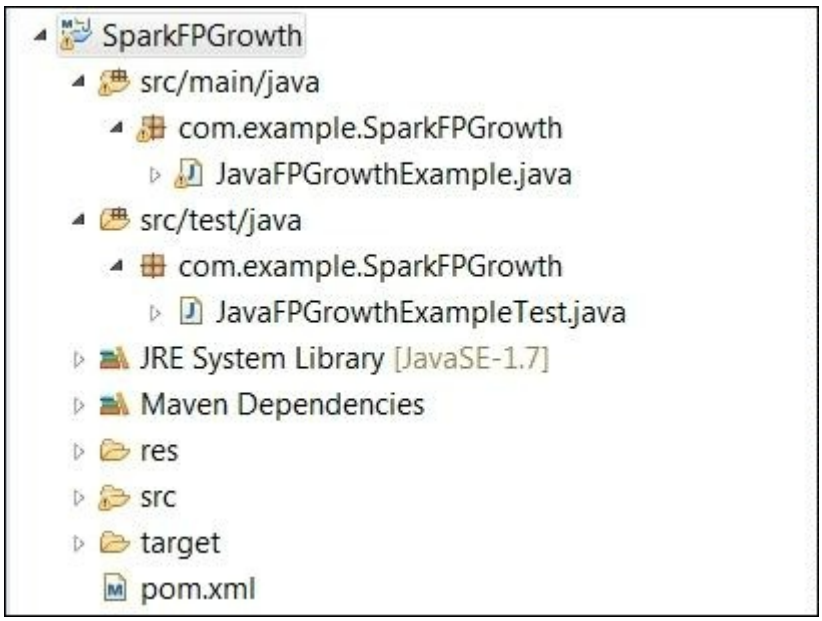


Figure 8: Maven project structure in Eclipse.

Step 2: Application development

Create a Java class and copy the following source code to under the `src/main/java` directory for the mining frequent pattern. Here, inputting the filename has been specified by the filename string to be provided through the command line argument or by specifying the source manually. For the time being, we have just provided line comments, however, you will get to know details from Chapter 3, *Understanding the Problem by Understanding the Data* and onwards:

```
import java.util.Arrays;
import java.util.List;
import org.apache.spark.api.java.JavaRDD;
import
org.apache.spark.api.java.function.Function;
import org.apache.spark.mllib.fpm.FPGrowth;
import org.apache.spark.mllib.fpm.FPGrowthModel;
import org.apache.spark.rdd.RDD;
import org.apache.spark.sql.SparkSession;

public class JavaFPGrowthExample {
  public static void main(String[] args) {
   //Specify the input transactional as command
line argument
   String fileName = "input/input.txt";
   //Configure a SparkSession as spark by
specifying the application name, master URL,
Spark config, and Spark warehouse directory
  SparkSession spark = SparkSession
                  .builder()
                  .appName("JavaFPGrowthExample")
                  .master("local[*]")

 .config("spark.sql.warehouse.dir", "E:/Exp/")
                  .getOrCreate();

   //Create an initial RDD by reading the input
database
   RDD<String> data =
spark.sparkContext().textFile(fileName, 1);

   //Read the transactions by tab delimiter &
mapping RDD(data)
   JavaRDD<List<String>> transactions =
data.toJavaRDD().map(
                   new Function<String,
List<String>>(){
                   public List<String>
call(String line) {
                           String[] parts =
line.split(" ");
                           return
Arrays.asList(parts);
                                   }
                              });
```

```
  //Create FPGrowth object by min. support & no.
of partition
  FPGrowth fpg = new  FPGrowth()
                       .setMinSupport(0.2)
                       .setNumPartitions(10);

  //Train and run your FPGrowth model using the
transactions
  FPGrowthModel<String> model =
fpg.run(transactions);

  //Convert and then collect frequent patterns as
Java RDD. After that print the frequent patterns
along with their support
    for (FPGrowth.FreqItemset<String> itemset :


 model.freqItemsets().toJavaRDD().collect()) {
       System.out.println(itemset.javaItems()
                             + "==> " +
itemset.freq());
      }
    }
  }
```

Step 3: Maven configuration

Now you need to configure Maven specifying related dependencies and configurations. First, edit your existing `pom.xml` file to copy each XML source code snippets inside the `<dependencies>` tag. Please note that your dependencies might be different based on Spark release so change the version accordingly:

1. Spark core dependency for Spark context and configuration:

```
<dependency>
<groupId>org.apache.spark</groupId>

<artifactId>spark-core_2.11</artifactId>
<version>2.0.0</version>
</dependency>
```

2. Spark MLib dependency for the FPGrowth:

```
<dependency>
  <groupId>org.apache.spark</groupId>

<artifactId>spark-mllib_2.11</artifactId>
<version>2.0.0</version>
</dependency>
```

Now you need to add the build requirements. Copy the following code snippets immediately after the `</dependencies>` tag. Here we are specifying the `<groupId>` as maven plugins, `<artifactId>` as maven shade plugins, and specifying the jar file naming convention using the `<finalName>` tag. Make sure that you have specified the source code download plugin, set the compiler level, and set the assembly plugin for the Maven, described as follows:

1. Specify the source code download plugin with Maven:

```
        <plugin>

 <groupId>org.apache.maven.plugins</groupId>

 <artifactId>maven-eclipse-plugin</artifactId>
        <version>2.9</version>
        <configuration>

 <downloadSources>true</downloadSources>

 <downloadJavadocs>false</downloadJavadocs>
        </configuration>
      </plugin>
```

2. Set the compiler level for Maven:

```
      <plugin>

 <groupId>org.apache.maven.plugins</groupId>

 <artifactId>maven-compiler-
plugin</artifactId>
        <version>2.3.2</version>
      </plugin>
      <plugin>

 <groupId>org.apache.maven.plugins</groupId>

 <artifactId>maven-shade-plugin</artifactId>
        <configuration>
          <shadeTestJar>true</shadeTestJar>
        </configuration>
      </plugin>
```

3. Set the Maven assembly plugin:

```
      <plugin>

 <groupId>org.apache.maven.plugins</groupId>

 <artifactId>maven-assembly-
plugin</artifactId>
        <version>2.4.1</version>
```

```
        <configuration>
          <!-- get all project dependencies
-->
          <descriptorRefs>

 <descriptorRef>jar-with-
dependencies</descriptorRef>
          </descriptorRefs>
          <!-- MainClass in mainfest make a
executable jar -->
          <archive>
            <manifest>
<mainClass>com.example.SparkFPGrowth.JavaFPGrowthExample</mainClass>
             </manifest>
          </archive>
          <property>

 <name>oozie.launcher.mapreduce.job.user.classpath.first</name>
            <value>true</value>
          </property>

 <finalName>FPGrowth-
${project.version}</finalName>
        </configuration>
        <executions>
          <execution>
            <id>make-assembly</id>
            <!-- bind to the packaging phase
-->
            <phase>package</phase>
            <goals>
              <goal>single</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
```

The complete `pom.xml` file, input data, and Java source file can be downloaded from our GitHub repositories at https://github.com/rezacsedu/PacktMLwithSpark. Please note that we used Eclipse Mars Eclipse IDE for Java Developers, and the version was Mars Release (4.5.0). You can go for this version or another distribution such as Eclipse Luna.

Step 4: The Maven build

In this section, we will describe how to create a Maven friendly project on Eclipse. After you have followed all the steps, you will be able to run a Maven project successfully. The steps should be in a chronological order as follows:

1. Run your project as Maven install.

2. If your code and maven configuration file are okay, the maven build will be successful.
3. Build the maven project.
4. Right-click on your project and run the maven project as **Maven build...** and write `clean package` in the **Goals** option.
5. Check the Maven dependencies.
6. Expand the Maven dependencies tree and check if the required jar files have been installed.
7. Check if the jar file is generated with dependencies.
8. As we specified, you should find two jar files under the `/target` directory tree (refer to *Figure 9*). The packaging file should contain exactly the same name as specified in the `<finalName>` tag. Now move your code (jar file) to a directory that aligns our experiment (that is, `/user/local/code`) and your data (that is, `/usr/local/data/`). We will use this jar file to execute the Spark job on an AWS EC2 cluster in a later stage. We will discuss the input dataset in the next step.

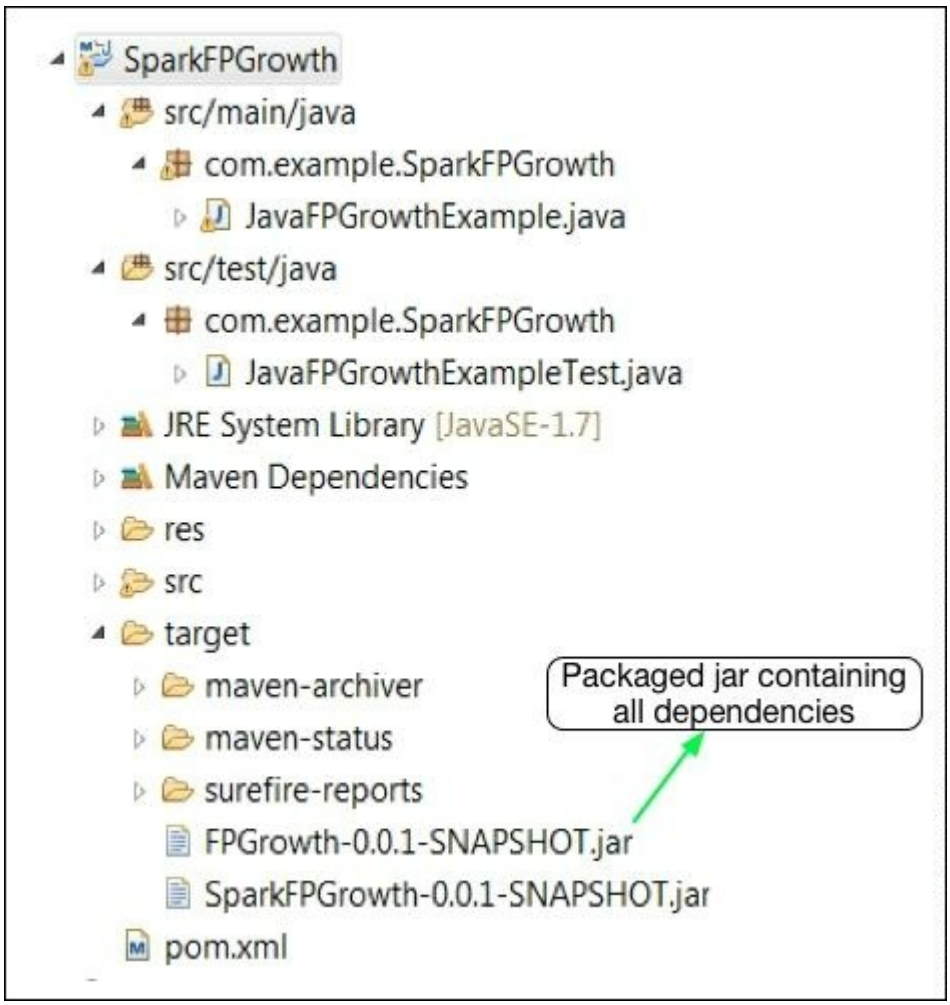


Figure 9: Maven project with the jar generated with all the required dependencies on Eclipse.

Running a sample machine learning application

In this section, we will describe how to run a sample machine learning application from the Spark shell, on the local machine as stand-alone mode, and finally we will show you how to deploy and run the application on the Spark cluster using Amazon EC2 (https://aws.amazon.com/ec2/).

Running a Spark application from the Spark shell

Please note that this is just an exercise that checks the installation and running of a sample code. Details on machine learning application development will be covered from Chapter 3, *Understanding the Problem by Understanding the Data* to *Chapter 9*, *Advanced Machine Learning with Streaming and Graph Data*.

Now we will further proceed with one of the popular machine learning problem also called frequent pattern mining using the Frequent Pattern-growth or FP-growth. Suppose we have a transactional database as shown in the following table. Each line indicates a transaction done by a particular customer. Our target is to find the frequent patterns from the database, which is the prerequisite for calculating association rules (https://en.wikipedia.org/wiki/Association_rule_learning) from customer purchase rules. Save this database as `input.txt` in the `/usr/local/data` directory without transaction IDs:

Transaction ID	**Transaction**
1	A B C D F
2	A B C E
3	B C D E F
4	A C D E
5	C D F
6	D E F
7	D E

8	C D F
9	C F
10	A C D E

Table 1: A transactional database.

Now let's move to the Spark shell by specifying the master and number of the computational core to use as standalone mode (here are four cores, for example):

```
$ spark-shell --master "local[4]"
```

Step 1: Loading packages

Load the required FPGrowth package and other dependent packages:

```
scala>import org.apache.spark.mllib.fpm.FPGrowth
scala>import org.apache.spark.{SparkConf,
SparkContext}
```

Step 2: Creating Spark context

To create a Spark context, at first you need to configure the Spark session by mentioning the application name and master URL. Then you can use the Spark configuration instance variable to create a Spark context as follows:

```
val conf = new
SparkConf().setAppName(s"FPGrowthExample with
$params")
val sc = new SparkContext(conf)
```

Step 3: Reading the transactions

Let's read the transactions as RDDs on the created Spark Context (`sc`) (see *Figure 6*):

```
scala> val transactions =
sc.textFile(params.input).map(_.split("
")).cache()
```

Step 4: Checking the number of transactions

Here is the code for checking the number of transactions:

```
Scala>println(s"Number of transactions:
${transactions.count()}")
Number of transactions: 22
Scala>
```

Step 5: Creating an FPGrowth model

Create the model by specifying the minimum support threshold (see also https://en.wikipedia.org/wiki/Association_rule_learning) and the number of partitions:

```
scala>val model = new FPGrowth()
                  .setMinSupport(0.2)
                  .setNumPartitions(2)
                  .run(transactions)
```

Step 6: Checking the number of frequent patterns

The following code explains how to check the number of frequent patterns:

```
scala> println(s"Number of frequent itemsets:
    ${model.freqItemsets.count()}")
Number of frequent itemsets: 18
Scala>
```

Step 7: Printing patterns and support

Print the frequent pattern and their corresponding support/frequency counts (see *Figure 10*). Spark job will be running on localhost (refer to *Figure 11*):

```
scala> model.freqItemsets.collect().foreach
{ itemset => println(itemset.items.mkString("[",
",", "]") + ", " + itemset.freq)}
```

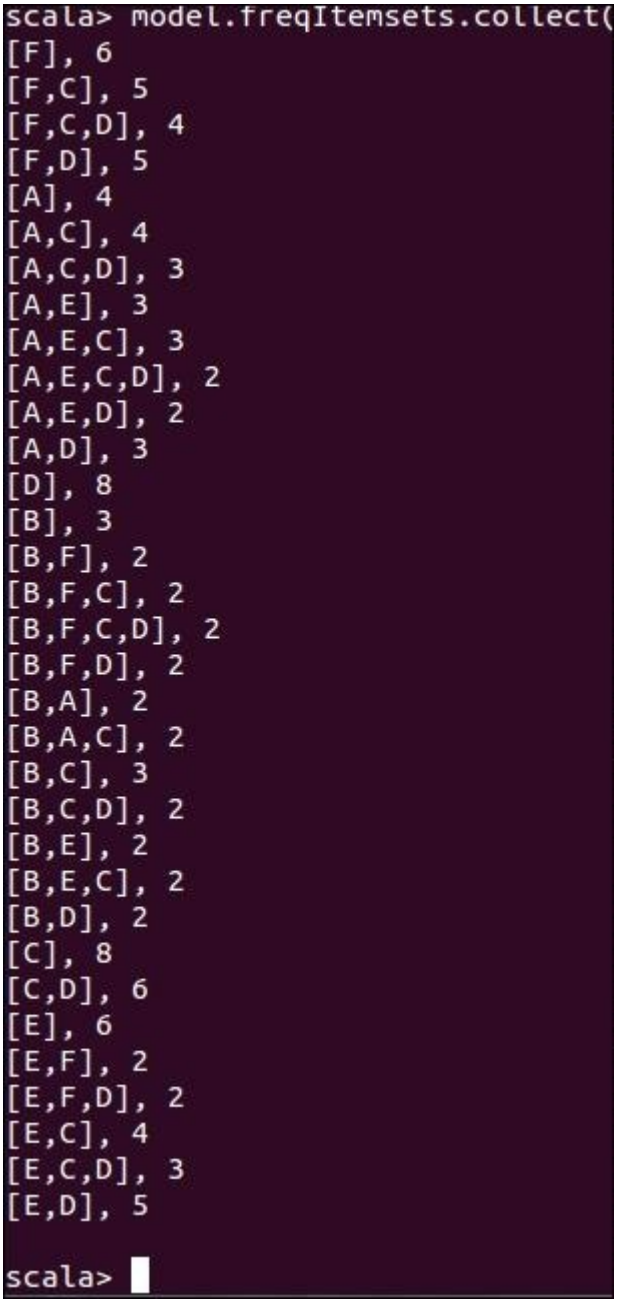


Figure 10: Frequent patterns.

Running a Spark application on the local cluster

Once a user application is bundled as either a jar file (written in Scala or Java) or a Python file, it can be launched using the `spark-submit` script located under the bin directory in the Spark distribution.

As per the API documentation provided by the Spark website at http://spark.apache.org/docs/2.0.0-preview/submitting-applications.html, this script takes care of setting up the class path with Spark and its dependencies, and can support different cluster managers and deploys

models that Spark supports. In a nutshell, Spark job submission syntax is as follows:

```
$spark-
submit [options] <app-jar | python-file> [app
arguments]
```

Here, `[options]` can be: `--class <main-class>--master <master-url>--deploy-mode <deploy-mode>`, and a number of other options.

To be more specific, `<main-class>` is the name of the main class name, which is the entry point for our application. `<master-url>` specifies the master URL for the cluster (for example, `spark://HOST:PORT` for connecting to the given Spark standalone cluster master, local for running Spark locally with one worker thread with no parallelism at all, `local [k]` for running a Spark job locally with K worker threads, which is the number of cores on your machine, `local[*]` for running a Spark job locally with as many worker threads as logical cores on your machine have, and `mesos://IP:PORT` for connecting to the available Mesos cluster, and even you could submit your job to the Yarn cluster - for more, see http://spark.apache.org/docs/latest/submitting-applications.html#master-urls).

`<deploy-mode>` is used to deploy our driver on the worker nodes (cluster) or locally as an external client (client). `<app-jar>` is the jar file we just built, including all the dependencies. `<python-file>` is the application main source code written using Python. `[app-arguments]` could be an input or output argument specified by an application developer:

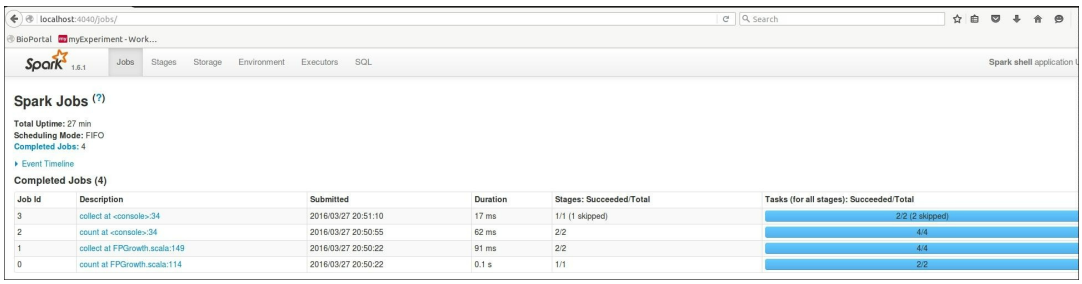


Figure 11: Spark job running on localhost

Therefore, for our case, the job submit syntax would be as follows:

```
$./bin/spark-
submit --class
com.example.SparkFPGrowth.JavaFPGrowthExample
--
master local[4]
```

```
FPGrowth-0.0.1-SNAPSHOT-jar-with-
dependencies.jar input.txt
```

Here, `JavaFPGrowthExample` is the main class file written in Java; local is the master URL; `FPGrowth-0.0.1-SNAPSHOT-jar-with-dependencies.jar` is the application `jar` file we just generated by maven project; `input.txt` is the transactional database as the text file, and output is the directory where the output to be generated (in our case, the output will be shown on the console). Now let's submit this job to be executed locally.

If it is executed successfully, you will find the following message including the output in *Figure 12* (abridged):

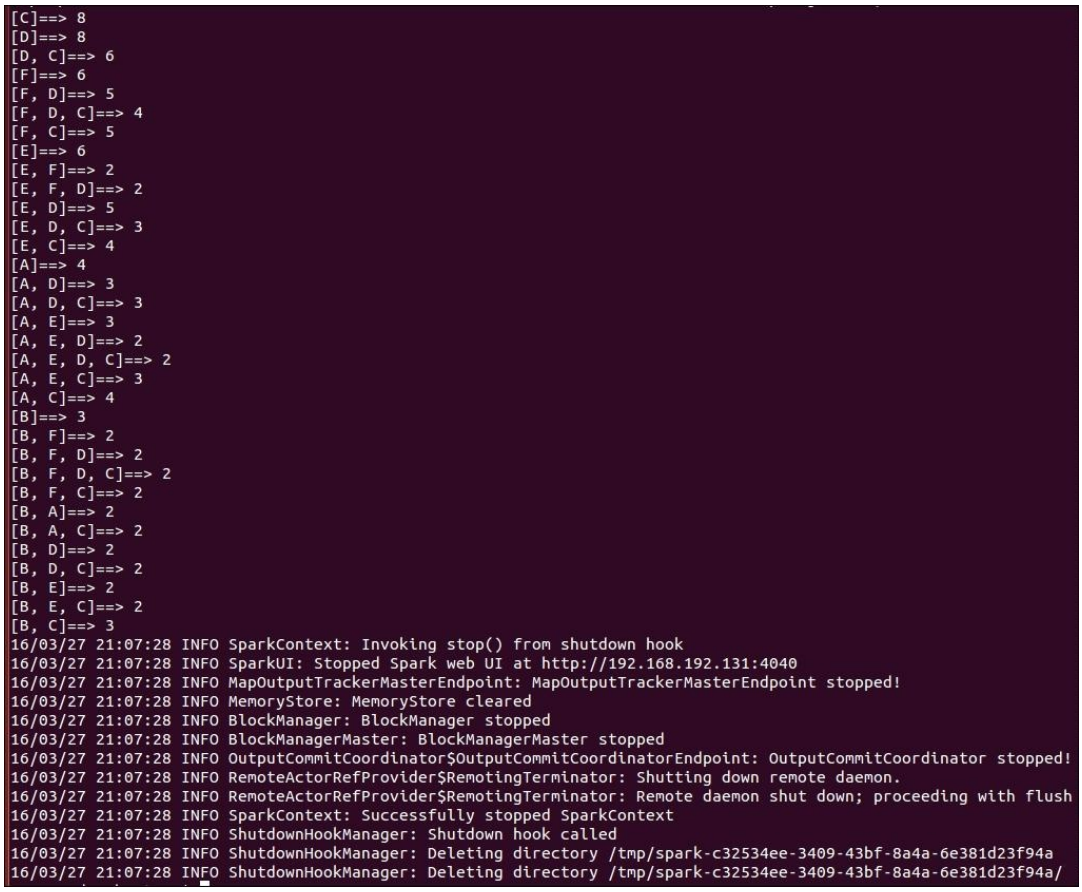


Figure 12: Spark job output on the terminal.

Running a Spark application on the EC2 cluster

In the previous section, we illustrated how to submit spark jobs in local or standalone mode. Here, we are going to describe how to run a spark application in cluster mode. To make our application run on the spark cluster mode, we consider the **Amazon Elastic Compute Cloud** (**EC2**) services, as **Infrastructure as a Service** (**IaaS**) or **Platform as a Service**

(**PaaS**). For pricing and related information, please refer to this URL https://aws.amazon.com/ec2/pricing/.

Step 1: Key pair and access key configuration

We assume you have EC2 accounts already created. The first requirement is to create EC2 key pairs and AWS access keys. The EC2 key pair is the private key that you need when you will make a secure connection through SSH to your EC2 server or instances. To make the key, you have to go through the AWS console at http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-key-pairs.html#having-ec2-create-your-key-pair. Please refer to *Figure 13*, which shows the key pair creation page for an EC2 account:

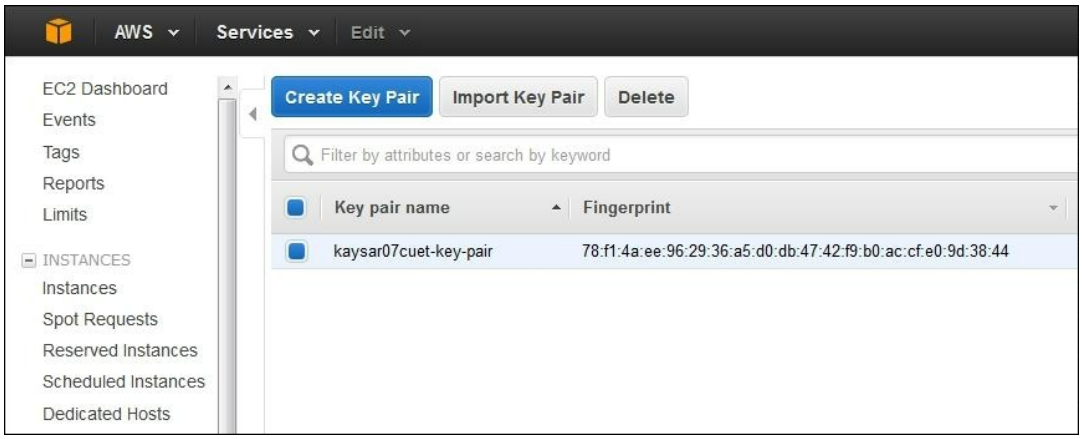


Figure 13: AWS key-pair generation window.

Name it `my-key-pair.pem` once you have downloaded it and saved it on your local machine. Then ensure the permission by executing the following command (you should store this file in a secure location for security purposes, say `/usr/local/key`):

```
$ sudo chmod  400  /usr/local/key/my-key-pair.pem
```

Now what you need is the AWS access keys, the credentials of your account, which are needed if you want to submit your Spark job to compute nodes from your local machine using spark-ec2 script. To generate and download the keys, login to your AWS IAM services at http://docs.aws.amazon.com/IAM/latest/UserGuide/id_credentials_access-keys.html#Using_CreateAccessKey. Upon the download completion (that is, `/usr/local/key`), you need to set two environment variables in your local machine. Just execute the following commands:

```
$ echo "export AWS_ACCESS_KEY_ID=<access_key_id>"
>> ~/.bashrc
```

```
$ echo " export
AWS_SECRET_ACCESS_KEY=<secret_access_key_id>" >>
~/.bashrc
```

Step 2: Configuring the Spark cluster on EC2

Spark distribution (that is, `/usr/local/spark/ec2`) provides a script called `spark-ec2` for launching Spark Clusters in EC2 instances from your local machine (driver program), which helps in launching, managing, and shutting down the Spark Cluster.

> **Note**
>
> Please note that starting a cluster on AWS will cost money. Therefore, it is always a good practice to stop or destroy a cluster when the computation is done. Otherwise, it will incur additional costs. For more about AWS pricing, please refer to this URL https://aws.amazon.com/ec2/pricing/.

Once you execute the following command to launch a new instance, it sets up Spark, HDFS, and other dependencies on the cluster automatically:

```
$./spark-
ec2 --key-pair=<name_of_the_key_pair>
--identity-
file=<path_of_the key_pair>
--instance-
type=<AWS_instance_type > --region=<region>
zone=<zone> --slaves=<number_of_slaves>
--hadoop-major-
version=<Hadoop_version>
--spark-
version=<spark_version> launch <cluster-name>
```

We believe that these parameters are self-explanatory, or alternatively, please see details at http://spark.apache.org/docs/latest/ec2-scripts.html. For our case, it would be something like this:

```
$./spark-
ec2 --key-pair=my-key-pair
--identity-file=/usr/local/key/my-key-
pair.pem  --instance-type=m3.2xlarge
--region=eu-west-
1 --zone=eu-west-1a --slaves=2
--hadoop-major-
```

```
version=yarn --spark-version=1.6.0 launch
ec2-spark-cluster-1
```

It is shown in the following screenshot:

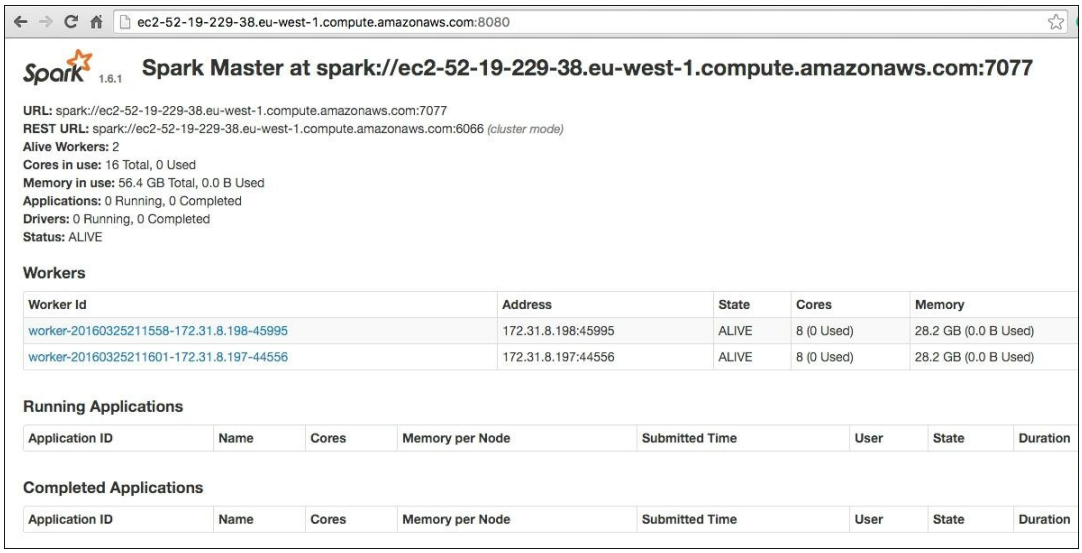


Figure 14: Cluster home.

After the successful completion, spark cluster will be instantiated with two workers (slave) nodes on your EC2 account. This task; however, sometimes might take half an hour approximately depending on your Internet speed and hardware configuration. Therefore, you'd love to have a coffee break. Upon successful competition of the cluster setup, you will get the URL of the Spark cluster on the terminal.

To check to make sure if the cluster is really running, check this URL `https://<master-hostname>:8080` on your browser, where the master hostname is the URL you receive on the terminal. If everything was okay, you will find your cluster is running, see cluster home in *Figure 14*.

Step 3: Running and deploying Spark job on Spark Cluster

Execute the following command to the SSH remote Spark cluster:

```
$./spark-
ec2 --key-pair=<name_of_the_key_pair>
--identity-
file=<path_of_the _key_pair> --region=<region>
login <cluster-name>
```

For our case, it should be something like this:

```
$./spark-
ec2 --key-pair=my-key-pair
--identity-file=/usr/local/key/my-key-
pair.pem --region=eu-west-1 login
ec2-spark-cluster-1
```

Now copy your application (the jar we generated as the Maven project on Eclipse) to a remote instance (that is, `ec2-52-48-119-121.eu-west-1.compute.amazonaws.com` in our case) by executing the following command (in a new terminal):

```
$ scp -i /usr/local/key/my-key-pair.pem
 /usr/local/code/FPGrowth-0.0.1-SNAPSHOT-jar-with-
dependencies.jar
ec2-user@ec2-52-18-252-59.eu-west-
1.compute.amazonaws.com:/home/ec2-user/
```

Then you need to copy your data (`/usr/local/data/input.txt` in our case) to the same remote instance by executing the following command:

```
$ scp
-i /usr/local/key/my-key-
pair.pem /usr/local/data/input.txt
ec2-user@ec2-52-18-252-59.eu-west-
1.compute.amazonaws.com:/home/ec2-user/
```

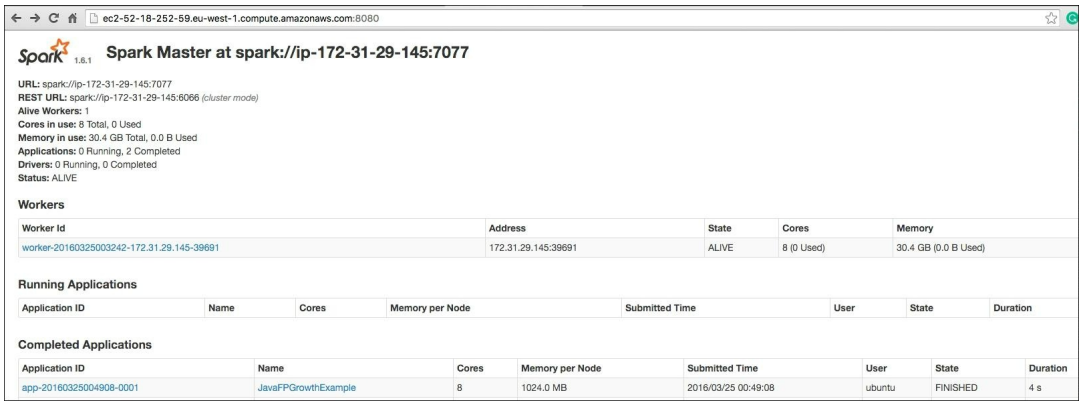


Figure 15: Job running status at Spark cluster.

Well done! You are almost done! Now, finally you will have to submit your Spark job to be computed by the slaves or worker nodes. To do so, just execute the following commands:

```
$./bin/spark-
submit --class
com.example.SparkFPGrowth.JavaFPGrowthExample
--
```

```
master
spark://ec2-52-48-119-121.eu-west-
1.compute.amazonaws.com:7077 /home/ec2-
user/FPGrowth-0.0.1-SNAPSHOT-jar-with-
dependencies.jar /home/ec2-user/input.txt
```

Upon successful completion of the job computation, you are supposed to see the status of your job at port 8080 like *Figure 15* (the output will be shown on the terminal).

Step 4: Pausing and restarting spark cluster

To stop your clusters, execute the following command from your local machine:

```
$./ec2/spark-
ec2 --region=<ec2-region> stop <cluster-name>
```

For our case, it would be:

```
$./ec2/spark-
ec2 --region=eu-west-1 stop ec2-spark-cluster-1
```

To restart the cluster later on, execute the following command:

```
$./ec2/spark-
ec2 -i <key-file> --region=<ec2-region> start
<cluster-name>
```

For our case, it will be something as follows:

```
$./ec2/spark-
ec2
--identity-file=/usr/local/key/my-key-
pair.pem --region=eu-west-1 start
ec2-spark-cluster-1
```

Tip

To terminate your spark cluster:`$./spark-ec2 destroy <cluster-name>`

In our case, it would be: `$./spark-ec2 --region=eu-west-1 destroy ec2-spark-cluster-1`

If you want your application to scale up for large-scale datasets, the fastest way is to load them from Amazon S3 or an Amazon EBS device into an instance of the **Hadoop Distributed File System** (**HDFS**) on your nodes. We will discuss this technique in later chapters throughout practical machine learning examples.

References

- *Resilient Distributed Datasets*: *A Fault-Tolerant Abstraction for In-Memory Cluster Computing*, *Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica.NSDI 2012. April 2012.*
- *Spark*: *Cluster Computing with WorkingSets*, *Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica,HotCloud 2010. June 2010.*

- *Spark SQL*: *Relational Data Processing in Spark*, *Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng*, *Tomer Kaftan*, *Michael J. Franklin*, *Ali Ghodsi*, *Matei Zaharia*, *SIGMOD 2015. June 2015.*
- *Discretized Streams*: *Fault-Tolerant Streaming Computation at Scale, Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, Ion Stoica.SOSP 2013. November 2013.*
- *Discretized Streams*: *An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters*. *Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, Ion Stoica.HotCloud 2012. June 2012.*
- *GraphX*: *Unifying Data-Parallel and Graph-Parallel Analytics*. *Reynold S. Xin*, *Daniel Crankshaw, Ankur Dave, Joseph E.Gonzalez, Michael J. Franklin*, *Ion Stoica.OSDI 2014. October 2014.*
- *MLlib*: *Machine Learning in Apache Spark*, *Meng et al*. *arXiv:1505.06807v1, [cs.LG], 26 May 2015.*
- *Recommender*: *An Analysis of Collaborative Filtering Techniques*, *Christopher R. Aberger*, *Stanford publication, 2014.*

Summary

This ends our rather quick tour of Spark. We have tried to cover some of the most basic features of Spark, its computing paradigm, and getting started with Spark by installing and configuring. Use of Spark ML is recommended if they fit the ML pipeline concept well (for example, feature extraction, transformation, and selection) since it is more versatile and flexible with DataFrames and Datasets. However, according to Apache's documentations, they will keep supporting and contributing Spark MLib along with the active development of Spark ML.

On the other hand, data science developers should be comfortable with using Spark MLlib's features and should expect more features in the future. However, some algorithms are not available or are yet to be added to Spark ML, most notably, dimensionality reduction. Nevertheless, developers can seamlessly combine the implementation of these techniques found in Spark MLib with the rest of the algorithms found in Spark ML as hybrid or interoperable ML applications. We also showed some basic techniques to deploy ML applications on clusters and cloud services, though you can also try other deployment options available.

> **Tip**
>
> For more updates, interested readers should refer to the Spark website at http://spark.apache.org/docs/latest/mllib-guide.html for release dates, APIs, and specifications. As Spark's open source community and developers from all over the globe are continually enriching and updating the implementation, therefore, it is better to be updated.

In the next chapter (Chapter 2, *Machine Learning Best Practices*), we will discuss some best practices while developing advanced machine learning with Spark, including machine learning tasks and classes, some practical machine learning problems and their related discussion, some best practices in machine learning application development, choosing the right algorithm for the ML application, and so on.

Chapter 2. Machine Learning Best Practices

The purpose of this chapter is to provide a conceptual introduction to statistical **machine learning** (**ML**) techniques for those who might not normally be exposed to such approaches during their typical required statistical training. This chapter also aims to take a newcomer from minimal knowledge of machine learning all the way to a knowledgeable practitioner in a few steps. The second part of the chapter is focused on giving some recommendations for choosing the right machine learning algorithms depending on the application types and requirements. It will then lead through some best practices when applying large-scale machine learning pipelines. In a nutshell, the following topics will be discussed in this chapter:

- What is machine learning?
- Machine learning tasks
- Practical machine learning problems
- Large scale machine learning APIs in Spark
- Practical machine learning best practices
- Choosing the right algorithm for your application

What is machine learning?

In this section, we will try to define the term machine learning from the computer science, statistics and data analytical perspectives. Then we will show the steps of analytical machine learning applications. Finally, we will discuss some typical and emerging machine learning tasks and then name some practical machine learning problems that need to be addressed.

Machine learning in modern literature

Let's see how a renowned professor of machine learning, Tom Mitchell, Chair of the CMU Machine Learning Department and Professor at the Carnegie Mellon University defines the term machine learning in his literature (*Tom M. Mitchell, The Discipline of Machine Learning, CMU-ML-06-108, July 2006*, http://www.cs.cmu.edu/~tom/pubs/MachineLearning.pdf):

> *Machine Learning is a natural outgrowth of the intersection of Computer Science and Statistics. We might say the defining question of Computer Science is 'How can we build machines that solve problems, and which problems are inherently tractable/intractable?' The question that largely defines Statistics is 'What can be inferred from data plus a set of modelling*

assumptions, with what reliability?' The defining question for Machine Learning builds on both, but it is a distinct question. Whereas Computer Science has focused primarily on how to manually program computers, Machine Learning focuses on the question of how to get computers to program themselves (from experience plus some initial structure). Whereas Statistics has focused primarily on what conclusions can be inferred from data, Machine Learning incorporates additional questions about what computational architectures and algorithms can be used to most effectively capture, store, index, retrieve and merge these data, how multiple learning subtasks can be orchestrated in a larger system, and questions of computational tractability.

We believe that this definition from Prof. Tom is self-explanatory. However, we will provide some clearer understanding of machine learning in the next two sub-sections from the computer science, statistics, and data analytical perspectives.

Tip

Interested readers should follow other resources to get more insights about machine learning and its theoretical perspective. Here we have provided some links as follows: *Machine learning*: https://en.wikipedia.org/wiki/Machine_learning.

Machine learning: what it is and why matters - http://www.sas.com/en_us/insights/analytics/machine-learning.html.

A Gentle Introduction To Machine Learning: https://www.youtube.com/watch?v=NOm1zA_Cats.

What is machine learning, and how does it work: https://www.youtube.com/watch?v=elojMnjn4kk.

Introduction to Data Analysis using Machine Learning: https://www.youtube.com/watch?v=U4IYsLgNgoY.

Machine learning and computer science

Machine learning is a branch of computer science that studies the design of algorithms that can learn from its heuristics that typically evolved from the study of pattern recognition and computational learning theory in artificial intelligence. An interesting question came into the mind of Alan Turing about the machine, which is, *Can a machine think?* In fact, there are some good reasons to believe a sufficiently complex machine could one day pass the unrestricted Turing test; let's postpone this question until the Turing test, but gets passed. However, machines can learn at least. Subsequently,

Arthur Samuel was the first man who defined the term **machine learning** as a f*ield of study that gives computers the ability to learn without being explicitly programmed* in 1959. Typical machine learning tasks are concept learning, predictive modeling, classification, regression, clustering, dimensionality reduction, recommender system, deep learning and finding useful patterns from the large-scale dataset.

The ultimate goal is to improve the learning in such a way that it becomes automatic, so that no human interactions are needed any more, or the level of human interaction is reduced as much as possible. Although machine learning is sometimes conflated with **Knowledge Discovery and Data Mining** (**KDDM**), the latter sub-field on the other hand focuses more on exploratory data analysis and is known as unsupervised learning - such as clustering analysis, anomaly detection, **Artificial Neural Networks** (**ANN**), and so on.

Other machine learning techniques include supervised learning, where a learning algorithm analyzes the training data and produces an inferred function that can be used for mapping new examples towards prediction. Classification and regression analysis are two typical examples of supervised learning. Reinforcement learning, on the other hand, is inspired by behaviorist psychology (see also https://en.wikipedia.org/wiki/Behaviorism), which is is typically concerned with how a software agent performs an action in a new *environment* by maximizing the `reward` function. Dynamic programming and intelligent agent are two examples of reinforcement learning.

Typical machine learning applications can be classified into scientific knowledge discovery and more commercial applications, ranging from Robotic or **Human Computer Interaction** (**HCI**) to anti-spam filtering and recommender systems.

Machine learning in statistics and data analytics

Machine learning reconnoitres the study and construction of algorithms (see also https://en.wikipedia.org/wiki/Algorithm) that can learn (see also https://en.wikipedia.org/wiki/Learning) from the heuristics and make meaningful predictions on data. However, in order to make data-driven predictions or decisions, such algorithms operate by building a model (see also https://en.wikipedia.org/wiki/Mathematical_model) from training datasets, quicker than following a stringently static program or instructions. Machine learning is also closely related and often overlaps with the nature of computational statistics. Computational statistics is, on the other hand, an applied field of statistics that focuses on making predictions through a computerised approach. In addition, it has strong stalemates to mathematical optimisation, which delivers methods and computing tasks along with theory and application domains. The tasks that are not feasible in mathematics due to the demands for a strong background knowledge of

mathematics, machine learning suits best and can be applied as the alternative to that.

Within the field of data analytics, on the other hand, machine learning is a method used to devise complex models and algorithms that advance themselves towards prediction for a future outcome. These analytical models allow researchers, data scientists, engineers, and analysts to produce reliable, repeatable, and reproducible results and mine hidden insights through learning from past relationships (heuristics) and trends in the data. Again we will refer to a famous definition from Prof. Tom, where he explained what learning really means from the computer science perspective in the literature (*Tom M. Mitchell, The Discipline of Machine Learning, CMU-ML-06-108, July 2006*, http://www.cs.cmu.edu/~tom/pubs/MachineLearning.pdf):

> *A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.*

Therefore, we can conclude that a computer program or machines can:

- Learn from data and histories
- Can be improved with experience
- Interactively enhance a model that can be used to predict the outcomes of questions

Furthermore, the following diagram helps us to understand the whole process of machine learning:

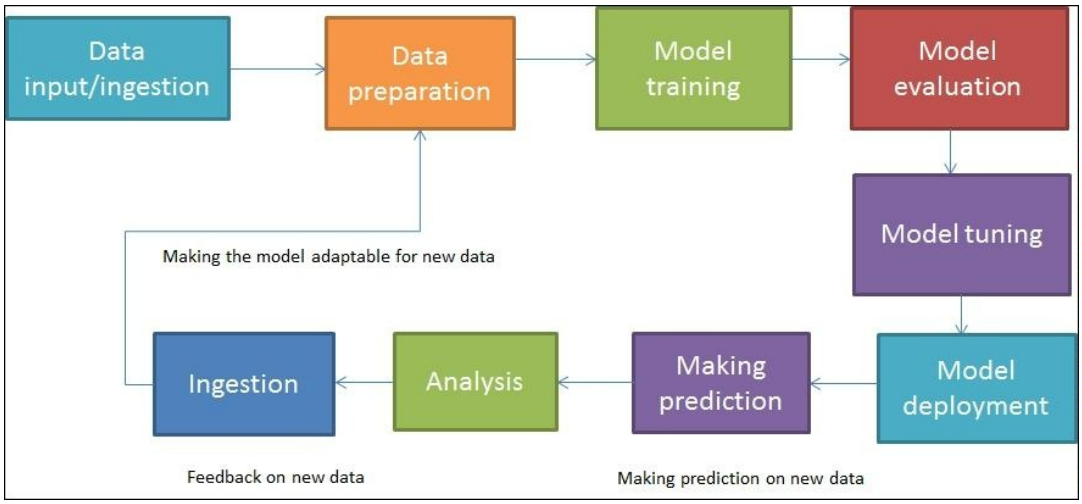


Figure 1: Machine learning at a glance.

Typical machine learning workflow

A typical machine learning application involving several steps from input, processing to output that form a scientific workflow is shown in Figure 2. The following steps are involved in typical machine learning applications:

1. Load the sample data.
2. Parse the data into the input format for the algorithm.
3. Pre-process the data and handle the missing values.
4. Split the data into two sets, one for building the model (training dataset) and one for testing the model (test dataset or validation dataset).
5. Run the algorithm to build or train your ML model.
6. Make predictions with the training data and observe the results.
7. Test and evaluate the model with the test data or alternatively validate the model with some cross-validator technique using the third dataset, called the validation dataset.
8. Tune the model for better performance and accuracy.
9. Scale-up the model so that it can handle massive datasets in the future.
10. Deploy the ML model in commercialization:

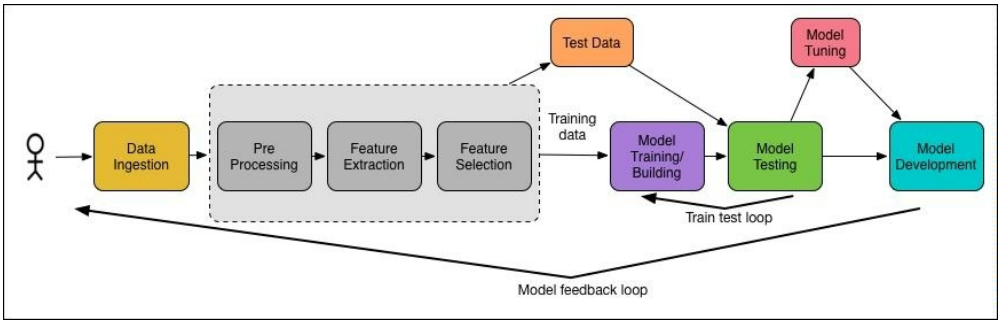


Figure 2: Machine learning workflow.

Often the machine learning algorithms have some ways to handle the skewness in the datasets; that skewness can sometimes be immensely skewed though. In step 4, the experimental dataset is split often into a training set and test sets randomly, which is called sampling. The training dataset is used to train the model, whereas the test dataset is used to evaluate the performance of the best model at the very end. The better practice is to use the training dataset as much as you can to increase the generalization performance. On the other side, it is recommended to use the test dataset only once to avoid the overfitting and underfitting problem while computing the prediction error and the related metrics.

Tip

Overfitting is a statistical property by which random error and noise is described apart from the normal and underlying relationships. It mostly

occurs when there are too many hyperparameters relative to the number of observations or features. Under fitting on the other hand refers to a model that can neither model the training data nor generalize to new data towards the model evaluation or adaptability.

However, these steps consist of several techniques and we will discuss those in Chapter 5, *Supervised and Unsupervised Learning by Examples* in detail. Step 9 and 10 are usually considered as advanced steps, and they consequently will be discussed in later chapters.

Machine learning tasks

Machine learning tasks or machine learning processes are typically classified into three broad categories, depending on the nature of the learning feedback available to a learning system. Supervised learning, unsupervised learning, and reinforcement learning; these three kinds of machine learning tasks are shown in *Figure 3*, and will be discussed in this section:

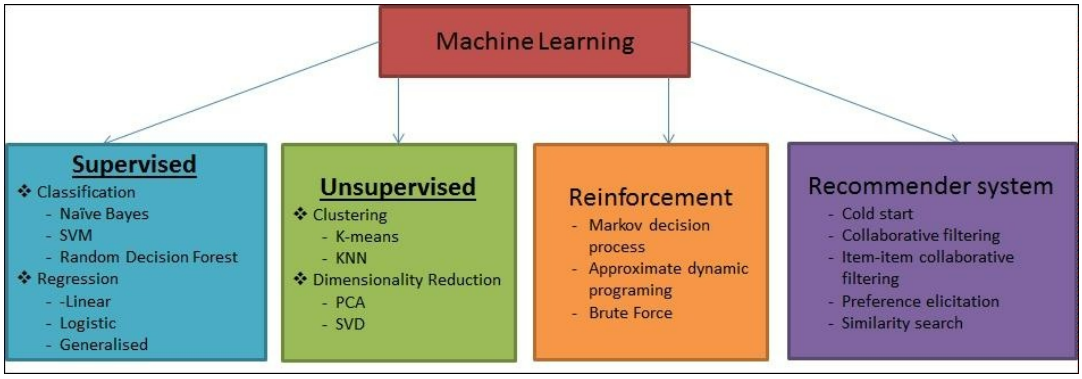


Figure 3: Machine learning tasks.

Supervised learning

A **supervised learning** application makes predictions based on a set of examples, and the goal is to learn general rules that map inputs to outputs aligning with the real world. For example, a dataset for spam filtering usually contains spam messages as well as non-spam messages. Therefore, we could know which messages in a training set are spams or non-spams. Nevertheless, we might have the opportunity to use this information to train our model in order to classify new and unseen messages. Figure 4 shows the schematic diagram of the supervised learning.

In other words, the dataset for training the ML model in this case is labeled with the value of interest and a supervised learning algorithm looks for patterns in those value labels. After the algorithm has found the required patterns, those patterns can be used to make predictions for unlabeled test data. This is the most popular and useful type of machine learning tasks, which is not an exception for Spark as well, where most of the algorithms are a supervised learning technique:

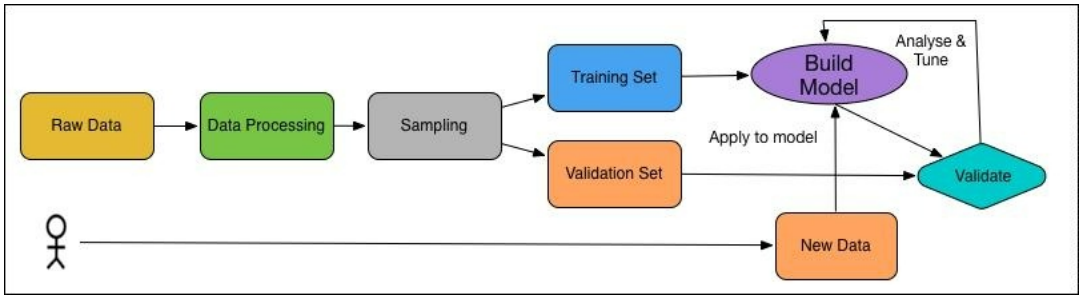


Figure 4: Supervised learning in action.

Unsupervised learning

In **unsupervised learning**, data points have no labels related or in other words, the correct classes of the training dataset in unsupervised learning are unknown, as shown in *Figure 5*. As a result, classes have to be inferred from the unstructured datasets, which implies that the goal of an unsupervised learning algorithm is to pre-process the data in some structured ways by describing its structure.

To overcome this obstacle in unsupervised learning, clustering techniques are used typically to group the unlabeled samples based on certain similarity measures, mining hidden patterns towards feature learning. More technically, we can write down a generative model, and then tell the data to find parameters that explain the data to us. Now what will happen next if we are not satisfied with the possibility of this elucidation? The answer is that we should tell the data to do it again until we are using some efficient algorithms or techniques.

Now a new question may arise in your mind, why do we have to put labels on the data? Or cannot we just appreciate the data in its current order recognizing that each datum is unique and pre snowflake? In other words, with a little supervision, our data can grow up to be whatever it wants to be! So why should the unlabeled data be taken into consideration too?

Well, there are some deeper issues regarding this. For example, most of the variation in the data comes from phenomena that are irrelevant to our desired labeling scheme. A more realistic example would be how Gmail classifies e-mails as spam and ham using the supervised learning technique, where the data might use its parameters to explain its semantics, when all we care about is its syntactic properties:

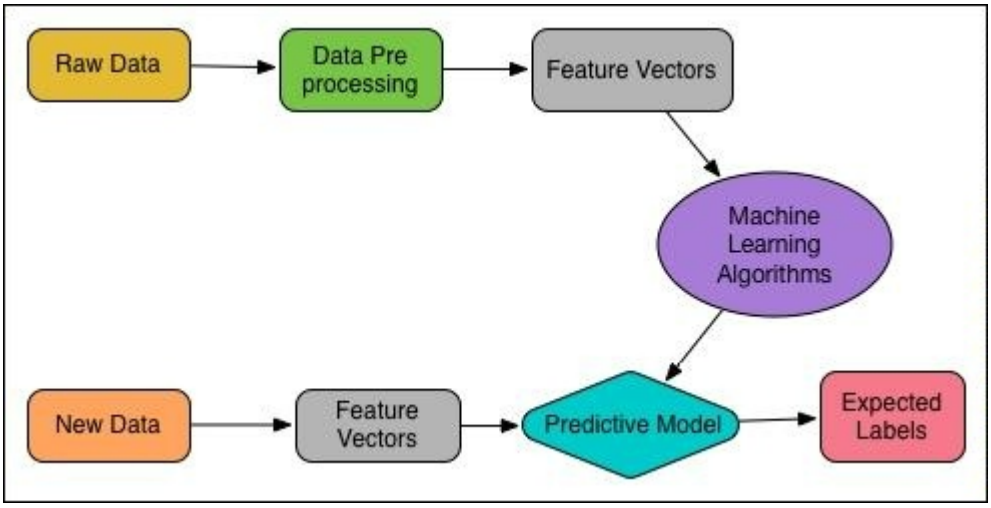


Figure 5: Unsupervised learning.

Reinforcement learning

Reinforcement learning is the technique where the model itself learns from a series of actions or behaviors. Complexity of datasets or sample complexity is very important in the reinforcement learning needed for the algorithms to learn a target function successfully. Moreover, in response to each data point for achieving the ultimate goal, maximization of the reward function should be ensured while interacting with an external environment, as demonstrated in *Figure 6*. To make the maximization easier, the reward function can either be exploited by penalizing the bad actions or rewarding for the good actions.

In order to achieve the highest reward, the algorithm should be modified with a strategy that also allows the machine or software agent to learn its behavior periodically. These behaviors can be learned once and for all, or the machine learning model can keep adapting as times passes:

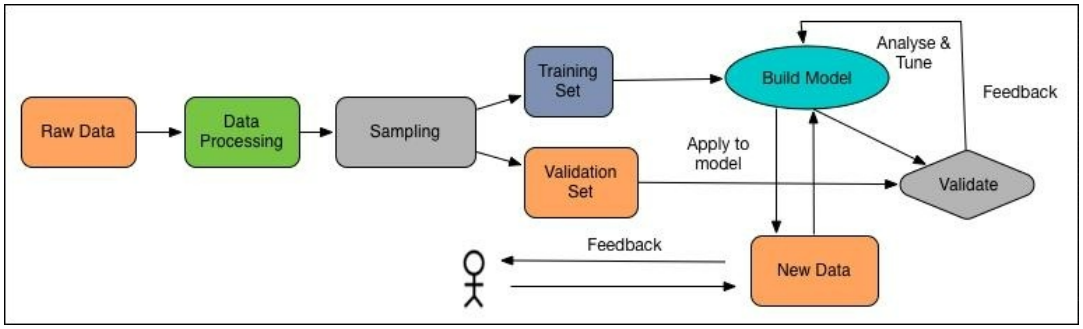


Figure 6: Reinforcement learning.

For example, reinforcement learning is common in robotics; the algorithm must choose the robot's next action based on a set of sensor readings. It is also a natural fit for **Internet of Things** (**IoT**) applications, where a computer program interacts with a dynamic environment in which it must

perform a certain goal, without an explicit mentor. Another example is the game **Flappy Bird**, which has been trained to play itself.

Recommender system

A recommender system is an emerging application, which is a subclass of information filtering system use for making a prediction of the rating or preference from the users that they usually provide to an item. The concept of recommender systems has become very common in recent years and subsequently applied in different applications. The most popular ones are probably products (for examples, movies, music, books, research articles, news, search queries, social tags, and so on). Recommender systems can be typed into four categories typically:

- Collaborative filtering system, where accumulation of a consumer's preferences and recommendations to other users is based on likeness in behavioral patterns.
- Content-based systems, where the supervised machine learning is used to persuade a classifier to distinguish between interesting and uninteresting items for the users.
- Hybrid recommender systems is a recent research and hybrid approach (that is, combining collaborative filtering and content-based filtering). Netflix is a good example of such a recommendation system that uses **Restricted Boltzmann Machines** (**RBM**) and a form of the Matrix Factorization algorithm for large movie databases, such as IMDb. This recommendation, which simply recommends movies or dramas or streaming by comparing the watching and searching habits of similar users, is called rating prediction.
- Knowledge-based systems, where knowledge about users and products is used to reason what fulfills the user's requirements, using the perception tree, decision support systems, and case-based reasoning.

Semi-supervised learning

Between supervised and unsupervised learning, there is a small place for **semi-supervised learning**; where the ML model usually receives an incomplete training signal. More statistically, the ML model receives a training set with some of the target outputs missing. The semi-supervised learning is more or less assumption-based and often uses three kinds of assumption algorithms as the learning algorithm for the unlabeled datasets. The following assumptions are used: smoothness, cluster, and manifold assumption.

In other words, semi-supervised learning can furthermore be denoted as a **weakly supervised** or **bootstrapping** technique for using the hidden wealth of unlabeled examples to enhance the learning from a small amount

of labeled data. Emerging examples include *semi-supervised expectation minimization, and concept learning in human cognition and transitive SVMs*.

Practical machine learning problems

What does machine learning really mean? We already saw some convincing definitions of this term as well as the meaning of the term *learning* at the very beginning of this chapter. However, the reality is machine learning itself is defined by the problems to be resolved. In this section, we will first emphasize the machine learning classes and then we will list some well-known and popularly-used examples of real world machine learning problems. The typical classes include classification, clustering, rule extraction, and regression, which will all be discussed.

In addition, we will also discuss those problems based on the main taxonomy of standard machine learning problems. This is important, since knowing the type of problems we could face allows us to think about the data we need. Another important fact is that before knowing some practical machine learning problems, you might face difficulties in having an idea about developing your machine learning applications. In other words, to know the problem we need to know the data in the very first place. Therefore, the types of algorithm and their optimality to be addressed will be discussed throughout this chapter; data manipulation, however, will be discussed to dig-down the problems in Chapter 3, *Understanding the Problem by Understanding the Data*.

Machine learning classes

The problem classes we mentioned above are standards for most of the problems we refer to in everyday life while doing and applying machine learning techniques. However, knowing only the ML classes is not enough we also need to know what type of problems machines are learning, since you will find many problems that are simply problem solving that does not help a ML model or agent to learn at all.

When you think a problem is a machine learning problem, more technically, you are thinking of a decision problem that needs to be modeled from data that could be termed as a machine learning problem. In other words, as a data scientist or human expert, if you have enough time to answer a particular question by knowing the available dataset, you can more or less apply a suitable machine learning problem. Therefore, we can assume that a solvable problem using some ML algorithms would have mainly two parts - the data itself, which could be used to point to specific observations of the problem, and secondly the quantitative measurement of the quality of an available solution. Once you have succeeded in identifing a problem as an ML problem, you would probably be able to think about what types of problems you could formulate with it easily, or the type of aftermath your

client will be asking for, or what sorts of requirements are to be satisfied. As already stated in the above section, the more frequently used machine learning classes are: classification, clustering, regression, and rule extraction. We will now provide a short overview of each class.

Classification and clustering

If the experimental dataset is labeled, it means a class has been assigned to it already. For instance, spam/non-spam during spam e-mail detection or fraud/non-fraud during credit card fraud identification. However, if the dataset based on which the fundamental decision will be made or modeled is unlabeled, new labels need to be made manually or algorithmically. This might be difficult, and can be thought of a judgment problem. On the contrary, sculpting the differences or resemblances between several groups might be computationally harder.

Clustering, on the other hand, handles the data that is not labeled or un-labeled. However, it still can be divided into groups based on similarity and other measures of natural structure in the data you have. Organizing pictures from a digital album by faces only without names could be an example, where the human users like us have to assign names to groups manually. Again, the same computational complexity might arise to label multiple image files manually; we will provide some examples in later chapters of how Spark provides several APIs to solve these issues.

Rule extraction and regression

From the given dataset, propositional rules can be generated by means of antecedent and consequent in the *if...then* style that defines the behavior of a machine learning agent. This type of rule generation technique is commonly referred to as *rule extraction*. You might be wondering if such rules might exist, however, they are typically not directed. That means the methods used to discover statistically meaningful or statistically significant relationships between attributes in your data.

An example of rule extraction is the mining association rules between items from business oriented transactional databases. Non-technically, a practical example could be the discovery of the relationship or association between the purchase of beer and diapers, which is illustrative of the desire and opportunity for the customers. However, some situation, might arise where some predictions out of the rules or data are not necessarily involved directly.

Now let's talk about the regression where the data is labeled with a real value. To be more exact, some floating point value rather than having labels in the data. The easiest way to understand an example would be time series data similar to the price of a stock or currency that changes over

time. In these types of data, the regression task is to make a prediction for new and unpredicted data by some regression modeling techniques.

Most widely used machine learning problems

You will find an extensive amount of examples of the use of machine learning related problems in daily life, since they solve the difficult parts of the available problems that are widely used techniques or algorithms. We often use many desktop or web-based applications that solve your problems out of the data even without knowing that what underlying techniques have been used. You will be wondered to know that many of them actually use widely used machine learning algorithms to make your life easier. There are many machine learning problems around. Here we will mention some example problems that really represent what machine learning is all about:

- **Spam detection or spam filtering**: Given some e-mails in an inbox, the task is to identify those e-mails that are spam and those that are non-spam (often called ham) e-mail messages. Now the challenging part is to develop an ML application that can be applied so that it can identify only the non-spam e-mails to stay in the inbox. and move the spam emails to the corresponding spam folder or delete them permanently from the email account. A typical example could be what you may do while using Gmail manually, but if you have an ML application, that application will do it automatically.
- **Anomaly detection or outlier detection**: The anomaly detection deals with the identification of items, events, or observations that are unexpected or non-confirming to the expected patterns in a dataset; in other words, the identification of suspect patterns. The most common example is network anomaly detection using some machine learning applications. Now the challenging task is to develop an ML application that can be applied successfully to simply identify the unusual data points from the data propagating across the network.
- **Credit card fraud detection**: Credit card fraud is very common nowadays. Stealing credit card related information from online shopping and using it in an illegal way happens in many countries. Suppose you have a transactional database for a customer for a particular month. Now the challenging task is to develop an ML application to identify those transactions that were made by the customer themselves and those done by others illegally.
- **Voice recognition**: Recognizing a voice and converting it into a corresponding text command and later performing some actions, as an intelligent agent does. The most widely used applications include Apple Siri, Samsung S-Voice, Amazon's Echo (consumer space), and Microsoft Cortana (especially because Cortana has SDKs for extensibility and integration, and so on). Another example would be locking or unlocking your smartphone by using the recognised voice.

- **Digit/character recognition**: Suppose you have a handwritten zip code or address or message on/inside an envelope, now the task of digit/character recognition is to identify and classify the digits or characters for each handwritten character that is made by different people. An efficient ML application could help in this regard to read and understand handwritten zip codes or characters and sort the contents of the envelope by the geographic region, or more technically, by the image segmentations.
- **Internet of Things**: Large-scale sensor data analytics for prediction and classification from real-time streamed data. For example, smart living room monitoring including water level checking, room temperature checking, home appliances controlling, and so on.
- **Gaming analytics**: Analytics for sports, games, and console-based gaming profiles in order to predict upsell and target in-app purchases and modifications.
- **Face detection**: Given a digital photo album of hundreds or thousands of photographs, the task is to identify those photos that resemble a given person. An efficient ML application, in this case, could help to organise photos by person.
- **Product recommendation**: Provided a purchase history of a customer along with a large inventory of products, the target is to identify those products that the customer will likely be interested in purchasing with an ML system. Business and tech giants such as Amazon, Facebook, and Google Plus have this recommended feature for the users.
- **Stock trading**: Given the current and historical prices for a stock market, predict whether stock should be bought or sold in order to profit with the help of an ML system.

The following are some examples of machine learning that are emerging and the demands of current research:

- **Privacy preserving data mining**: Mining customer's purchase rules from the maximal frequent pattern and association rules from business oriented retail databases to increase purchases in the future
- **Author name disambiguation**: Disambiguation performance is evaluated with manual verification of random samples of pairs from clustering results from a list of authors from a set of given publications
- **Recommendation systems**: Recommender system based on click stream data using association rule mining
- **Text mining**: Plagiarism checking from a given text corpus for example
- **Sentiment analysis**: A lot of decisions these days are being made by business and tech companies based on the opinion of others, and it will be a good place to innovate machine learning
- **Speech understanding**: Given an utterance from a user, the target is to identify the specific request made by the user. A model of this problem would allow a program to understand and make an attempt to

fulfill that request. For example, iPhone with Siri and Samsung Voice Recorder in meeting mode have this feature implemented

Some of these problems are the hardest problems in artificial intelligence, natural language processing, and computer vision that can be addressed and solved using ML algorithms. Similarly, we will try to develop some ML applications emphasizing these problems in upcoming chapters.

Large scale machine learning APIs in Spark

In this section, we will describe two key concepts introduced by the Spark machine learning libraries (Spark MLlib and Spark ML) and the most widely used implemented algorithms that align with the supervised and unsupervised learning techniques we discussed in the above sections.

Spark machine learning libraries

As already stated, in the pre-Spark era, big data modelers typically used to build their ML models using statistical languages such as R, STATA, and SAS. Then the data engineers used to re-implement the same model in Java, for example, to deploy on Hadoop.

However, this kind of workflow lacks efficiency, scalability, throughput, and accuracy as well as extended execution time.

Using Spark, the same ML model can be re-built, adopted, and deployed, making the whole workflow much more efficient, robust, and faster, which allows you to provide hands-on insight to increase the performance. The Spark machine learning libraries are divided into two packages: Spark MLlib (`spark.mllib`) and Spark ML (`spark.ml`).

Spark MLlib

MLlib is Spark's scalable machine learning library, which is the extension of the Spark Core API that provides a library of easy to use machine learning algorithms. Algorithms are implemented and written in Java, Scala, and Python. Spark provides support for local vectors and matrix data types stored on a single machine, as well as distributed matrices backed by one or multiple RDDs:

Spark MLlib		
ML tasks	**Discrete**	**Continuous**
Supervised	Classification: Logistic regression	Regression: Linear regression

	and regularized variants Linear SVM Naïve Bayes Decision trees Random forests Gradient-boosted trees	 and regularized variants Linear least squares Lasso and ridge regression Isotonic regression
Unsupervised	Clustering: K-means Gaussian matrix **Power iteration clustering** (**PIC**) **Latent Dirichlet Allocation** (**LDA**) Bisecting K-means Streaming K-means	Dimensionality reduction, matrix factorization: Principal components analysis Singular value decomposition Alternate least square
Reinforcement	N/A	N/A
Recommender systems	Collaborative filtering: Netflix recommendation	N/A

Table 1: Spark MLlib at a glance.

- **Legend**: Continuous: making predictions about continuous variables, for example, prediction of the maximum temperature for the upcoming days
- **Discrete**: Assigning discrete class labels to particular observations as outcomes of a prediction, for example, in weather forecasting it could be the prediction of a sunny, rainy, or snowy day

The beauty of Spark MLlib is numerous. For example, the algorithms implemented using Scala, Java, and Python are highly-scalable and leverage Spark's ability to work with a massive amount of data. They are fast towards designed for parallel computing with in-memory based operation, which is 100 times faster compared to MapReduce data processing (they also support disk-based operation that is 10 times faster than what MapReduce has as normal data processing) using Dataset, DataFrame, or **Directed Acyclic Graph** (**DAG**)-based RDD APIs.

They are also diverse, since they cover common machine learning algorithms for regression analysis, classification, clustering, recommender systems, text analytics, frequent pattern mining, and they obviously cover all the steps required to build scalable machine learning applications.

Spark ML

Spark ML adds a new set of machine learning APIs to let users quickly assemble and configure practical machine learning pipelines on top of Datasets. Spark ML targets to offer a uniform set of high-level APIs built on top of DataFrames rather than RDDs that help users create and tune practical machine learning pipelines. Spark ML API standardizes machine learning algorithms to make the learning tasks easier to combine multiple algorithms into a single pipeline or data workflow for data scientists.

Spark ML uses the concept of DataFrame (although it's obsolete in Java but still the main programming interface in Python and R), which is introduced in the Spark 1.3.0 release from Spark SQL as machine learning Datasets. The Datasets hold diverse data types such as columns storing text, feature vectors, and true labels for the data. In addition to this, Spark ML also uses the transformer to transform one DataFrame into another or vice-versa, where the concept of the estimator is used to fit on a DataFrame to produce a new transformer. The pipeline API, on the other hand, can restrain multiple transformers and estimators together to specify an ML data-workflow. The concept of the parameter was introduced to specify all the transformers and estimators to share a common API under an umbrella during the development of an ML application:

Spark ML		
ML tasks	**Discrete**	**Continuous**
Supervised	Classification:	Regression:

	Logistic regression Decision tree classifier Random forest classifier Gradient-boosted tree classifier Multilayer perception classifier One-vs-Rest classifier	Linear regression Decision tree regression Random forest regression Gradient-boosted tree regression Survival regression
Unsupervised	Clustering: K-means **Latent Dirichlet allocation** (**LDA**)	Tree Ensembles: Random forests Gradient-boosted Trees
Reinforcement	N/A	N/A
Recommender systems	N/A	N/A

Table 2: Spark ML at a glance (legend same as Table 1).

As shown in table 2, Spark ML also provides several classifications, regression, decision trees, and tree ensembles as well as a clustering algorithm implemented for developing ML pipelines on top of DataFrames. The optimization algorithm under active implementation is called **Orthant-Wise Limited-memory QuasiNewton** (**OWL-QN**), which is also an advanced algorithm that is an extension of L-BFGS that can effectively handle L1 regularization and elastic net (see also at Spark ML Advanced topic, https://spark.apache.org/docs/latest/ml-advanced.html).

Important notes for practitioners

However, currently only Pearson's and Spearman's correlation are supported and more are to be added in future Spark releases. Unlike the other statistical functions, stratified sampling is also supported by Spark and

it can be performed on RDDs as key-value pairs; however, some functionalities are yet to be added to Python developers. Currently there are no reinforcement learning algorithm modules in Spark Machine Learning libraries (please refer to *Table 1* and *Table 2*). The current implementation of Spark MLlib provides a parallel implementation of FP-growth for mining frequent patterns and the association rules. However, you will have to customize the algorithm for mining maximal frequent patterns accordingly. We will provide a scalable ML application for mining privacy preserving maximal frequent pattern in upcoming chapters.

Another fact is that the current implementation of the collaborative based recommendation system in Spark does not support the use of real time stream data, however, in later chapters we will try to show a practical recommender system based on click stream data using association rule mining (see Mitchell, Tom M. *The Discipline of Machine Learning*, 2006, http://www.cs.cmu.edu/. CMU. Web. Dec. 2014). However, some algorithms are not available or are yet to be added to Spark ML, most notably dimensionality reduction is such an example.

However, developers can seamlessly combine the implementation of these techniques found in Spark MLlib with the rest of the algorithms found in Spark ML as hybrid or interoperable ML applications. Spark's neural networks and perception are brain-inspired learning algorithms covering multiclass, two-class, and regression problems that are not yet implemented in Spark ML APIs.

Practical machine learning best practices

In this section, we will describe some good machine learning practices that need to be followed before developing a machine learning application of particular interest, as described in *Figure 7*:

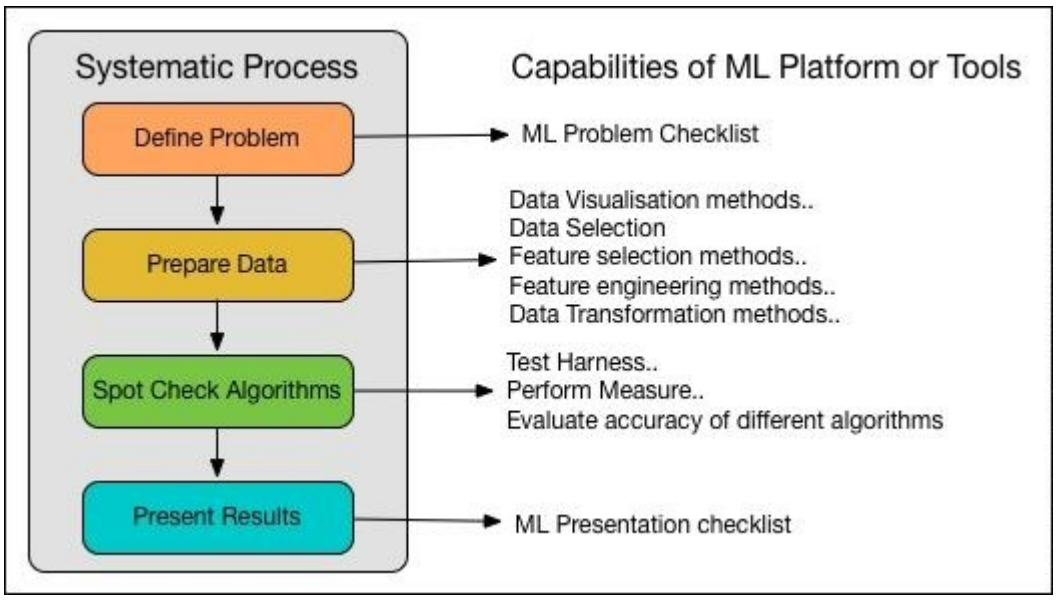


Figure 7: Machine learning systematic process.

A scalable and accurate ML application demand for following a systematic approach to its development from problem definition to presenting results can be summarized into four steps: problem definition and formulation, data preparation, finding suitable algorithms for machine learning, and finally, presenting the results after the machine learning model deployment. Well, these steps can be depicted as shown in *Figure 6*.

Best practice before developing an ML application

The learning of a machine learning system can be formulated as the sum of representation, evaluation, and optimisation. In other words, according to Pedro D et al. (Pedro Domingos, *A Few Useful Things to Know about Machine Learning*, https://homes.cs.washington.edu/~pedrod/papers/cacm12.pdf):

Learning = Representation + Evaluation + Optimization

Taking this formulation into consideration, we will provide some recommendations for practitioners before getting into ML application development.

Good machine learning and data science worth huge

So what do we need for an effective machine learning applications development? We actually need four arsenals before we start developing an ML application; including:

- The data primitives (or the experimental data to be more frank).
- A pipeline synthesis tool (to understand the data and control flow during the machine learning steps).
- An effective and robust error analysis tools.
- A verification or validation tool (to verify or validate the prediction accuracy or performance of the ML model). However, most importantly, without some strong theoretical basement with good data science that is worth a huge amount, the whole process will be in vain. In fact, many data scientists and machine learning experts often quote something like this statement: *if you can pose your problem as a simple optimization problem then you is almost done* (see *Data Analytics & R*, http://advanceddataanalytics.net/2015/01/31/condensed-news-7/).

That means before you start your machine learning voyage, if you can identify if your problem is a machine learning problem, you will be able to find some suitable algorithms to develop your ML application altogether. Of course, in practice, most machine learning applications can't be changed into simple optimization problems. Therefore, it's the duty of a data scientist like you to manage and maintain complex datasets. After that, you will have to handle other issues such as the analytical problems that evolve when engineering the machine learning pipeline to tackle those issues we mentioned earlier.

Therefore, the best practice is to use Spark MLlib, Spark ML, GraphX, and Spark Core APIs along with the best practice data science heuristics for developing your machine learning applications together. Now you might think of getting benefits out of it; yes, the benefits are obvious, and they are as follows:

- Built-in distributed algorithms
- In-memory and disk-based data computation and processing
- In-memory capabilities for iterative workloads
- Algorithmic accuracy and performance
- Faster data cleaning, feature engineering and feature selection, training, and testing
- Real-time visualization of the predictive results

- Tuning towards better performance
- Adaptability for new datasets
- Scalability with the increasing datasets

Best practice – feature engineering and algorithmic performance

In best practice, feature engineering should be considered as one of the most important parts of machine learning. The thing is to find a better representation of features out of the experimental dataset non-technically. In parallel to this, which learning algorithms or techniques are to be used are also important. Parameter tuning, of course in addition, however, the final choice is more about experimentation through the ML model you will be developing.

In practice, however, it is trivial to grasp the naive performance baseline by means of an **out-of-the-box** method (also referred to as functionality or **OOTB** in short, which is a feature of a product of interest that works straight away after installing or configuring) and good data pre-processing. Therefore, you might be doing it continually in order to know where the baseline is and whether this performance is of a satisfactory level or good enough for your requirements.

Once you've trained all of your out-of-the-box methods, it's always recommended and is a good idea to try bagging them together. Moreover, in order to solve the ML problems, very often you might need to know the reality that computationally hard problems (shown in section 2, for example) need either domain-specific knowledge or lots of digging down in the data or both. Consequently, the combination of a widely accepted feature engineering technique and domain-specific knowledge would help your ML algorithm/application/system to solve prediction related problems.

In a nutshell, if you have the required dataset and a robust algorithm that can take the advantages of the dataset by learning the complex features, it's almost guaranteed that you will be successful. Furthermore, sometimes domain experts might be wrong in selecting the good features; therefore, incorporation of multiple domain experts (problem domain expert), more well-structured data, and ML expertise is always helpful.

Last but not least, sometimes it is recommended from our side to consider the error rate rather than only the accuracy. For example, suppose an ML system with 99% accuracy and 50% errors is worse than the one with 90% accuracy but 25% errors, for example.

Beware of overfitting and underfitting

A common mistake often made by novice data scientists is subject to the overfitting issue that might evolve while building your ML model by hearing

without generalizing. More technically, if you evaluate your model on the training data instead of test or validated data, you probably won't be able to articulate whether your model is overfitting or not. The common symptoms are:

- Predictive accuracy of the data used for training can be over accurate (that is, sometimes even 100%)
- And the model might show a little better compared to the random prediction for new data

Sometimes the ML model itself becomes under-fit for a particular tuning or data point, which means the model has become too simplistic. Our recommendation (like others as well we believe) is as follows:

- Split the dataset into two sets to detect overfitting situations, the first one being for training and model selection, called the training set; the second one is the test set for evaluating the model stated in place of the ML workflow section
- Alternatively, you also could void the overfitting by consuming simpler models (for example, linear classifiers in preference to Gaussian kernel SVM) or by swelling the regularisation parameters of your ML model (if available)
- Tune the model with a correct data value of parameters to avoid both overfitting as well as underfitting

Hastie et al. (Hastie Trevor, Tibshirani Robert, Friedman Jerome, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, Second Edition, 2009) on the other hand, have recommended splitting the large-scale dataset into three sets: Training set (50%), Validation set (25%), and Test set (25%) (roughly). They also suggested building the model using the training set and calculating the prediction errors using the validation set. The test set was recommended to be used to assess the generalization error of the final model.

If the amount of labeled data available during the supervised learning is smaller, it is not recommended to split the datasets. In that case, use cross-validation or Train split techniques (this will be discussed in Chapter 7, *Tuning Machine Learning Models,* with several examples). More specifically, divide the data set into 10 parts of (roughly) equal size, after that for each of these ten parts, train the classifier iteratively and use the 10th part to test the model.

Stay tuned and combining Spark MLlib with Spark ML

The first step of the pipeline designing is to create the building blocks (as a directed or undirected graph consisting of nodes and edges) and make a link between those blocks. Nevertheless, as a data scientist, you should be focused on scaling and optimizing nodes (primitives) too, so that you are

able to scale-up your application for handling large-scale datasets in the later stage to make your ML pipeline consistently perform. The pipeline process will also help you to make your model adaptive for new datasets. However, some of these primitives might be explicitly defined to particular domains and data types (for example, text, images, video, audio, and spatiotemporal).

And beyond these types of data, the primitives should also be working for the general purpose domain statistics or mathematics. The casting of your ML model in terms of these primitives will make your workflow more transparent, interpretable, accessible, and explainable. A recent example would be the ML-Matrix, which is a distributed matrix library that can be used on top of Spark:

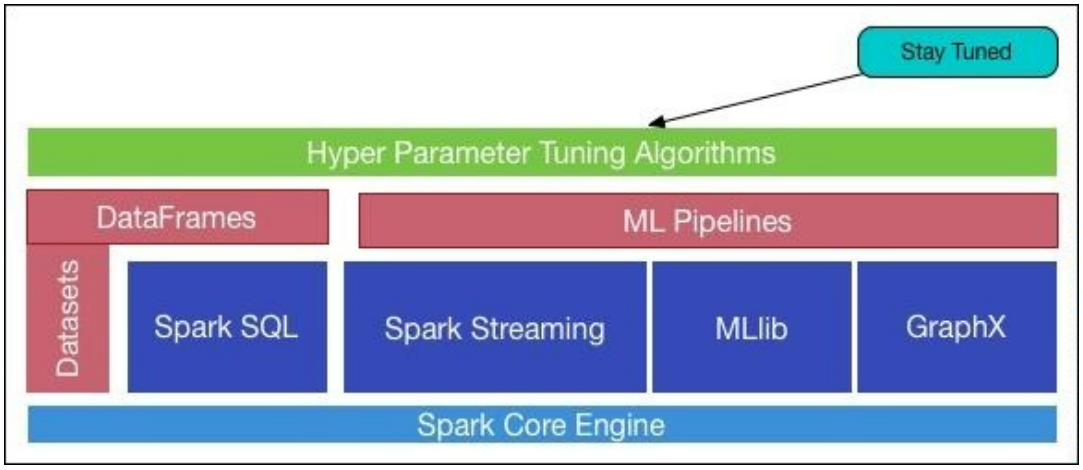


Figure 8: Stay tune and interoperate ML, MLlib, and GraphX.

As we already stated in the previous section, as a developer you can seamlessly combine the implementation techniques in Spark MLlib along with the algorithms developed in Spark ML, Spark SQL, GraphX, and Spark Streaming as hybrid or interoperable ML applications on top of RDD, DataFrame, and Datasets, as shown in Figure 8. For example, an IoT-based real-time application could be developed using a hybrid model. Therefore, the recommendation here is to stay tuned or synchronized with the latest technologies around you for the betterment of your ML application.

Making ML applications modular and simplifying pipeline synthesis

Another good and often used practice when building your ML pipeline is to make the ML system modular. Some supervised learning problems can be solved using very simple models commonly referred to as generalized linear models. However, it depends on the data you will be using and others simply don't.

Therefore, to conglomerates a series of simple linear binary classifiers, try to employ a lightweight modular architecture. This might be at the workflow stems or at the algorithms level. The advantages are obvious, since the modular architecture of your application handles massive amounts of data flow in a parallel and distributed way. Consequently, we suggest you have the three key innovative mechanisms: weighted threshold sampling, logistic calibration, and intelligent data partitioning as mentioned in the literature (for example, Yu Jin; Nick Duffield; Jeffrey Erman; Patrick Haffner; Subhabrata Sen; Zhi Li Zhang, *A Modular Machine Learning System for Flow-Level Traffic Classification in Large Networks*, ACM Transactions on Knowledge Discovery from Data, V-6, Issue-1, March 2012). The target is to achieve scalability and high-throughput while attaining a high accuracy of the predicted results from your ML application/system. While primitives can serve as building blocks, you still need some other tools that enable users to build ML pipelines.

Subsequently, workflow tools have become more common these days, and such tools exist for data engineers, data scientists, and even for business analysts such as Alteryx, RapidMiner, Alpine Data, and Dataiku. At this point, we are talking about and stressing the business analysts since at the very last phase your target customer will be a business company who will value your ML model, right? The latest release of Spark comes with Spark ML APIs for building machine learning pipelines and making a domain specific language (see https://en.wikipedia.org/wiki/Domain-specific_language) for pipelines.

Thinking of an innovative ML system

However, in order to develop the algorithms to learn the ML models continuously with the help of available data, the viewpoint behind the machine learning is to automate the creation of analytical models. Unremittingly evolving models produce increasingly positive results and reduce the need for human interaction. This enables the ML models to automatically produce reliable and repeatable predictions.

More technically, suppose you are planning to develop a recommender system using ML algorithms. So, what is the target of developing that recommender system? And what are some innovative ideas for product development in machine learning? These two are typical questions that should be considered before you start developing your ML application or system. Consistent innovation might be challenging, especially when stirring advancing with new ideas, it can also be tough to comprehend where the greatest benefit lies. Machine learning can provision innovation from end to end of a variety of paths, such as determining weaknesses with current products, predictive analysis, or identifying previously concealed patterns.

As a result, you will have to think of large-scale computing to train your ML model offline, and later on your recommender system has to be able to work as a conventional search engine analysis for online recommendations. Thus, your ML application will be valued by a business company if your system:

- Can forecast buying items using your machine learning application
- Can do product analysis
- Can work as an emerging trend in production

Thinking and becoming smarter about Big Data complexities

As shown in Figure 9, new business models are the unavoidable extension of the available data utilisation, so consideration of big data and its business values can make the business analyst's job, life and thinking smarter, which results in your targeted company delivering value to customers. In addition to this, you will also have to investigate (analyze to be more exact) rival or better companies.

Now the question is, how do you collect and use enterprise data? Big data is not only about the size (volume), it is also related to its velocity, veracity, variety, and value. For these types of complexities, for example, velocity can be addressed using Spark Streaming since streaming-based data is also big data that needs a real-time analytical approach. Other parameters such as volume and variety can be handled using Spark Core and Spark MLlib/ML towards big data processing.

Well, you will have to manage the data by hook or by crook. If you are able to manage the data, the insights from the data can really shake up the way businesses operate with the useful features of big data:

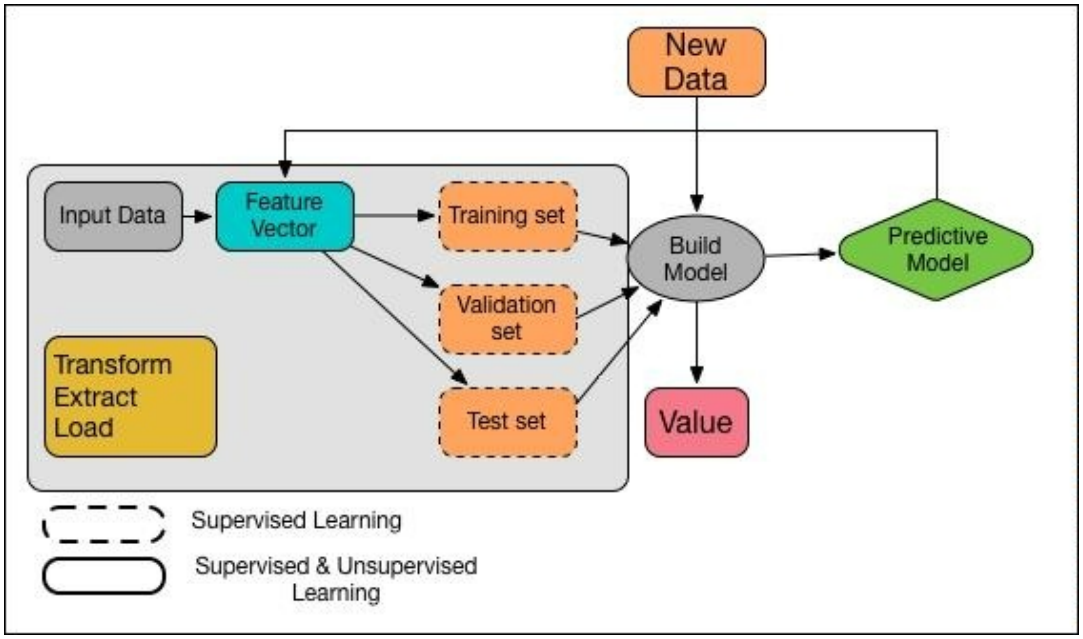


Figure 9: Machine learning in Big Data best practice.

At this point, data alone is not enough (see Pedro Domingos, *A Few Useful Things to Know about Machine Learning,* https://homes.cs.washington.edu/~pedrod/papers/cacm12.pdf), but extracting meaningful features from the data and putting semantics of data into the model is more important. This is like what most of the tech giants such as LinkedIn are developing through large-scale machine learning frameworks from feature targeting for their community, which is more or less a supervised learning technique. The workflow is as follows:

- Fetch the data, extract the feature, and set the target
- Feature and target join
- Create a snapshot from the concatenated data
- Partition the snapshot into two parts: training set and test set
- From the training set, prepare the sample data by sampling techniques
- Train the model using the sampled data
- Scoring
- Evaluate the model from the previously developed persistent model, as well as the test data prepared in step 4
- If the best model is found
- Deploy the model for the target audience

So what's next? Your model also should be adaptable to large-scale dynamic data such as real-time streaming IoT data PLUS real-time feedback is also important so that your ML system can learn from the mistakes. The next sub-section discusses that.

Applying machine learning to dynamic data

The reasons are obvious, since machine learning brings concrete and dynamic aspects to IoT projects. Recently, machine learning has experienced a pep talk in popularity amongst industrial companies and they profit out of the box. As a result, all but every IT vendor are precipitously announcing IoT platforms and consulting services. But achieving financial benefits through IoT data is not an easy job. Moreover, many businesses have failed to clearly determine what areas will change with the implementation of an IoT strategy.

Considering these positive and negative issues together, your ML model should adapt to large dynamic data since the large-scale data means billions of records, large feature spaces, and low positive rates from the sparsity issue. Nevertheless, data is dynamic so consequently, the ML models have to be adaptive enough; otherwise you will have to face a bad experience or be lost in the black hole.

Best practice after developing an ML application

The typical steps that are best practice after an ML model/system has been developed are: visualization for understanding the predictive values, model validation, error and accuracy analysis, model tuning, model adapting, and scaling up for handling large-scale datasets with ease.

How to enable real-time ML visualization

Visualization provides an interactive interface to stay tune the ML model itself. Therefore, without visualizing the predictive results, it merely becomes difficult to further improve the performance of an ML application. The best practice could be something like this:

- Incorporate some third-party tools along with GraphX for your visualization for large-scale graph related data (more to be discussed in *Chapter 9*, *Advanced Machine Learning with Streaming and Graph Data*)
- For non-graph data, a call-back interface for the Spark ML algorithm to send and receive messages by incorporating other tools like Apache Kafka:
    - Algorithms decide when and what message to send
    - Algorithms don't care how the message is delivered
- A task channel to handle the message delivery service from the Spark Driver program to Spark Client or Spark cluster nodes. The task channel would be communicating using Spark Core at a lower level of abstraction:

    - It does not care about the content of the message or recipient of the message

- The message is delivered from Spark Client to the browser or visualization client:

    - We recommend using HTML5 **Server-Sent Events** (**SSE**) and HTTP Chunked Response (PUSH) together. Incorporation of Spark with this type of technology will be discussed in Chapter 10, *Configuring and Working with External Libraries*
    - Pull is possible; however, it requires a message queue

- Visualization using JavaScript frameworks such as `Plot.ly` (please refer to https://plot.ly/) and `D3.js` (please refer to https://d3js.org/)

Do some error analysis

As algorithms become more prevalent, we need better tools for building complex hitherto, robust, and stable machine learning systems. A popular distributed framework like Apache Spark takes these ideas to extremely large datasets for the wider audience. Therefore, it would be better if we could bind approximation errors and convergence rates for the layered pipelines.

Assuming we can compute error bars for nodes, the next step would be to have a mechanism for extracting error bars for these pipelines. However, in practice, when the ML model is deployed for the production, we might need tools to confirm that the pipeline will work and will not do make malfunction or stop halfway through and that it can provide some expected measure of the errors.

Keeping your ML application tuned

Devising one or two algorithms that perform solidly well on a simple problem can be considered as a good kick-off. However, sometimes you may be thirsty to get the best accuracy, by even sacrificing your valuable time and available computational resources. This would be a smarter way, and it will help you not only to squeeze out extra performance, but also to improve the results in terms of accuracy that you were receiving out of the machine learning algorithms you designed previously. In order to do that, when you tune the model and related algorithm, essentially, you must have a high confidence in the results.

Obviously, those results will be available after you specify the testing and validation. This means you should only be using those techniques that reduce the variance of the performance measure so that you can assess the algorithms that are running more smoothly.

In parallel, like most data practitioners, we also suggest you to use the cross-validation technique (also often called rotation estimation) with a reasonably high number of folds (that is, K-fold cross-validation, where a single subsample is used as the validation dataset for testing the model itself , and the remaining K-1 subsamples are used to train the data). Although the exact number of folds, or K, depends on your dataset, however, 10-fold cross-validation is commonly used, but most often the value of K remains unfixed. We will mention three strategies here that you will need to tune your machine learning model:

- **Algorithm tuning**: Makes your machine learning algorithm parameterized. After that, adjust the value of those parameters (if they have multiple parameters) to influence the outcome of the overall learning process.
- **Ensembles**: Sometimes it is good to be naïve! Therefore, in order to get improved results, keep trying to combine the outcomes from multiple machine learning methods or algorithms.
- **Extreme feature engineering**: If your data has complex and multi-dimensional structures embedded in it, ML algorithms know how to find and exploit it to make decisions.

Keeping your ML application adaptive and scale-up

As shown in Figure 10, the adaptive learning conglomerates the previous generations of rule-based, simple machine learning, and deep learning approaches to machine intelligence according to Rob Munro:

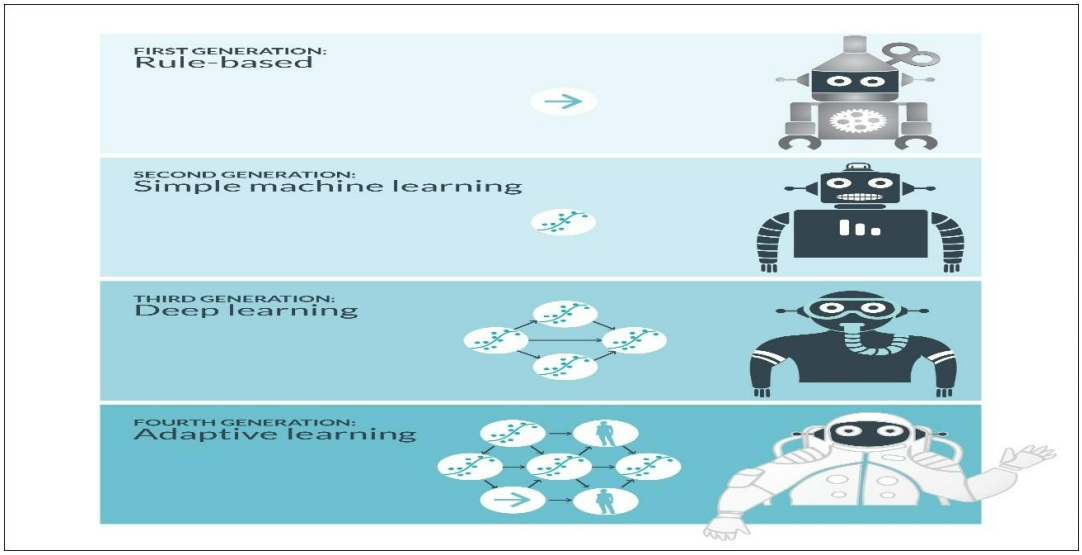


Figure 10: Four generation of machine intelligence (Figure courtesy of Rob Munro).

The fourth generation of machine learning: adaptive learning, (http://idibon.com/the-fourth-generation-of-machine-learning-adaptive-learning/#comment-175958).

Research also shows that adaptive learning is 95% accurate in predicting people's intention to purchase a car, for example (please refer to Rob Munro, *The fourth generation of machine learning: Adaptive learning*, http://idibon.com/the-fourth-generation-of-machine-learning-adaptive-learning/#comment-175958). Moreover, if your ML application is adaptive with the new environment and new data, it is expected that if enough infrastructure is provided, your ML system can be scaled-up for the increasing data loads.

Choosing the right algorithm for your application

What machine learning algorithm should I use? is a very frequently asked question for the Naive machine learning practitioners, but the answer is always i*t depends on*. More elaborately:

- It depends on the volume, quality, complexity, and the nature of the data that has to be tested/used
- It depends on external environments and parameters such as your computing system's configuration or underlying infrastructures
- It depends on what you want to do with the answer
- It depends on how the mathematical and statistical formulation of the algorithm was translated into machine instructions for the computer
- And it depends on how much time you have
- *Figure 11* provides a complete work-flow for choosing the right algorithm for your ML problem. However, note that some tricks might not work-flow depending upon data and problem types:

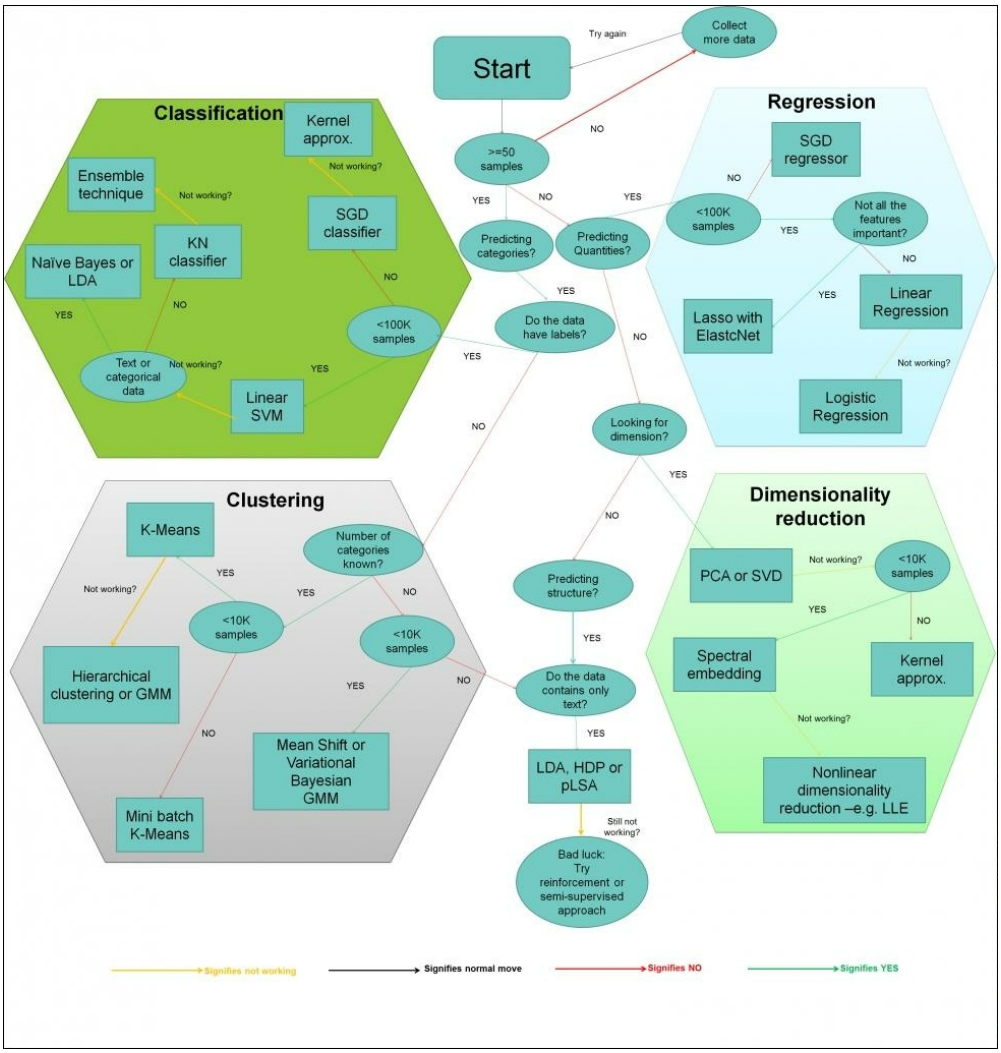


Figure 11: A work-flow for choosing the right algorithm

The reality is, even the most experienced data scientists or data engineers can't give a straight recommendation about which ML algorithm performs best before trying them all together. Most of the statements of agreement/disagreement begins with *It depends on...hmm...*Habitually, you might be contemplative if there are cheat sheets of machine learning algorithms and if so, how to use that cheat sheet. Several data scientists we talked to said that the only sure way to find the very best algorithm is to try all of them; therefore, there is no shortcut dude! Let's make it clear, suppose you do have a set of data and you want to do some clustering. Thus, technically, this could be classification or regression if your data is labeled/unlabeled or values or training set data. Now, the first concern that evolves in your mind is:

- Which factors should I consider before choosing an appropriate algorithm? Or should I just choose an algorithm randomly?

- How do I choose any data pre-processing algorithm or tools that can be applied to my data?
- What sort of feature engineering techniques should I be using to extract the useful features?
- What factors can improve the performance of my ML model?
- How can I adopt my ML application for new data types?
- Can I scale-up my ML application for large-scale datasets? And so on.

You will always expect the best answer that is much more justified and explains everything that someone should consider. In this section, we will try to answer these questions with our little machine learning knowledge.

Considerations when choosing an algorithm

The recommendation or suggestions we are providing here are for the novice data scientist with learner machine learning to expert data scientists who are trying to choose an optimal algorithm to start with the Spark ML APIs. That means, it makes some overviews and oversimplifications, but it will point you in a safe direction, believe us! Suppose you are planning to develop an ML system to answer the following question based on the rule:

- `IF` feature X has property Z `THEN` do Y

Affirmatively, there should be such rules:

- IF X `THEN` it is sensible to try Y using property Z and avoid W

However, what is sensible and what is not depends on:

- Your application and the expected complexity of the problem.
- Size of the data set (that is, how many rows/columns, how many independent cases).
- Is your dataset labeled or unlabeled?
- Type of data and the kind of measurement, since different nature of data suggests a different order or structure, right?
- And obviously in practice your experience in applying different methods efficiently and intelligently.

Moreover, if you want to have a general answer to a general problem, we recommend the Elements of Statistical Learning (Hastie Trevor, Tibshirani Robert, Friedman Jerome, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, Second Edition, 2009) for a fresh start. Nevertheless, we also recommend going with the following algorithmic properties that:

- Show excellent accuracy

- Have fast training times
- And the use of linearity

Accuracy

Getting the most accurate results from your ML application isn't always indispensable. Depending on what you want to use it for, sometimes an approximation is adequate enough. If the situation is something like this, you may be able to reduce the processing time drastically by incorporating the better-estimated methods. When you are familiar with the workflow with the Spark machine learning APIs, you will enjoy the advantage of having more approximation methods, because those approximation methods will tend to avoid the overfitting problem out of your ML model automatically.

Training time

The execution time requires finishing the data preprocessing or building the model and varies a great deal across different algorithms, the inherited complexities, and of course the robustness. The training time is often closely related to the accuracy. In addition, often you will discover that some of the algorithms you will be using are elusive to the number of data points compared to others. However, when your time is sufficient and especially when the dataset is larger, for doing all the formalities, it can get-up-and-go the choice of algorithm. Therefore, if you are concerned particularly with the time, try to sacrifice the accuracy or performance and use a simple algorithm that fulfils your minimum requirements.

Linearity

There are many machine learning algorithms developed recently that make use of linearity (also available in the Spark MLlib and Spark ML). For example, the linear classification algorithms allow classes to be separated by plotting a differentiating straight line or otherwise by the higher-dimensional equivalents of the datasets. A linear regression algorithm, on the other hand, assumes that data trends follow a simple straight line. This assumption is not naive for some machine learning problems; however, there might be some other cases where the accuracy will be down. Despite their hazards, linear algorithms are very popular for the data engineers or data scientists as the first line of the outbreak. Moreover, these algorithms also tend to be algorithmically simple and fast to train your models during the whole process.

Talking to your data when choosing an algorithm

You will find many machine learning datasets available for free here at http://machinelearningmastery.com/tour-of-real-world-machine-learning-

problems/ or at the UC Irvine Machine Learning Repository (at http://archive.ics.uci.edu/ml/). The following data properties should also be placed first:

- Number of parameters
- Number of features
- Size of the training dataset

Number of parameters

Parameters or data properties are the handholds for a data scientist like you that gets to turn when setting up an algorithm. They are numbers that affect the algorithm's performance, such as error tolerance or the number of iterations, or options between variants of how the algorithm acts. The training time and accuracy of the algorithm can sometimes be quite sensitive to getting the right settings. Typically, algorithms with a large number of parameters require trial and error to find an optimal combination.

Despite the fact that this is a great way to span the parameter space, the model building or training time increases exponentially with the increased number of parameters. This is a dilemma as well as a time-performance trade-off. The positive sides are having many parameters characteristically indicates greater flexibility of the ML algorithms. And secondly, your ML application achieves much better accuracy.

How large is your training set?

If your training set is smaller, high bias with low variance classifiers such as Naive Bayes have an advantage over low bias with high variance classifiers such as kNN. Therefore, the latter will over fit. But low bias with high variance classifiers, on the other hand, start to win out as your training set grows linearly or exponentially since they have lower asymptotic errors. This is because high bias classifiers aren't powerful enough to provide accurate models. You can also think of this as a trade-off between generative models versus discriminative model distinction.

Number of features

For certain types of experimental datasets, the number of extracted features can be very large compared to the number of data points itself. This is often the case with genomics, biomedical, or textual data. A large number of features can swamp some learning algorithms, making training time ridiculously high. Support vector machines are particularly well suited in this case for its high accuracy, nice theoretical guarantees regarding overfitting, and an appropriate kernel.

Special notes on widely used ML algorithms

In this section, we will provide some special notes for the most commonly used machine learning algorithm or techniques. The techniques we will emphasis are logistic regression, linear regression, recommender system, SVM, decision tree, random forest, Bayesian method and decision forests, decision jungles, and variants. Table 3 shows the pros and cons of some widely used algorithms including where and when to chose these algorithms.

Algorithm	**Pros**	**Cons**	**Better at**
Linear regression (LR)	Very fast and often runs in a constant time Easy to understand the modelling Less prone to overfitting and underfitting Intrinsically simple Very fast so less model building time Less prone to overfitting and underfitting Has low variance	Often unable for complex data modelling Often unable to conceptualize the nonlinear relationships without transforming the input Dataset Not suitable for complex modelling Works better with only single decision boundary Requires large sample size to achieve stable results High bias	Numerical dataset with large collection of features Widely used in biological, behavioral and social sciences to predict possible relationships among variables Works well for numerical as well as categorical variables Used in various fields, including the medical and social sciences

Decision trees (DT)	Less model building and prediction time Robust against the noise and missing values High accuracy	Interpretation is hard with large and complex trees Duplication may occur within the same sub-tree Possible issues with diagonal decision boundaries	Targeting high accurate classification Medical diagnosis and prognosis Credit risk analytics
Neural networks (NN)	Extremely powerful and robust Capable of modelling very complex relationships Can be working without knowing the underlying data	Highly overfitting and underfitting prone High training and prediction time Computationally expensive requiring significant computing power Model is not readable or reusable	Image processing Video processing Human-intelligence Robotics Deep learning
Random forest (RF)	Good for bagged trees Low variance High accuracy Can handle the overfitting problem	Not as easy to visually and interpret High training and prediction time	When dealing with multiple features which may be correlated Biomedical diagnosis and prognosis Can be applied both for classification and regression

Support vector machines (SVM)	High accuracy	Susceptible to overfitting and underfitting No numerical stability Computationally expensive requiring large computing power	Image classification Handwriting recognition
K-nearest neighbors (K-NN)	Simple and powerful Lazy training involved Can be applied for both multiclass classification and regression	High training and prediction time Need to have accurate distance function Low performance with high dimensional dataset	Low-dimensional datasets Anomaly detection like outlier detection Fault detection in semiconductor Gene expression Protein-protein interaction
K-means	Linear execution time Perform better than hierarchical clustering Excellent with hyper-spherical clusters	Repeatable and lack consistency Requires prior knowledge of K	Is not a good choice if the natural clusters occurring in the dataset are non-spherical Good for large dataset
Latent Dirichilet Allocation (LDA)	Can be applied for large-scale text datasets Can overcome the overfitting problem of pLSA	Cannot be applied with high dimensional and complex texts databases	Document classification and clustering through topic modelling from large-scale text dataset

	Can be applied for both document classification and clustering through topic modelling	Requires the specification of the number of topics Cannot find the granularity at optimum level Hierarchical Dirichlet Process (HDP) is the better choice	Can be applied in NLP and other text analytics
Naive Bayes (NB)	Computationally fast Simple to implement Works well with high dimensions Can handle missing values Is adaptable since the model can be modified with new training data without rebuilding the model	Relies on independence assumption so performs badly if the assumption does not met Relatively low accuracy	When data has lots of missing values Dependencies of features from each other are similar between features Spam filtering and classification Classifying a news article about technology, politics, or sports Text mining
Singular Value decomposition (SVD) and Principal Component Analysis (PCA)	Reflects the real intuitions about the data Allows estimation probabilities in high-dimensional data	Too expensive for many applications like Twitter and web analytics Disastrous for task with fine-grained classes	SVD is applied for low-rank matrix approximation, image processing, bioinformatics, signal processing, NLP PCA is used for interest rate derivatives

	Dramatic reduction in size of data Both are based on strong linear algebra	Need proper understanding of the linearity Often complexity is cubic Computationally slower	portfolios, neuroscience and so on Both are suitable for the dataset having high dimension and multivariate data

Table 3: Pros and cons of some widely used algorithms

Logistic regression and linear regression

Logistic regression is a powerful tool developed around the globe for its two-class and multiclass classification since it's fast as well as simple. The fact is that it uses an *S*-shaped curve instead of a straight line. making it a natural fit for partitioning data into groups. It provides linear class boundaries, so that when you use it, make sure a linear approximation is something you can survive with. Unlike the decision trees or SVMs, you also have a nice probabilistic interpretation, so you will be able to update your model to adapt for new datasets easily.

Therefore, the recommendation is, use it if you want to have a flavor of probabilistic framework or if you expect to receive more training data in the future to be incorporated into your model. As mentioned previously, linear regression fits a line, plane, or hyperplane to the dataset. It's a workhorse, simple and fast, but it may be overly simplistic for some problems.

Recommendation systems

We already talked about the accuracy and performance issues of mostly used ML algorithms and tools. However, beyond the accuracy research on recommender systems is concern about finding another environmental factor or/and parameter diversity. Therefore, a recommendation system with good accuracy and higher intra-list diversity will be the winner. As a result, your product will be precious to your target customers. It would be, however, more effective to let the users re-rate the items, rather than showing new items only. If your clients have some extra requirements that need to be fulfilled, such as privacy or security, your system has to be able to deal with the privacy related issues.

This is particularly important because customers have to provide some personal information as well, so it is recommended not to expose that sensitive information publicly.

Building user profiles using some robust techniques or algorithms such as collaborative filtering, on the other hand, could be problematic from the privacy perspective. Moreover, research in this area has found that user demographics information may influence how satisfied the other users are with recommendations (see also in Joeran Beel, Stefan Langer, Andreas Nürnberger, Marcel Genzmehr, *The Impact of Demographics (Age and Gender) and Other User Characteristics on Evaluating Recommender Systems*. In Trond Aalberg and Milena Dobreva and Christos Papatheodorou and Giannis Tsakonas and Charles Farrugia. *Proceedings of the 17th International Conference on Theory and Practice of Digital Libraries, Springer, pp. 400-404, Retrieved 1 November 2013*).

Although the serendipity is a crucial measure of how surprising the recommendations are, ultimately trust needs to be built using the recommender system. This can be made possible by explaining how it generates the recommendations, and why it recommends an item even with little demographic information, from the user.

Therefore, if the user does not trust the system at all, they will not provide any demographic information or will not re-rate the items. A SVMs, according to *Cowley et al*. (G. C. Cawley and N. L. C. Talbot, *Over-fitting in model selection and subsequent selection bias in performance evaluation, Journal of Machine Learning Research, vol. 11, pp. 2079-2107, July 2010*), there are several advantages of Support Vector Machines:

- You can tackle the problem of the over-fitting problem since SVMs provide you with a regularization parameter
- SVM use the kernel trick that helps to build the machine learning model via engineering the kernel with ease
- An SVM algorithm is developed, designed, and defined based on a convex optimization problem, therefore, there is no concept of local minima
- It is a ballpark figure to a bound on the test error rate, where there is a significant and well-studied theory that works

These promising features of SVM really would help you, and it is suggested that it should be used frequently. On the other hand, the cons are:

- The theory only can really cover determination of the parameters for a given value of the regularization and kernel parameters. Therefore, you could only choose the kernel.
- There might be a worse scenario as well, where the kernel model itself can be quite sensitive to over-fitting during the model selection criterion.

Decision trees

Decision trees are cool because of their usability they are easy to interpret and explain the machine learning problem around. In parallel, they can easily be handled for the feature related interactions. Most importantly, they are often non-parametric. Therefore, even if you are an ordinary data scientist with limited working proficiencies, you don't need to be worried about the issues such as outliers, parameter setting, and tuning. Sometimes fundamentally, you can relay with the decision trees so that they will make your stress for handling issue of the data linearity, or more technically, whether your data is linearly separable or not, you need not be worried. On the contrary, there are some cons as well. For example:

- In some cases, the decision tree will not be suitable, sometimes they don't support online learning for real-time datasets. In that case, you have to rebuild your tree when new examples or datasets come; more technically, gaining model adaptability would not be possible.
- Secondly, if you are not aware, they will easily become over-fitting.

Random forests

Random forests are quite popular and are a winner for the data scientist, since they are divine for a package with plenty of classification problems. They are usually slightly ahead of SVMs in terms of usability and have faster operation for most of the classification problems. In addition to this, they are also scalable when increasing the datasets you have available. In parallel, you don't need to be worried about tuning a cluster of parameters. On the contrary, you need to take care of many parameters and tuning when handling your data.

Decision forests, decision jungles, and variants

Decision forests, decision jungles, and boosted decision trees are all based on decision trees, a foundational machine learning concept that is less used. There are many variants of decision trees are there; nonetheless, they all do the same thing, which is subdividing the feature space into regions with the same label. In order to avoid the over-fitting problem, a large set of trees are constructed with mathematical and statistical formulations, where the trees are not correlated at all.

The average of this is referred to as a decision forest; which is a tree that avoids the overfitting problem as stated earlier. However, the disadvantage is that decision forests can use a lot of memory. Decision jungles, on the other hand, are a variant that consume less memory by sacrificing a slightly longer training time. Fortunately, the boosted decision trees avoid overfitting by limiting the number of subdivision and the number of permitted data points in each region.

Bayesian methods

When the experimental or sample dataset size is large, the Bayesian method often provides results for parametric models that are very similar to the results produced by other classical statistical methods. Some potential advantages of using the Bayesian method was summarized by Elam et al (W. T. Elam, B. Scruggs, F. Eggert, and J. A. Nicolosi, *Advantages and Disadvantages of Methods for Obtaining XRF NET Intensities*, Copyright ©JCPDS-International Centre for Diffraction Data 2011 ISSN 1097-0002). For example, it provides a natural way of combining prior information with data. Therefore, as a data scientist, you can incorporate that past information regarding the parameters and form a prior distribution for future analysis for new datasets. It also provides inferences that are conditional on the data without the need of asymptotic approximation of the algorithm.

It provides some suitable settings for a wide range of models, such as hierarchical models and missing data problems. There are also disadvantages of using Bayesian analysis. For example, it does not tell you how to select a prior over world models or even that there is no correct way to choose a prior. Therefore, if you do not proceed with caution, you might generate many false positive or false negative results that often come with a high computational cost, if the number of parameters in a model is large.

Summary

This ends our rather quick tour of machine learning and the best practice that needs to be followed. Although we have tried to cover some of the most basic things to remember, suitable data often beats better algorithms and better demand. Most importantly, to design good features out of your data might take a long time; however, it would very much aid you. However, if you have a large-scale dataset to be applied to your machine learning algorithms or model, whichever classification, clustering, or regression algorithm you use might not be a matter of fact concerning the machine learning classes and their respective classification performance.

Therefore, it would be a wise decision to choose an appropriate machine learning algorithm that can fulfill requirements such as speed, memory usage, throughput, scalability, or usability. In addition to going over what we said in the sections above, if you are really concerned about achieving the accuracy, you should undoubtedly try a group of different classifiers to find the best one using the cross-validation technique or just use an ensemble method to choose them alltogether.

You can also be motivated and take a lesson from the Netflix Prize PLUS. We spoke at length about the Spark machine learning APIs, some best practice in ML application development, machine learning tasks and classes, some widely used best practices, and so on. However, we have not shown in depth analysis of the machine learning techniques. We intend to talk about this in more detail in Chapter 4, *Extracting Knowledge through Feature Engineering*.

In the next chapter, we will cover in detail the DataFrame, Dataset, and **Resilient Distributed Dataset** (**RDD**) APIs for working with structured data targeting to provide a basic understanding of machine learning problems with the available data. Therefore, at the end, you will be able to apply from basic to complex data manipulation with ease.

Chapter 3. Understanding the Problem by Understanding the Data

This chapter will cover in details of the DataFrame, Datasets, and **Resilient Distributed Dataset** (**RDD**) APIs for working with structured data targeting to provide a basic understanding of machine learning problems with the available data. At the end of the chapter you will be able to apply basic to complex data manipulation with ease. Some comparisons will be made available with basic abstractions in Spark using RDD, DataFrame, and Dataset based data manipulation to show both gains in terms of programming and performance. In addition, we will guide you on the right track so that you will be able to use Spark to persist an RDD or data objects in memory, allowing it to be reused efficiently across the parallel operations in the later stage. In a nutshell, the following topics will be covered throughout this chapter:

- Analyzing and preparing your data
- Resilient Distributed Dataset (RDD) basics
- Dataset basics
- Dataset from string and typed class
- Spark and data scientists, workflow
- Deeper into Spark

Analyzing and preparing your data

In practice, several factors affect the success of **machine learning** (**ML**) applications on a given task. Therefore, the representation and quality of the experimental dataset is first and foremost considered as the first class entities. It is always advisable to have better data. For example, irrelevant and redundant data, data features with null values or noisy data result in unreliable source of information. The bad properties in datasets make the knowledge discovery process during the machine learning model training phase more tedious and time inefficient.

As a result, the data preprocessing will contribute a considerable amount of computational time across the total ML workflow steps. As we stated in the previous chapter, unless you know your available data, it would be difficult to understand the problem itself. Moreover, knowing the data will help you to formulate your problem. In parallel, and more importantly, before trying to apply an ML algorithm to a problem, first you have to identify if the problem is really a machine learning problem and whether an ML algorithm could directly be applied to solve the problem. The next step that you need to take is to know the machine learning classes. More technically, you need to

know if an identified problem falls under classification, clustering, rule retraction, or regression classes.

For the sake of simplicity, we assume you have a machine learning problem. Now you need to do some data pre-processing that includes some steps like data cleaning, normalization, transformation, feature extraction, and selection. The product of a data pre-processing workflow step is the final training set that is typically used to build/train the ML model.

In the previous chapter, we also argued that a machine learning algorithm learns from the data and activities during the model building and feed backing. It is critical that you feed your algorithm with the right data for the problem you want to solve. Even if you have good data (or well-structured data to be more precise), you need to make sure that the data is in an appropriate scale, with a well-known format to be parsed by the programming languages and, most importantly, if the most meaningful features are also included.

In this section, you will learn how to prepare your data so that your machine-learning algorithm becomes spontaneous towards best performance. The overall data processing is a huge topic; however, we will try to cover essential techniques to make some large scale machine learning applications in *Chapter 6*, *Building Scalable Machine Learning Pipelines*.

Data preparation process

If you are more focused and disciplined during the data handling and preparation steps, you are likely to get more consistent and better results in the first place. However, the data preparation is a tedious process consisting of several steps. Nevertheless, the process for getting data ready for a machine learning algorithm can be summarized in three steps:

- Data selection
- Data pre-processing
- Data transformation

Data selection

This step will focus on selecting the subset of all available datasets that you will be using and working with within your machine learning application development and deployment. There is always a strong urge to include all the available data in machine learning application development since more data will provide more features. In other words, by holding the well-known aphorism, *more is better*. However, essentially, this might not be true in all cases. You need to consider what data you need to have before you actually answer the question. The ultimate goal is to provide a solution of a particular hypothesis. You might be doing some assumptions about the

data as well in the first place. Although it is difficult, if you are a domain expert of that problem, you can make some assumption to know at least some insights before applying your ML algorithms. However, be careful to record those assumptions so that you can test them at a later stage when required. We will present some common question to help you out in thinking through the data selection process:

- The first question would be, *what is the extent of the data you have available?* For example, the extent could be the throughout time, database tables, connected system files, and so on. Therefore, the better practice is to ensure that you have a clear understanding and low-level structure of everything that you can use, or holding informally the available resources (while of course including the available data and computational resources).
- The second question is a little bit weird! *What data are not yet available but important to solve the problem?* In this case, you might have to wait for the data to be available or alternatively you can at least generate or simulate these types of data using some generator or software.
- The third question might be: *what data don't you need to address the problem?* That means again the redundancies so excluding these redundant or unwanted data is almost always easier than including it altogether. You might be wondering whether or not to note down the data you excluded and why? We think it should be yes since you might need some trivial data in the later stages.

Moreover, in practice in this case small problems or games, toy competition data will already have been selected for you; therefore, you don't need to be worried at all!

Data pre–processing

After you have selected the data you will be working with, you need to consider how you could use the data and the proper utilization required. This pre-processing step will address some steps or techniques for getting the selected data into a form that you can work and apply during your model building and validation steps. The three most common data pre-processing steps that are used are formatting, cleaning, and sampling the data:

- **Formatting**: The selected data may not be in a good shape so might not be suitable for you to work with directly. Very often, your data might be in a raw data format (a flat file format such as a text format or a less used proprietary format) and if you are lucky enough then data might be in a relational database. If this is the case, then it would better be to apply some conversion steps (that is, converting a relational database to its format for example, since using Spark you cannot make any conversion). As already stated, the beauty of Spark

is its support for diverse file formats. Therefore, we will be able to take advantage in the following sections.

- **Cleaning**: Very often the data you will be using comes with many unwanted records or sometimes with missing entries against a record. This cleaning process deals with the removal or fixing of missing data. There may be always some trivial data objects that are insignificant or incomplete and addressing them should be the first priority. Consequently, these instances may need to be removed, ignored or deleted from the datasets to get rid of this problem. Additionally, if the privacy or security is a concern because of the presence of the sensitive information against some attributes, those attributes need to be anonymized or removed from the data entirely (if appropriate).
- **Sampling**: The third step would be the sampling over the top of the formatted and cleaned datasets. Sampling is often required since there might be a time when the available data size is large or a number of records are huge. However, we argue to use the data as much as possible. Another reason is that more data can result in a longer execution time during the whole machine learning process. If this is the case, this also increases the running times of the algorithms and requires a more powerful computational infrastructure. Therefore, you can take a smaller representative sample of the selected data that may be much faster for exploring and prototyping the machine learning solution before considering the whole dataset. It is obvious that whatever the machine learning tools you apply for your machine learning application development and commercialization, data will influence the pre-processing you will be required to perform.

Data transformation

After selecting appropriate data sources and pre-processing those data, the final step is to transform the processed data. Your specific ML algorithm and knowledge of the problem domain will be influenced in this step. Three common data transformations techniques are scaling attributes, decompositions and attribute aggregations. This step is also commonly referred to as feature engineering that will be discussed in more details in the next chapter:

- **Scaling**: The pre-processed data may contain attributes with a mixture of scales for various quantities and units, for example dollars, kilograms, and sales volume. However, the machine-learning methods have the data attributes within the same scale such as between 0 and 1 for the smallest and largest value for a given feature. Therefore, consider any feature scaling you may need to perform the proper scaling of the processed data.
- **Decomposition**: The data might have some features that represent a complex concept that provides a more powerful response from the machine learning algorithms when you split the datasets into the fundamental parts. For example, consider a day that is composed of

24 hours, 1,440 minutes, and 86,400 seconds that in turn could be split out further. Probably some specific hours or only the hours in a day are relevant to the problem which to be investigated and resolved. Therefore, consider an appropriate feature extraction and selection to perform the proper decomposition of the processed data.

- **Aggregation**: Often segregated or scattered features may be trivial on their. However, those features can be aggregated into a single feature that would be more meaningful to the problem you are trying to solve. For example, several data instances can be presented in an online shopping website for each time a customer logged on the site. These data objects could be aggregated into a count for the number of logins by discarding additional instances. Therefore, consider appropriate feature aggregation to process the data properly.

Apache Spark has its distributed data structures includes RDD, DataFrame, and Datasets by which you can perform the data pre-processing efficiently. These data structures have different advantages and performance for processing the data. In the next sections, we will describe those data structures individually and also show examples of how to process the large Dataset using them.

Resilient Distributed Dataset basics

In Chapter 1, *Introduction to Data Analytics with Spark*, we have described the Resilient Distributed Datasets in brief including the data transformation and action as well as the caching mechanism. We also stated that RDDs are basically an immutable collection of records that can only be created by operations such as map, filter, group by, and so on. In this chapter, we are going to use this native data structure of Spark for data manipulation and data pre-processing for a practical machine learning application commonly referred to as spam-filtering. Spark provides another two higher label APIs such as DataFrame and Datasets for data manipulation.

We will, however, show all the APIs including RDD here because you might need this API to handle more complex data manipulation. We have referred to some commonly used definitions regarding Spark actions and operations from the Spark programming guide.

As we already discussed some basics of RDD operations using action and transformations. The RDDs can be created by both stable storages such as the **Hadoop Distributed File System** (**HDFS**) and by transformations on existing RDDs. Spark periodically logs those transformations while creating RDDs over a set of transformations, rather than actual data, therefore, technically the original RDD and the Datasets do not get changed.

A transformed Dataset can be created from an existing one; however, the reverse is not possible in Spark. After finishing a computation on the Dataset, an action returns a value to the driver program. For example, according to the Spark programming guidelines, the map is a transformation that passes each Dataset element using a function and returns a brand new RDD that represents and holds the results. In contrast, reduce is also an action that aggregates all the elements of an RDD by using a function and returns a brand new RDD too as the final result to the driver program.

More technically, suppose we have a text file that contains a sequence of number separated by commas (that is, a CSV file). Now after reading the same you will have an RDD and consequently, you might want to count the frequencies of each number. For doing this, you need to convert the RDD into key value pairs, where the key is the number and the value will be the frequency of each number.

On the other hand, you might need to collect the result in the driver program by doing some operations. In the next few sections, we will provide more

details on some useful topics such as transformations and actions by showing some examples based on a practical machine learning problem.

Reading the Datasets

For reading Datasets from different data sources like a local filesystem, HDFS, Cassandra, HBase, and more, Spark provides different APIs that are easy to use. It supports the different representation of data including text file, sequence file, Hadoop input format, CSV, TSV, TXT, MD, JSON, and other data formats. The input API or methods support running on compressed files, directories and wildcards. For example, *Table 1* shows the list of reading formats. The `textFile()` method reads different file formats such as `.txt` and `.gz` from the directory `/my/directory:`

textFile("/my/directory"), textFile("/my/directory/*.txt"), textFile("/my/directory/*.gz").

Table 1: Reading files formats

Reading from files

You might need to read a Dataset from local or HDFS. The following code show the different methods for creating RDDs from a given Dataset stored on your local machine or in HDFS.

However, before reading and writing with Spark, we need to create the Spark entry point by means of a Spark session that can be instantiated as follows:

```
static SparkSession spark = SparkSession
      .builder()
      .appName("JavaLDAExample")
      .master("local[*]")
      .config("spark.sql.warehouse.dir",
"E:/Exp/")
      .getOrCreate();
```

Here, the Spark SQL warehouse is set to as `E:/Exp/` path. You should set your path accordingly based on OS types you are on. Well, now we have our Spark session as variable `spark`, let's see how to use it with ease for reading from a text file.

Reading from a text file

It uses `textFile()` methods of `SparkContext()` and returns an RDD of a string containing a collection of lines. In Chapter 1, *Introduction to Data Analytics with Spark*, we explained what SparkContext is. Nevertheless, Spark Context is the entry point of a Spark application. Suppose we have a Dataset called `1.txt` containing some tweets data as unstructured texts. You can download the data from the Packt materials and store under the `project_path/input/test/` directory, defined as follows:

```
String csvFile = "input/test/1.txt";
RDD<String> distFile =
spark.sparkContext().textFile(csvFile, 2);
```

Here we have created RDDs of a string that is stored with the variable `distFile` in two partitions. However, to work with Java, the RDDs have to be converted into JavaRDD. Let's do it by calling the `toJavaRDD()` method as follows:

```
JavaRDD<String> distFile2 = distFile.toJavaRDD();
```

Reading multiple text files from a directory

It will return RDD as (filename and content) pairs. Suppose we have multiple files stored to read in the directory `csvFiles/` is defined as follows:

```
RDD<Tuple2<String, String>> distFile =
spark.sparkContext().wholeTextFiles("csvFiles/*.txt",
2);
JavaRDD<Tuple2<String, String>> distFile2 =
distFile.toJavaRDD();
```

Please note, when the data objects in an RDD do not hold in the main memory or HDD, we need to perform a partition on the RDD to increase the parallelism.

Reading from existing collections

The second source for creating an RDD is from the collections of your driver program such as list a containing integers. Before going deeper into this, let's initialize Spark in an alternative way as follows:

```
SparkConf conf = new
SparkConf().setAppName("SampleAppliation").setMaster("local[*]");
JavaSparkContext sc = new JavaSparkContext(conf);
```

Here Java Spark context is available as a variable `sc`. This time, we have created the Spark context so we will be able to create the Java RDDs of string without using the `toJavaRDD()` method.

Now, you can do it by using the parallelized method of Spark Context as shown here:

- **Reading list of integers**: It returns a parallelized RDD of integers:

```
      List<Integer> list =
Arrays.asList(1,2,3,4,5);
      JavaRDD<Integer> rdd =
sc.parallelize(list);
```

- **Reading list of pairs**: It returns a parallelized `pairRDD` of the list of pairs (integer, string):

```
      List<Tuple2<Integer, String>> pairs =
Arrays.asList(
                  new Tuple2<>(1, "Hello"),
                  new Tuple2<>(2, "World"),
                  new Tuple2<>(3, "How are
you?"));
      JavaPairRDD<Integer, String> pairRDD =
sc.parallelizePairs(pairs);
```

Pre–processing with RDD

To continue the discussion about the data pre-processing that we started in the previous section, we will show an example of a machine learning problem and how to pre-process the Dataset using RDD in this section.

We are considering the **Spam filter** application that is a popular example of s supervised learning problem. The problem is to predict and identify the spam messages from the incoming e-mails (please refer to *Table 2*). As usual, to train the model, you have to train a model by using the historical data (the historical e-mail that you have received over a couple of days, hours or months an even year). The final output of pre-processing tasks is to make the feature vectors or extract the features including its labels or classes. Typically, you might be doing the following steps:

- **Stop word removal**: The text file might contain some word that is useless or redundant for the feature vector such as *and*, *the*, and *of*, since these are very common in all forms of English sentences. Another reason is they are not very meaningful in deciding spam or ham status or they may contain trivial significance. These words, therefore, need to be filtered from the e-mail Dataset before moving the next step.

- **Lemmatization**: Some words possessing the same meaning but with different endings, need to be readjusted in order to make them consistent across the data set and if they all carry the same form will be easier to make them transform into feature vectors. For example, *attached*, *attachment*, and *attach* could all be represented and later on interpreted as e-mail *attachments*. The `SMSSpamCollection` Dataset was downloaded from the UCI ML repositories at https://archive.ics.uci.edu/ml/machine-learning-databases/00228/.

  Please note, all the words in the email body are usually converted to lowercase for simplicity in this phase. Now, let's take a look at the following table:

ham: What you doing? how are you?
ham: Ok lar... Joking wif u oni.
ham: dun say so early hor... U c already then say.
ham: MY NO. IN LUTON 0125698789 RING ME IF UR AROUND! H*
spam: FreeMsg: Txt: CALL to No: 86888 & claim your reward of 3 hours talk time to use from your phone now! ubscribe6GBP/ mnth inc 3hrs 16 stop?txtStop
ham: Siva is in hostel aha.
ham: Cos i was out shopping wif darren jus now n i called him 2 ask wat present he wan lor. Then he started guessing who i was wif n he finally guessed darren lor.
spam: Sunshine Quiz! Win a super Sony DVD recorder if you can name the capital of Australia? Text MQUIZ to 82277. B

  Table 2: Test file for training set containing ham and spam messages

- **Removal of non-words**: Numbers and punctuation have to be removed too. However, we will not show here all the possible transformation for pre-processing data due to page limitation and brevity, but we will try to show some basic transformations and actions for pre-processing segment of the Dataset that contains some labels data as spam or ham presented in *Table 2*. Where the Dataset or e-mails are labelled as ham or spam followed by the message or e-mails. Ham means non-spam and spams are identified as junk emails messages.

The overall pre-processing using RDD can be described using the following steps. The first step we need is to prepare the feature vectors using RDD operations and transformations. The remaining steps are given as follows:

- **Reading Dataset**: The following code is for reading Dataset that creates a `linesRDD` of strings from the `SMSSpamCollection` Dataset. Please download this Dataset from the Packt materials and store it in your disk or HDFS in `Project+path/input/` directory. A detailed description of this Dataset will be provided later on:

```
      String filePath =
"input/SMSSpamCollection.txt";
      JavaRDD<String> linesRDD =
sc.textFile(filePath);
```

However, the `linesRDD` contains both the spam as well as the ham messages. Therefore, we need to separate the spam and ham message from the files.

- **Filter out the spam messages**: To filter the data from existing RDDs, Spark provides a method called `filter()`, which returns a new Dataset containing only the selected elements. In the following code you can see we have passed `new Function()` as a parameter that takes two arguments of the type String and Boolean of the`filter()` method. Basically, Spark APIs heavily rely on passing functions to the driver program for running on the cluster. There are two ways to create a function that includes:
    - Implementing the function interfaces either creating anonymous inner class or named one and passing an instance of it to Spark
    - Using lambda expressions (you will have to have Java 8 installed to take advantage of lambda expressions though)
    - The following code segments that we have used explain the concept of anonymous class as a parameter that contains a `call()` method that returns `true` if the line contains the word `spam`:

```
      JavaRDD<String> spamRDD =
linesRDD.filter(new Function<String,
        Boolean>() {
        @Override
        public Boolean call(String line)
throws Exception {
          return
line.split("\t")[0].equals("spam");}});
```

- Filter out the ham messages: similarly, we can filter out the ham messages as shown here:

```
        JavaRDD<String> hamRDD =
linesRDD.filter(new Function<String,
        Boolean>() {
        @Override
        public Boolean call(String line)
throws Exception {
          return
line.split("\t")[0].equals("ham");
        }
      });
```

- **Split the words from the lines**: To extract the features and labels from each line, we need to split those using space or tab characters. After that, we can have the lines without the spam or ham words.

  The following code segments show the separation of spam and ham features from the lines. We have used the `map` transformation that returns a new RDD formed by passing each line of the existing RDD through a function `call`. Here the `call` method always returns a single item. You will find a difference with `flatMap` in the later section:

```
        JavaRDD<String> spam = spamRDD.map(new
Function<String, String>() {
          @Override
          public String call(String line)
throws Exception {
           return line.split("\t")[1];
          }
      });
```

Output: `ham.collect()`:

```
Free entry in 2 a wkly comp to win FA Cup final tkts 21st May 2005. Text FA to
FreeMsg Hey there darling it's been 3 week's now and no word back! I'd like
WINNER!! As a valued network customer you have been selected to receivea £900
Had your mobile 11 months or more? U R entitled to Update to the latest colour
SIX chances to win CASH! From 100 to 20,000 pounds txt> CSH11 and send to 87575
URGENT! You have won a 1 week FREE membership in our £100,000 Prize Jackpot!
XXXMobileMovieClub: To use your credit, click the WAP link in the next txt
England v Macedonia - dont miss the goals/team news. Txt ur national team to
Thanks for your subscription to Ringtone UK your mobile will be charged £5/month
07732584351 - Rodger Burns - MSG = We tried to call you re your reply to our
SMS. ac Sptv: The New Jersey Devils and the Detroit Red Wings play Ice Hockey.
Congrats! 1 year special cinema pass for 2 is yours. call 09061209465 now! C
As a valued customer, I am pleased to advise you that following recent review
Urgent UR awarded a complimentary trip to EuroDisinc Trav, Aco&Entry41 Or £1000
Did you hear about the new "Divorce Barbie"? It comes with all of Ken's stuff!
```

Figure 1: A snapshot of the spam RDD

```
        JavaRDD<String> ham = hamRDD.map(new
Function<String, String>() {
          @Override
          public String call(String line) throws
Exception {
```

```
            return line.split("\t")[1];
        }
        });
```

Output: `ham.collect()`:

```
1  Go until jurong point, crazy.. Available only in bugis n great world la e
2  Ok lar... Joking wif u oni...
3  U dun say so early hor... U c already then say...
4  Nah I don't think he goes to usf, he lives around here though
5  Even my brother is not like to speak with me. They treat me like aids patent.
6  As per your request 'Melle Melle (Oru Minnaminunginte Nurungu Vettam)
7  I'm gonna be home soon and i don't want to talk about this stuff anymore tonight
8  I've been searching for the right words to thank you for this breather.
9  I HAVE A DATE ON SUNDAY WITH WILL!!
10 Oh k...i'm watching here:)
11 Eh u remember how 2 spell his name... Yes i did. He v naughty make until i v wet.
12 Fine if the way u feel. the way its gota b
13 Is that seriously how you spell his name?
14 I'm going to try for 2 months ha ha only joking
15 So ü pay first lar... Then when is da stock comin...
16 Aft i finish my lunch then i go str down lor. Ard 3 smth lor
17 Fffffffff. Alright no way I can meet up with you sooner?
```

Figure 2: A snapshot of the ham RDD

- **Split the words from the lines of spam RDD**: After we get the feature lines against the spam and ham RDDs separately, we have to split the words for makinga feature vector in the future. The following codes do this split with a space by returning the wordlist RDD. The call method returns a list of words for each line:

```
      JavaRDD<ArrayList<String>> spamWordList =
spam.map(new
        Function<String, ArrayList<String>>() {
          @Override
      public ArrayList<String> call(String line)
throws Exception{
            ArrayList<String> words = new
ArrayList<>();

 words.addAll(Arrays.asList(line.split(" ")));
            return words;
          }});
      JavaRDD<ArrayList<String>> hamWordList =
ham.map(new Function<String,
        ArrayList<String>>() {
          @Override
      public ArrayList<String> call(String line)
throws Exception{
            ArrayList<String> words = new
ArrayList<>();

 words.addAll(Arrays.asList(line.split(" ")));
            return words;}
      });
```

- **Make label and feature pair RDD**: Now we have two RDDs for spam and ham. We want to label them by 1.0 or 0.0 for spam and ham words or features respectively. For ease of use, we can again create a new RDD-containing tuple of a label and features or wordlist for each line. In the following code we have used `Tuple2` for making a pair. You can also use `JavaPairRDD` for making a pair of labels and features:

```
      JavaRDD<Tuple2<Double,
ArrayList<String>>> spamWordsLabelPair =
      spamWordList.map(new
Function<ArrayList<String>, Tuple2<Double,
          ArrayList<String>>() {
          @Override
            public Tuple2<Double,
ArrayList<String>> call(
            ArrayList<String> v1) throws
Exception {
            return new Tuple2<Double,
ArrayList<String>>(1.0, v1);
          }});
      JavaRDD<Tuple2<Double,
ArrayList<String>>> hamWordsLabelPair =
      hamWordList.map(new
Function<ArrayList<String>, Tuple2<Double,
          ArrayList<String>>() {
          @Override
          public Tuple2<Double,
ArrayList<String>> call(
            ArrayList<String> v1) throws
Exception {
            return new Tuple2<Double,
ArrayList<String>>(0.0, v1);
          }});

      [Output: print
spamWordsLabelPair2.collect() using for loop]
      1.0: [FreeMsg:, Txt:, CALL, to, No:,
86888, &, claim, your, reward,
      of, 3, hours, talk, time, to, use,
from, your, phone, now!,
      ubscribe6GBP/, mnth, inc, 3hrs, 16,
stop?txtStop]
      1.0: [Sunshine, Quiz!, Win, a, super,
Sony, DVD, recorder,
      if, you, canname, the, capital, of,
Australia?, Text, MQUIZ,
      to, 82277., B]
```

- **Union of the two RDDs**: Now we have two labels for the Dataset in *Table 2*, the feature pair RDD of spam and ham. Now to make the training Dataset, we can join these two RDDs into one. Spark has

`union()` method for doing this that returns a new RDD, containing the union of the Dataset and the argument or another Dataset:

```
      JavaRDD<Tuple2<Double,
ArrayList<String>>> train_set =

 spamWordsLabelPair.union(hamWordsLabelPair);
```

- Counting all the lines from the preceding operations, are called a transformation. This returns a new Dataset from existing one in the worker nodes in each case. If you want to bring the return in the driver program or print the results that would be called an action operation. Spark supports several built-in methods as actions. The `count()` method counts the number of elements in the Dataset:

```
      System.out.println(train_set.count());
      The following output is 8
```

- **Printing the RDD**: The `collect()` and `take()` are also action method that are used to print or collect the Dataset as an array in the driver program, where, `take()` takes an argument of, say *n* that returns the first n elements of that Dataset. The following code segments print the first 10 elements or tuples out of the train set:

```
      for (Tuple2<Double, ArrayList<String>>
tt : train_set.collect()) {
          System.out.println(tt._1 + ": " +
tt._2.toString()); }
```

The output is as follows:

```
      1.0: [FreeMsg:, Txt:, CALL, to, No:,
86888,
       &, claim, your, reward, of, 3, hours,
talk, time, to, use, from,
       your, phone, now!, ubscribe6GBP/,
mnth, inc, 3hrs, 16, stop?txtStop]
      1.0: [Sunshine, Quiz!, Win, a, super,
Sony, DVD, recorder, if,
      you, canname, the, capital, of,
Australia?, Text, MQUIZ,
      to, 82277., B]
      0.0: [What, you, doing?, how, are,
you?]
      0.0: [Ok, lar..., Joking, wif, u,
oni...]
      0.0: [dun, say, so, early, hor..., U,
c, already, then, say...]
      0.0: [MY, NO., IN, LUTON, 0125698789,
RING, ME, IF, UR, AROUND!, H*]
      0.0: [Siva, is, in, hostel, aha:-.]
      0.0: [Cos, i, was, out, shopping, wif,
darren, jus, now,
```

```
        n, i, called,
       him, 2, ask, wat, present, he, wan,
lor., Then, he,
       started, guessing,
       who, i, was, wif, n, he, finally,
guessed, darren, lor.]
```

- **Save the result in the local filesystem**: Sometimes you might need to save the RDD in the filesystem as text. You can use the following code for saving your RDDs straight away:

```
train_set.saveAsTextFile("output.txt");
```

Getting insight from the SMSSpamCollection dataset

The following source code shows the basic ham and spam statistics:

```
String path = "input/SMSSpamCollection.txt";
RDD<String> lines =
spark.sparkContext().textFile(path, 2);
System.out.println(lines.take(10));

JavaRDD<Row> rowRDD = lines.toJavaRDD().map( new
Function<String, Row>() {
    public Row call(String line) throws Exception
{
      return RowFactory.create(line);
      }});
System.out.println(rowRDD.collect());
List<StructField> fields = new
ArrayList<StructField>();
fields.add(DataTypes.createStructField("line",
DataTypes.StringType, true));
org.apache.spark.sql.types.StructType schema =
DataTypes.createStructType(fields);
Dataset<Row> df =
spark.sqlContext().createDataFrame(rowRDD,
schema);
df.select("line").show();
Dataset<Row> spam =
df.filter(df.col("line").like("%spam%"));
Dataset<Row> ham =
df.filter(df.col("line").like("%ham%"));
System.out.println(spam.count());
System.out.println(ham.count());
spam.show();
```

The preceding code generates the following spam and ham counts:

```
747
4831
```

This means that out of 5,578 emails `747` e-mails are spam and `4,831` e-mails are labelled as ham or non-spam. In other words, the spam and ham ratio is 13.40% and 86.6%.

Working with the key/value pair

This subsection describes the key/value pair that is frequently needed in the data analytics, especially in the text processing.

mapToPair()

This method will return a Dataset of (K, V) pair where K is key and V is value. For example, if you have an RDD with a list of integer then you want to count the number of duplicate entries in the list then the first task is to map each number to 1. After that you can do the reduce operation on it. The code produces the output and cache as shown here:

```
JavaRDD<Integer> rdd =
sc.parallelize(Arrays.asList(1,2,1,3,4,5));
JavaPairRDD<Integer, Integer> pairs =
rdd.mapToPair(
  new PairFunction<Integer, Integer, Integer>() {
    @Override
    public Tuple2<Integer, Integer> call(Integer
x) {
      return new Tuple2<>(x, 1);
    }
}).cache();

[Output: pairs.collect()]
[(1,1), (2,1), (1,1), (3,1), (4,1), (5,1)]
```

More about transformation

In this section, you can see more about transformation including some differences between similar types of methods. Mainly `map` and `flatMap`, `groupByKey`, `reduceByKey` and `aggregateByKey`, `sortByKey` and `sortBy` will be discussed in this section. However, interested readers can refer to the Spark programming guidelines for RDD operation in [2].

map and flatMap

The `flatMap` is similar to the map we have showed in the preceding examples, but each input item or each time calling the `call()` method of the anonymous class can be mapped to zero or more output items. So ,the `call()` function returns a Sequence rather than a

single item like a map. For example, for input of following RDD, the output should be as follows:

```
JavaRDD<String> rdd =
sc.parallelize(Arrays.asList("Hello World!",
  "How are you."));
JavaRDD<String> words = rdd
  .flatMap(new FlatMapFunction<String, String>()
{
    @Override
    public Iterable<String> call(String t) throws
Exception {
      return Arrays.asList(t.split(" "));
    }});

[output: words.collect()]
[Hello, World!, How, are, you.]
```

For the previous example, you could not do the map operation because the `call()` method of map return only one object rathers than a sequence of the objects.

groupByKey, reduceByKey, and aggregateByKey

In order to perform some operation while pre-processing your Dataset you might need to do some aggregation such as sum and average based on key values. Spark provides some methods for doing these kinds of operations. Let's say, you have the following pairs of RDD and you want to group the values based on the keys and do some aggregations:

```
JavaPairRDD<Integer, Integer> rdd_from_integer =
sc
.parallelizePairs(Arrays.asList( new Tuple2<>(1,
1),
new Tuple2<>(1, 1), new Tuple2<>(3, 2),
new Tuple2<>(5, 1), new Tuple2<>(5, 3)), 2);
```

The aggregations you want to do can be done by the three methods of Spark including `groupByKey`, `reduceByKey`, and `aggregateByKey`. But they differ in term of performance, efficiency and flexibility to do an operation such as counting, computing summary statistics, finding unique elements from a data set and so on. The `groupByKey` method returns a Dataset of (k, `Iterable<v>`) pairs where k is the key and `Iterable<v>` is the sequence of values of the key k. The output of previous Dataset using this method is given as follows and it shows the collection values of each key:

```
[Output: pairs.groupByKey(2).collect() ]
```

```
[(1,[1, 1]), (3,[2]), (5,[1, 3])]
```

Figure 3: Pairs using groupBykey

In order to make the sum of values of each unique key, `groupByKey` is inefficient in terms of performance because it does not perform the map side in combination. You have to make more transformation to do this summation explicitly. So, it increases the network I/O and shuffle size. Better performance can be gained by `reduceByKey` or `aggregateByKey` because they perform the map side combination.

The methods return the Dataset with the result of each key aggregation such as summations of the values of each key. The following code show the operation of those methods which return the Dataset of (k,v) pairs where values (v) of the keys are aggregated by the given functions.

The `reduceByKey` takes one function that reduces the values of each key while the `aggregateByKey` takes two functions where the first function is for specifying how the aggregation will take place inside each partition and the second function is for specifying how the aggregation will take place between the partitions:

- Code: `reduceByKey()`:

```
      JavaPairRDD<Integer, Integer> counts =
rdd
        .reduceByKey(new Function2<Integer,
Integer, Integer>() {
          @Override
          public Integer call(Integer a,
Integer b) {
            return a + b;}});
```

- Code: `aggregateByKey()`:

```
      JavaPairRDD<Integer, Integer> counts =
pairs.aggregateByKey(0,
          new Function2<Integer, Integer,
Integer>() {
            @Override
            public Integer call(Integer v1,
Integer v2) {
              return v1 + v2;
            }
            }, new Function2<Integer,
Integer, Integer>() {
            @Override
            public Integer call(Integer v1,
Integer v2) {
                  return v1 + v2;
                }
              });
```

For both cases, the output will be as shown here:

Output: `counts.collect()`:

```
[(1,2), (3,2), (5,4)]
```

Figure 4: RDD using count

sortByKey and sortBy

Sorting is a common operation in data pre-processing. Spark provides two methods that transform one Dataset to another sorted paired Dataset that includes `sortByKey` and `sortBy`. For instance, we have a Dataset as shown here:

```
List<Tuple2<Integer, Integer>> pairs = new
ArrayList<>();
pairs.add(new Tuple2<>(1, 5));
pairs.add(new Tuple2<>(4, 2));
pairs.add(new Tuple2<>(-1, 1));
pairs.add(new Tuple2<>(1, 1));
```

The `sortByKey()` method performs on (k,v) pairs and returns (k,v) pairs sorted by keys in ascending or descending order. You can also customize the sorting by providing a comparator as parameters. The following code shows the sorting by key of the preceding Dataset:

- Code: `sortByKey()`:

```
      JavaPairRDD<Integer, Integer> rdd =
sc.parallelizePairs(pairs);
      JavaPairRDD<Integer, Integer>
sortedRDD=rdd.sortByKey(Collections.
      <Integer> reverseOrder(), false);
      [Output: sortedRDD.collect()]
```

```
[(-1,1), (1,5), (1,1), (4,2)]
```

Figure 5: Pairs using sortByKey

The `sortBy()` method takes a function as a parameter where you can specify the sorting method either by key or by value. The following code shows the sorting by values of the preceding Dataset:

- Code: `sortBy()`:

```
      JavaRDD<Tuple2<Integer, Integer>> rdd_new =
sc.parallelize(pairs);
```

```
      JavaRDD<Tuple2<Integer, Integer>>
sortedRDD=rdd.sortBy(
      new Function<Tuple2<Integer, Integer>,
Integer>() {
      @Override
          public Integer call(Tuple2<Integer,
Integer> t) {
              return t._2();
          }
      ,} true, 2);
```

Output: `sortedRDD.collect()`:

```
[(-1,1), (1,1), (4,2), (1,5)]
```

Figure 6: Pairs using sortBy

Dataset basics

As discussed in Chapter 1, *Introduction to Data Analytics with Spark* that in Spark 2.0.0 release, the DataFrame remains the primary computation abstraction for the Scala, Python and R, however, while using Java the same will be replaced with Dataset. Consequently, Dataset of type Row will be used throughout this book.

The Dataset is a distributed collection of data is structured the Rows. This is this is one of the more convenient ways for interacting with Spark SQL module.

In other words, it can be considered as an equivalent entity to a tabular data like a **Relational Database** (**RDB**) format.. The Like the other data abstractions like DataFrame and RDD, the Dataset can also be created from various data sources like structured data files (TSV, CSV, JSON, and TXT), Hive tables, secondary storages, external databases, or existing RDDs and DataFrames. However, upon the Spark 2.0.0 release, the Java based computation does not support the DataFrame but you are developing your applications using Python, Scala or R, still you will be able making use of the DataFrames.

In the next few sections, you will find the operations and actions using Dataset and how to create a Dataset from different sources.

Reading datasets to create the Dataset

As mentioned above, the Dataset is a component of Spark SQL module introduced from the Spark 1.5.0 release. Therefore, all the entry point of all functionally starts from the initialization of Spark `SQLContext` . Basically, Spark SQL is used for executing SQL queries written either as a basic SQL syntax or HiveQL.

A Dataset object will be returning when running SQL within another programming language. The following code segment will initialize the `SQLContext` within Spark Context. On the other hand, you might require having the `HiveContext` initialized for reading a data set from the Hive. You can also create a different context like `HiveContext` which provides a superset of basic functionalities of `SQLContext`:

```
JavaSparkContext sc = new
JavaSparkContext("local","DFDemo");
SQLContext sqlContext = new
org.apache.spark.sql.SQLContext(sc);
```

Reading from the files

For example, you have a JSON file as shown here. Now you want to read this file using SQL context which basically returns a DataFrame which you can perform all the basic SQL operations and other DSL operations of Spark:

```
[Input File]
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}
[Code]
Dataset<Row> df =
sqlContext.read().json("people.json");

[Output: df.show()]
+----+-------+
| age|   name|
+----+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+
```

Reading from the Hive

The following code connects with Hive context where one table is created and people JSON file is loaded into hive create. The output of the DataFrame will be the same as above:

```
The code is as follows:]hiveContext.sql("CREATE
TEMPORARY TABLE people USING
org.apache.spark.sql.json OPTIONS ( path
"people.json" )");
Dataset<Row> results = hiveContext.sql("SELECT *
FROM people ");
results.show();
```

Pre-processing with Dataset

In the previous section, we have described the pre-processing with RDD for a practical machine learning application. Now we will do the same example using **DataFrame** (**DF**) API. You will find it very easy to manipulate the `SMSSpamCollection` Dataset (see at http://www.dt.fee.unicamp.br/~tiago/smsspamcollection/). We will show the same example by tokenizing the spam and ham messages for preparing a training set:

- **Reading a Dataset:** You can read that Dataset using the Spark session variable `spark`that we have to initialize before using it. After

reading the file as Dataset the output will be a tabular format of a single column. The default name of this column is `value`:

```
        Dataset<Row> df =
spark.read().load("input/SMSSpamCollection.txt");
        df.show();
```

Output:



```
+--------------------+
|               value|
+--------------------+
|ham    Go until juro...|
|ham    Ok lar... Jok...|
|spam   Free entry i...|
|ham    U dun say so ...|
|ham    Nah I don't t...|
|spam   FreeMsg Hey ...|
|ham    Even my broth...|
|ham    As per your r...|
|spam   WINNER!! As ...|
|spam   Had your mob...|
|ham    I'm gonna be ...|
|spam   SIX chances ...|
|spam   URGENT! You ...|
|ham    I've been sea...|
|ham    I HAVE A DATE...|
|spam   XXXMobileMov...|
|ham    Oh k...i'm wa...|
|ham    Eh u remember...|
|ham    Fine if that...|
|spam   England v Ma...|
+--------------------+
only showing top 20 rows
```

Figure 7: A snapshot of the SMS spam dataset

- **Create Row RDD from existing Dataset**: From the preceding output you can see one column containing all the lines together. In order to make two columns such as label and features, we have to split it. Since Dataset is immutable you cannot modify the existing columns or Dataset. So you have to create new Dataset using the existing Dataset. Here the code converts the Dataset to RDD that is the collection of Row dataset. The row is an interface, which represents one row of output from a relational operator. You can create a new Row using `RowFactory` class of Spark:

```
        JavaRDD<Row> rowRDD =
df.toJavaRDD();
```

- **Create new row RDD from an existing row RDD**: After having the Row RDD you can perform normal map operation which is all contains Row Dataset but having two values. The following code split the each row and returns a new one:

```
      JavaRDD<Row> splitedRDD =
rowRDD.map(new Function<Row, Row>() {
          @Override
         public Row call(Row r) throws
Exception {
           String[] split =
r.getString(0).split("\t");
           return
RowFactory.create(split[0],split[1]);
         }});
```

- **Create Dataset from Row RDD**: Now you have Row RDD, which contains two values for each Row. For creating a DF, you have to define the column names or schemas and its data types. There are two methods to define including inferring the schema using reflection and programmatically specify the schema. The methods are as follows:

    - The 1st method basically uses the POJO classes and fields names will be the schema
    - The 2nd method create list of StruchFields by defining the datatypes and create the structype. For this example, we have used the 2nd method for creating DF from existing row RDD as shown here:

```
      List<StructField> fields  = new
ArrayList<>();

 fields.add(DataTypes.createStructField("labelString",
      DataTypes.StringType, true));

 fields.add(DataTypes.createStructField("featureString",

      DataTypes.StringType, true));

 org.apache.spark.sql.types.StructType
schema = DataTypes
      .createStructType(fields);
      Dataset<Row>
schemaSMSSpamCollection = sqlContext
      .createDataFrame(splitedRDD,
schema);

 schemaSMSSpamCollection.printSchema();
      [Output:
schemaSMSSpamCollection.printSchema()]
```

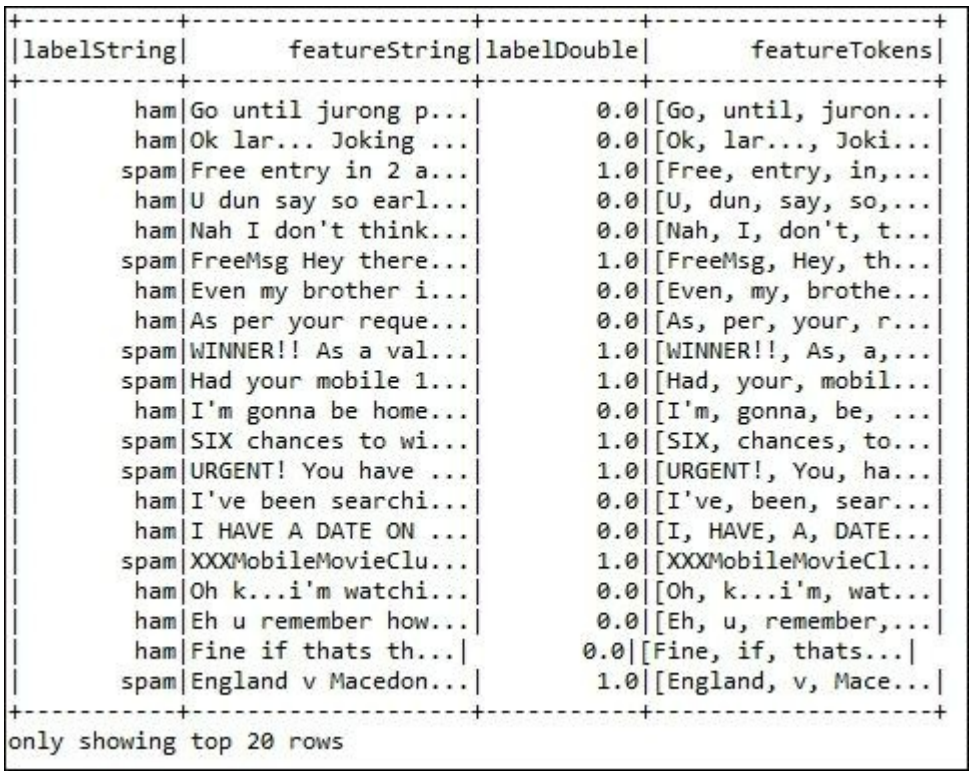
```
root
 |-- labelString: string (nullable = true)
 |-- featureString: string (nullable = true)

16/10/07 16:26:28 INFO SparkContext: Invoking stop() from shutdown hook
16/10/07 16:26:28 INFO SparkUI: Stopped Spark web UI at http://10.2.17.24:
16/10/07 16:26:28 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMas
16/10/07 16:26:28 INFO MemoryStore: MemoryStore cleared
16/10/07 16:26:28 INFO BlockManager: BlockManager stopped
```

Figure 8: Schema of the collection

- **Adding a new column**: Now that we have the DF of two columns. But we want to add new columns which convert the `labledSting` to `labedDouble` and `featureString` to `featureTokens`. You can do it similarly as previous code. After adding to new fields create new schema. Then create new DF after having normal map transformation in existing DF. The following code gives output of new DF having four columns:

```
 fields.add(DataTypes.createStructField("labelDouble",

      DataTypes.DoubleType, true));

 fields.add(DataTypes.createStructField("featureTokens",

      DataTypes.StringType, true));
      org.apache.spark.sql.types.StructType
schemaUpdated =
      DataTypes.createStructType(fields);
      Dataset Row> newColumnsaddedDF =
sqlContext

 .createDataFrame(schemaSMSSpamCollection.javaRDD().map(
      new Function<Row, Row>() {
          @Override
          public Row call(Row row) throws
Exception {
            double label;

 if(row.getString(0).equalsIgnoreCase("spam"))
              label = 1.0;
            else
              label = 0.0;
            String[] split =
row.getString(1).split(" ");
            ArrayList<String> tokens = new
ArrayList<>();
            for(String s:split)
              tokens.add(s.trim());
            return
RowFactory.create(row.getString(0),
```

```
                row.getString(1),label,
        tokens.toString());
                   }}), schemaUpdated);
              [Output: newColumnsaddedDF.show()]
```



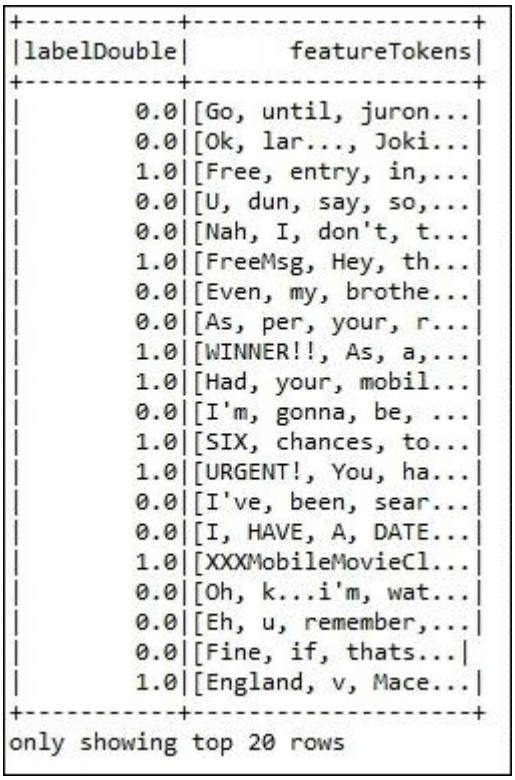
labelString	featureString	labelDouble	featureTokens
ham	Go until jurong p...	0.0	[Go, until, juron...
ham	Ok lar... Joking ...	0.0	[Ok, lar..., Joki...
spam	Free entry in 2 a...	1.0	[Free, entry, in,...
ham	U dun say so earl...	0.0	[U, dun, say, so,...
ham	Nah I don't think...	0.0	[Nah, I, don't, t...
spam	FreeMsg Hey there...	1.0	[FreeMsg, Hey, th...
ham	Even my brother i...	0.0	[Even, my, brothe...
ham	As per your reque...	0.0	[As, per, your, r...
spam	WINNER!! As a val...	1.0	[WINNER!!, As, a,...
spam	Had your mobile 1...	1.0	[Had, your, mobil...
ham	I'm gonna be home...	0.0	[I'm, gonna, be, ...
spam	SIX chances to wi...	1.0	[SIX, chances, to...
spam	URGENT! You have ...	1.0	[URGENT!, You, ha...
ham	I've been searchi...	0.0	[I've, been, sear...
ham	I HAVE A DATE ON ...	0.0	[I, HAVE, A, DATE...
spam	XXXMobileMovieClu...	1.0	[XXXMobileMovieCl...
ham	Oh k...i'm watchi...	0.0	[Oh, k...i'm, wat...
ham	Eh u remember how...	0.0	[Eh, u, remember,...
ham	Fine if thats th...	0.0	[Fine, if, thats...
spam	England v Macedon...	1.0	[England, v, Mace...

only showing top 20 rows

Figure 9: The dataset after adding a new column

- **Some Dataset operations**: For data manipulation DF provides domain specific language in Java, Scala and others. You can do select, counting, filter, `groupBy` and so on operations into a DF. The following codes show some operations on the above DF:

```
newColumnsaddedDF.select(newColumnsaddedDF.col("labelDouble"),

newColumnsaddedDF.col("featureTokens")).show();
```



```
+-----------+--------------------+
|labelDouble|       featureTokens|
+-----------+--------------------+
|        0.0|[Go, until, juron...|
|        0.0|[Ok, lar..., Joki...|
|        1.0|[Free, entry, in,...|
|        0.0|[U, dun, say, so,...|
|        0.0|[Nah, I, don't, t...|
|        1.0|[FreeMsg, Hey, th...|
|        0.0|[Even, my, brothe...|
|        0.0|[As, per, your, r...|
|        1.0|[WINNER!!, As, a,...|
|        1.0|[Had, your, mobil...|
|        0.0|[I'm, gonna, be, ...|
|        1.0|[SIX, chances, to...|
|        1.0|[URGENT!, You, ha...|
|        0.0|[I've, been, sear...|
|        0.0|[I, HAVE, A, DATE...|
|        1.0|[XXXMobileMovieCl...|
|        0.0|[Oh, k...i'm, wat...|
|        0.0|[Eh, u, remember,...|
|        0.0|[Fine, if, thats...|
|        1.0|[England, v, Mace...|
+-----------+--------------------+
only showing top 20 rows
```

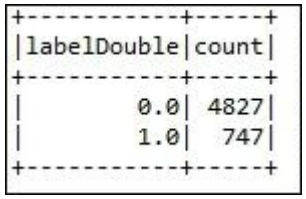
Figure 10: Dataset showing the label and features

```
newColumnsaddedDF.filter(newColumnsaddedDF.col
     ("labelDouble").gt(0.0)).show();
```

```
+-----------+--------------------+-----------+--------------------+
|labelString|       featureString|labelDouble|       featureTokens|
+-----------+--------------------+-----------+--------------------+
|       spam|Free entry in 2 a...|        1.0|[Free, entry, in,...|
|       spam|FreeMsg Hey there...|        1.0|[FreeMsg, Hey, th...|
|       spam|WINNER!! As a val...|        1.0|[WINNER!!, As, a,...|
|       spam|Had your mobile 1...|        1.0|[Had, your, mobil...|
|       spam|SIX chances to wi...|        1.0|[SIX, chances, to...|
|       spam|URGENT! You have ...|        1.0|[URGENT!, You, ha...|
|       spam|XXXMobileMovieClu...|        1.0|[XXXMobileMovieCl...|
|       spam|England v Macedon...|        1.0|[England, v, Mace...|
|       spam|Thanks for your s...|        1.0|[Thanks, for, you...|
|       spam|07732584351 - Rod...|        1.0|[07732584351, -, ...|
|       spam|SMS. ac Sptv: The...|        1.0|[SMS., ac, Sptv:,...|
|       spam|Congrats! 1 year ...|        1.0|[Congrats!, 1, ye...|
|       spam|As a valued custo...|        1.0|[As, a, valued, c...|
|       spam|Urgent UR awarded...|        1.0|[Urgent, UR, awar...|
|       spam|Did you hear abou...|        1.0|[Did, you, hear, ...|
|       spam|Please call our c...|        1.0|[Please, call, ou...|
|       spam|Your free rington...|        1.0|[Your, free, ring...|
|       spam|GENT! We are tryi...|        1.0|[GENT!, We, are, ...|
|       spam|You are a winner ...|        1.0|[You, are, a, win...|
|       spam|PRIVATE! Your 200...|        1.0|[PRIVATE!, Your, ...|
+-----------+--------------------+-----------+--------------------+
only showing top 20 rows
```

Figure 11: Dataset showing that the label has been converted into double value

```
newColumnsaddedDF.groupBy("labelDouble").count().show();
```



```
+-----------+-----+
|labelDouble|count|
+-----------+-----+
|        0.0| 4827|
|        1.0|  747|
+-----------+-----+
```

Figure 12: showing the Dataset statistics after manipulations

More about Dataset manipulation

This section will describe how to use SQL queries on DF and different way to create Datasets across the datasets. Mainly running the SQL queries on DataFrame and the creating DataFrame from the JavaBean will be discussed in this section. However, interested readers can refer Spark programing guidelines for SQL operation in [3].

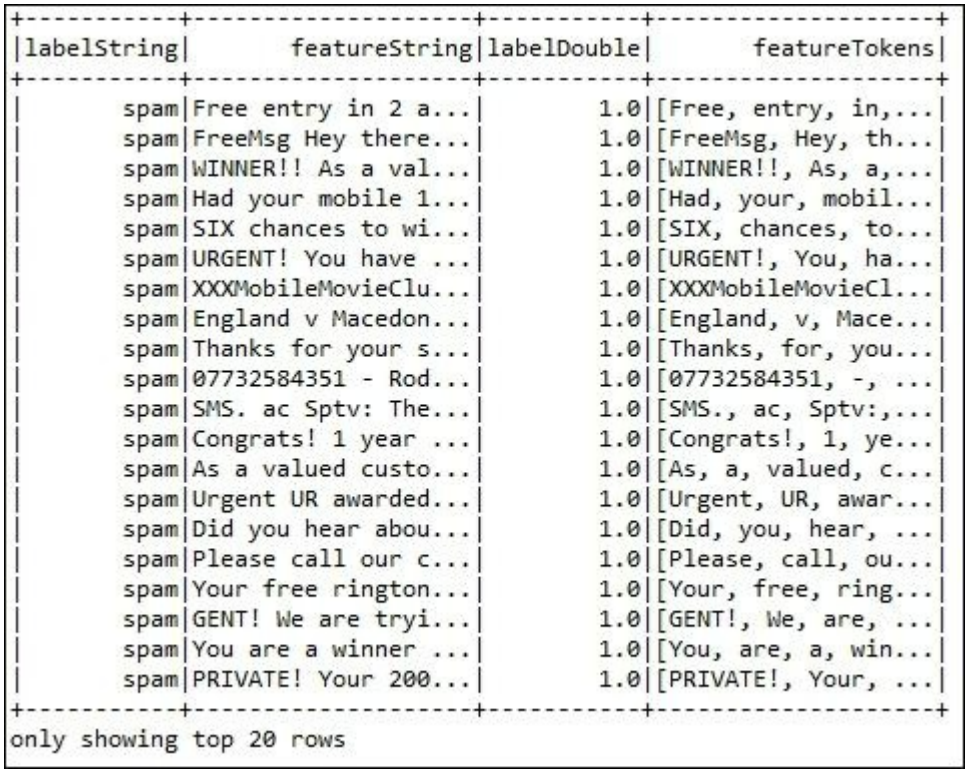
Running SQL queries on Dataset

The `SQLContext` of Spark has `sql` method enables applications to run SQL queries. This method returns a DataFrame as a result:

- [`FilternewColumnsAddedDF.createOrReplaceTempView(SMSSpamCollection)`]:

```
Dataset<Row> spam =
spark.sqlContext().sql("SELECT * FROM
SMSSpamCollection
WHERE labelDouble=1.0");
spam.show();
```

The following is the output of the preceding code:



labelString	featureString	labelDouble	featureTokens
spam	Free entry in 2 a...	1.0	[Free, entry, in,...
spam	FreeMsg Hey there...	1.0	[FreeMsg, Hey, th...
spam	WINNER!! As a val...	1.0	[WINNER!!, As, a,...
spam	Had your mobile 1...	1.0	[Had, your, mobil...
spam	SIX chances to wi...	1.0	[SIX, chances, to...
spam	URGENT! You have ...	1.0	[URGENT!, You, ha...
spam	XXXMobileMovieClu...	1.0	[XXXMobileMovieCl...
spam	England v Macedon...	1.0	[England, v, Mace...
spam	Thanks for your s...	1.0	[Thanks, for, you...
spam	07732584351 - Rod...	1.0	[07732584351, -, ...
spam	SMS. ac Sptv: The...	1.0	[SMS., ac, Sptv:,...
spam	Congrats! 1 year ...	1.0	[Congrats!, 1, ye...
spam	As a valued custo...	1.0	[As, a, valued, c...
spam	Urgent UR awarded...	1.0	[Urgent, UR, awar...
spam	Did you hear abou...	1.0	[Did, you, hear, ...
spam	Please call our c...	1.0	[Please, call, ou...
spam	Your free rington...	1.0	[Your, free, ring...
spam	GENT! We are tryi...	1.0	[GENT!, We, are, ...
spam	You are a winner ...	1.0	[You, are, a, win...
spam	PRIVATE! Your 200...	1.0	[PRIVATE!, Your, ...

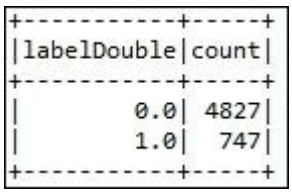
only showing top 20 rows

Figure 13: using SQL query to retrieve same result as Figure 11

- Count:

```
        Dataset<Row> counts =
sqlContext.sql("SELECT labelDouble, COUNT(*)
        AS count FROM SMSSpamCollection GROUP
BY labelDouble");
        counts.show();
```

Output:



labelDouble	count
0.0	4827
1.0	747

Figure 14: Showing the Dataset statistics

Creating Dataset from the Java Bean

You can create Dataset from a Java Bean; where you don't need to define the schemas programmatically. For example, you can see **Plain Old Java Object** (**POJO**) named as Bean in the following code:

```
public class SMSSpamBean implements Serializable
{
```

```
  private String labelString;
  private String featureString;
public SMSSpamBean(String labelString, String
featureString) {
    super();
    this.labelString = labelString;
    this.featureString = featureString;
  }
  public String getLabelString() {
    return labelString;
  }
  public void setLabelString(String labelString)
{
    this.labelString = labelString;
  }
  public String getFeatureString() {
    return featureString;
  }  public void setFeatureString(String
featureString) {    this.featureString =
featureString;
  }}
```

Create DF:

```
JavaRDD<SMSSpamBean> smsSpamBeanRDD =
rowRDD.map(new Function<Row, SMSSpamBean>() {
      @Override
    public SMSSpamBean call(Row r) throws
Exception {
        String[] split =
r.getString(0).split("\t");
        return new
SMSSpamBean(split[0],split[1]);
      }});
Dataset<Row> SMSSpamDF =
spark.sqlContext().createDataFrame(smsSpamBeanRDD,
SMSSpamBean.class);
SMSSpamDF.show();
```

The following output is as follows:



featureString	labelString
Go until jurong p...	ham
Ok lar... Joking ...	ham
Free entry in 2 a...	spam
U dun say so earl...	ham
Nah I don't think...	ham
FreeMsg Hey there...	spam
Even my brother i...	ham
As per your reque...	ham
WINNER!! As a val...	spam
Had your mobile 1...	spam
I'm gonna be home...	ham
SIX chances to wi...	spam
URGENT! You have ...	spam
I've been searchi...	ham
I HAVE A DATE ON ...	ham
XXXMobileMovieClu...	spam
Oh k...i'm watchi...	ham
Eh u remember how...	ham
Fine if thats th...	ham
England v Macedon...	spam

only showing top 20 rows

Figure 15: Corresponding feature and label string

Dataset from string and typed class

As already mentioned that the Dataset is a typed and immutable collection of objects. Datasets are basically mapped to a relational schema. With the Dataset abstraction, a new concept has been brought in Spark called an encoder. The encoder helps in entity conversion for example conversion between the JVM objects and the corresponding tabular representation. You will find this API quite similar to RDDs transformations such as `map, mapToPair, flatMap` or `filter.`

We will show the spam filter example using Datasets API in the following section. It reads the text file using and returns a Dataset as a tabular format. Then perform map transformation like RDDs for making (label, tokens) columns with adding an additional encoder parameter. Here, we have used the bean encoder with `SMSSpamTokenizedBean` class.

In this sub-section, we will show how to create Dataset from string and typed class `SMSSpamTokenizedBean`. Let's create the Spark session at first place as follows:

```
static SparkSession spark =
SparkSession.builder()
      .appName("DatasetDemo")
      .master("local[*]")
      .config("spark.sql.warehouse.dir",
"E:/Exp/")
      .getOrCreate();
```

Now create a new Dataset of type String from the `smm` filtering Dataset that means `Dataset<String>` and show the result as follows:

```
Dataset<String> ds =
spark.read().text("input/SMSSpamCollection.txt").as(org.apache.spark.sql.
ds.show();
```

Here is the output of the preceding code:



```
+--------------------+
|               value|
+--------------------+
|ham    Go until juro...|
|ham    Ok lar... Jok...|
|spam   Free entry i...|
|ham    U dun say so ...|
|ham    Nah I don't t...|
|spam   FreeMsg Hey ...|
|ham    Even my broth...|
|ham    As per your r...|
|spam   WINNER!! As ...|
|spam   Had your mob...|
|ham    I'm gonna be ...|
|spam   SIX chances ...|
|spam   URGENT! You ...|
|ham    I've been sea...|
|ham    I HAVE A DATE...|
|spam   XXXMobileMov...|
|ham    Oh k...i'm wa...|
|ham    Eh u remember...|
|ham    Fine if that...|
|spam   England v Ma...|
+--------------------+
only showing top 20 rows
```

Figure 16: Showing the snapshot of the spam filtering dataset using Dataset

Now let's create a second Dataset from the typed class `SMSSpamTokenizedBean` by mapping the Dataset of string we created immediate before as follows:

```
Dataset<SMSSpamTokenizedBean> dsSMSSpam = ds.map(
new MapFunction<String, SMSSpamTokenizedBean>() {
          @Override
public SMSSpamTokenizedBean call(String value)
throws Exception {
      String[] split = value.split("\t");
      double label;
      if(split[0].equalsIgnoreCase("spam"))
          label = 1.0;
      else
          label=0.0;
ArrayList<String> tokens = new ArrayList<>();
  for(String s:split)
    tokens.add(s.trim());
      return new SMSSpamTokenizedBean(label,
tokens.toString());
         }
},
org.apache.spark.sql.Encoders.bean(SMSSpamTokenizedBean.class));
```

Now let's print the Dataset along with its schema as follows:

```
dsSMSSpam.show();
```

```
dsSMSSpam.printSchema();
```

The following output is:

```
+-----------+--------------------+
|labelDouble|              tokens|
+-----------+--------------------+
|        0.0|[ham, Go until ju...|
|        0.0|[ham, Ok lar... J...|
|        1.0|[spam, Free entry...|
|        0.0|[ham, U dun say s...|
|        0.0|[ham, Nah I don't...|
|        1.0|[spam, FreeMsg He...|
|        0.0|[ham, Even my bro...|
|        0.0|[ham, As per your...|
|        1.0|[spam, WINNER!! A...|
|        1.0|[spam, Had your m...|
|        0.0|[ham, I'm gonna b...|
|        1.0|[spam, SIX chance...|
|        1.0|[spam, URGENT! Yo...|
|        0.0|[ham, I've been s...|
|        0.0|[ham, I HAVE A DA...|
|        1.0|[spam, XXXMobileM...|
|        0.0|[ham, Oh k...i'm ...|
|        0.0|[ham, Eh u rememb...|
|        0.0|[ham, Fine if tha...|
|        1.0|[spam, England v ...|
+-----------+--------------------+
only showing top 20 rows

root
 |-- labelDouble: double (nullable = true)
 |-- tokens: string (nullable = true)
```

Figure 17: Showing the token and label and the lower side the schema

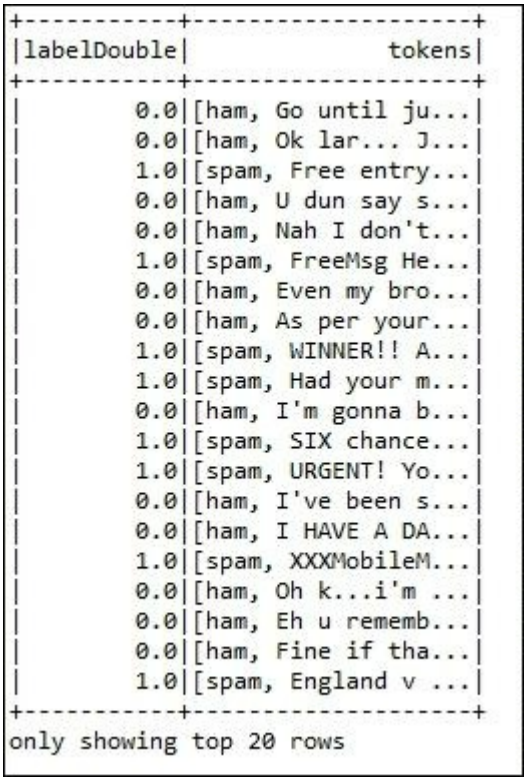
Now if you would like to convert this typed Dataset as type Row then you can use the `toDF()` method and to further create a temporary view out of the new `Dataset<Row>` you can use the `createOrReplaceTempView()` method with ease as follows:

```
Dataset<Row> df = dsSMSSpam.toDF();
df.createOrReplaceTempView("SMSSpamCollection");
```

Similarly, might want to view the same Dataset by calling show `method()` as follows:

```
df.show();
```

Output:



labelDouble	tokens
0.0	[ham, Go until ju...
0.0	[ham, Ok lar... J...
1.0	[spam, Free entry...
0.0	[ham, U dun say s...
0.0	[ham, Nah I don't...
1.0	[spam, FreeMsg He...
0.0	[ham, Even my bro...
0.0	[ham, As per your...
1.0	[spam, WINNER!! A...
1.0	[spam, Had your m...
0.0	[ham, I'm gonna b...
1.0	[spam, SIX chance...
1.0	[spam, URGENT! Yo...
0.0	[ham, I've been s...
0.0	[ham, I HAVE A DA...
1.0	[spam, XXXMobileM...
0.0	[ham, Oh k...i'm ...
0.0	[ham, Eh u rememb...
0.0	[ham, Fine if tha...
1.0	[spam, England v ...

only showing top 20 rows

Figure 18: Corresponding labels and tokens. Labels are converted into double value

Now let's explore the typed class `SMSSpamTokenizedBean`. The class works as a Java tokenized bean class for the labeling the texts. More technically, the class takes the input then it sets the labels and after that gets the labels. Secondly, it also sets and gets the token for spam filtering. Including the setter and methods, here is the class:

```
public class SMSSpamTokenizedBean implements
Serializable {
private Double labelDouble;
private String tokens;
public SMSSpamTokenizedBean(Double labelDouble,
String tokens) {
  super();
  this.labelDouble = labelDouble;
  this.tokens = tokens;
  }
  public Double getLabelDouble() {
    return labelDouble;
  }
  public void setLabelDouble(Double labelDouble)
{
    this.labelDouble = labelDouble;
  }
  public String getTokens() {
    return tokens;
  }
  public void setTokens(String tokens) {
    this.tokens = tokens;
```

```
}
}
```

Comparison between RDD, DataFrame and Dataset

There are some objectives to bring Dataset as a new Data Structure of Spark. Although RDD API is very flexible, it is sometimes harder to optimize the processing. On the other hand, the DataFrame API is very easier to optimize but it lacks some of the nice features of RDD. So, the goal of the Datasets is to allow the users to easily express transformations on objects and also providing the advantages (performance and robustness) of the Spark SQL execution engine.

The Dataset can perform many operations such as sorting or shuffling without de-serializing of an object. For doing this it requires an explicit Encoder that is used to serialize the object into a binary format. It is capable of mapping the schema of a given object (Bean) to the Spark SQL type system. On the other hand, RDDs are based on run-time reflection based serialisation and the operations that change the types of object of a Dataset also need an encoder for the new type.

Spark and data scientists workflow

As already stated that, a common task for a data scientist is to select the data, data pre-processing (formatting, cleaning and sampling) and data transformation (scaling, decomposition and aggregation) the raw data into a format that can be passed into machine learning models to build the models. As the size of the experimental datasets increases, the traditional single-node databases will not be feasible to handle these kinds of datasets, therefore, you need to switch a big data processing computing like Spark. Fortunately, we have the Spark to be an excellent option as a scalable distributed computing system to coup with your datasets.

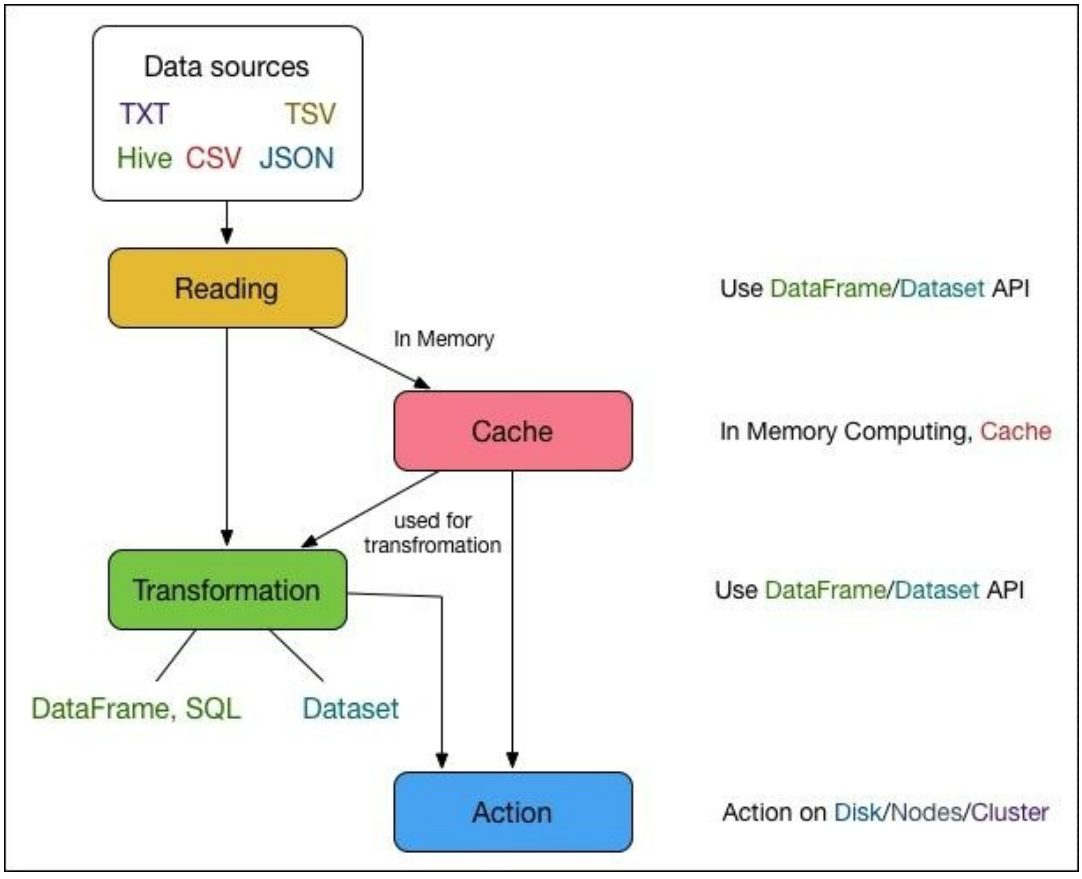


Figure-19: Data scientist's workflow for using the Spark

Now let's move to the exact point, as a data scientist at first you will have to read the Dataset available in diverse formats. Then reading the Datasets will provide you with the concept of RDDs, DataFrames and Datasets that we already describe. You can cache the Dataset into the main memory; you can transform the read data sets from the DataFrame, SQL or as Datasets. And finally, you will perform an action to dump your data to the disks, computing nodes or clusters. The step what we describe here essentially forms a workflow that you will follow for the basic data processing using Spark that showed in *Figure 1*.

Deeper into Spark

In this section, we will show you the advanced features of Spark including the use of shared variables (both the broadcast variables and accumulators) and their underlying concept will be discussed. However, we will discuss the data partition in later chapters.

Shared variables

The concept of shared variables in the context programming is not new. The variables that are required to use by many functions, and methods in parallel are called shared variables. Spark has some mechanism to use or implement the shared variables. In spark, the functions are passed to a spark operation like a map or reduce is executed on remote cluster nodes. The codes or functions work as a separate copy of variables on the nodes and no updates of the results are propagated back to the driver program. However, Spark provides two types of shared variables for two common usage patterns: broadcast variables and accumulators.

Broadcast variables

Broadcast variables provide the facility to persist a read-only to be variable cached on local machine rather than sending a copy to the computing nodes or driver program. Providing the copy of large input Dataset to every node in an efficient manner of spark. It also reduces the communication cost because Spark uses an efficient broadcast. Broadcast variables can be created from a variable `v` by calling `SparkContext.broadcast(v)`. The following code shows this:

```
Broadcast<int[]>
broadcastVariable=sc.broadcast(new int[]
{2,3,4});
int[] values = broadcastVariable.value();
for(int i:values){
  System.out.println(i);}
```

Accumulators

Accumulators are another shared variable can be used to implement counters (as in MapReduce) or sums. Spark provides the supports for accumulators to be of numeric types only. However, you can also add support for new data types using existing techniques [1]. It is created from an initial value say `val` by calling:

```
SparkContext. accumulator(val)
```

The following code shows the uses of accumulator for adding the elements of an array:

```
Accumulator<Integer> accumulator =
sc.accumulator(0);
sc.parallelize(Arrays.asList(1, 5, 3, 4))
.foreach(x -> accumulator.add(x));
System.out.println(accumulator.value());
```

There are many types and method in the Spark APIs needed to be known. However, more and details discussion is out of the scope of this book.

Tip

Interested readers should refer Spark and related materials on the following web pages:

Spark programming guide: http://spark.apache.org/docs/latest/programming-guide.html.

Spark RDD operation: http://spark.apache.org/docs/latest/programming-guide.html#rdd-operations.

Spark SQL operation: http://spark.apache.org/docs/latest/sql-programming-guide.html.

Summary

In this Chapter, we applied the basic data manipulations with RDDs, Dataset and DataFrame APIs. We also learn how to do some complex data manipulation through these APIs. We tried to focus on data manipulations, to understand a practical machine learning problem Spam-filtering. In addition to these, we showed how to read the data from different sources. Analyzing and preparing your data to understand the spam filtering as an example.

However, we did not develop any complete machine learning application, since our target was just to show you the basic data manipulation on the experimental Datasets. We intended to develop complete ML application in Chapter 6, *Building Scalable Machine Learning Pipelines*.

Which features should be used to create a predictive model is not only a vital question but also a difficult question that may require deep knowledge of the problem domain to be answered. It is possible to automatically select those features in data that are most useful or most relevant for the problem someone is working on. Considering these questions, the next chapter covers the feature engineering in detail, explaining the reasons why to apply it along with some best practices in feature engineering. Some topics which are still unclear will be clearer in the next chapter.

Chapter 4. Extracting Knowledge through Feature Engineering

Which features should be used to create a predictive model is not only a vital question, but also a difficult question that may require deep knowledge of the problem domain to be answered. It is possible to automatically select those features in data that are most useful or most relevant for the problem someone is working on. Considering these questions, this chapter covers Feature Engineering in detail, explaining the reasons why to apply it along with some best practices in feature engineering.

In addition to this, we will provide the theoretical descriptions and examples of feature extraction, transformations, and selection applied in large scale machine learning techniques, using both Spark MLlib and Spark ML APIs. Furthermore, this chapter also covers the basic idea of advanced feature engineering (also known as extreme feature engineering).

Please note that you will require having R and RStudio installed on your machine prior to proceeding with this chapter since an example towards exploratory data analysis will be shown using R.

In a nutshell, the following topics will be covered throughout this chapter:

- The state of the art of feature engineering
- Best practices in feature engineering
- Feature engineering with Spark
- Advanced feature engineering

The state of the art of feature engineering

Even though feature engineering is an informal topic, however, it is considered as an essential part in applied machine learning. Andrew Ng, who is one of the leading scientists in the area of machine learning, defined the term feature engineering in his book *Machine Learning and AI via Brain simulations* (see also, feature engineering defined at: https://en.wikipedia.org/wiki/Andrew_Nghttps://en.wikipedia.org/wiki/Andrew_Ng) as follows:

> *Coming up with features is difficult, time-consuming, requires expert knowledge. Applied machine learning is basically feature engineering.*

Based on the preceding definition, we can argue that feature engineering is actually human intelligence, not artificial intelligence. Moreover, we will explain what feature engineering is from other perspectives. Feature engineering also can be defined as the process of converting raw data into useful features (also often called feature vectors). The features help you in better representation of the underlying problem to the predictive models eventually; so that the predictive modeling can be applied to new data types to avail high predictive accuracy.

Alternatively, we can define the term feature engineering as a software engineering process of using or reusing someone's advanced domain knowledge about the underlying problem and the available data to create features, which makes the machine learning algorithms work with ease.

This is how we define the term feature engineering. If you read it carefully, you will see four dependencies in these definitions:

- The problem itself
- The raw data you will be working with to find out useful patterns or features
- The type of the machine learning problem or classes
- The predictive models you'll be using

Now based on these four dependencies, we can conclude a workflow out of this. First, you have to understand your problem itself, then you have to know your data and if it is in good order, if not, process your data to find a certain pattern or features so that you can build your model.

Once you have identified the features, you need to know which categories your problem falls under. In other words, you have to be able to identify if it is a classification, clustering, or a regression problem based on the features. Finally, you will build the model to make a prediction on the test set or validation set using a well-known method such as random forest or **Support Vector Machine** (**SVMs**).

Throughout this chapter, you will see and argue that feature engineering is an art that deals with uncertain and often unstructured data. It's also true that there are many well-defined procedures of applying the classification, clustering, regression model, or methods such as SVMs that are both methodical and provable; however, the data is a variable and comes often with a variety of characteristics at different times.

Feature extraction versus feature selection

You will get to know when and how you might be good at deciding which procedures to be followed by practice from the empirical apprenticeship. The main tasks involved in feature engineering are:

- **Data exploration and feature extraction**: This is the process of uncovering the hidden treasure in the raw data. Generally, this process does not vary much by algorithms consuming the features. However, a better understanding of the hands-on experience, business domain, and intuition play a vital role in this regard.
- **Feature selection**: This is the process for deciding which features to be selected based on the machine learning problem you are dealing with. You can use diverse techniques for selecting the features; however, it may vary in algorithms and using the features.

Importance of feature engineering

When the ultimate goal is to achieve the most accurate and reliable results from a predictive model, you have to invest your best in what you have. The best investment, in this case, would be the three parameters: time and patience, data and availability, and best algorithm. However, *how do you get the most valuable treasures out of your data for the predictive modeling?* is the problem that the process and practice of feature engineering solves in an emerging way.

In fact, the success of most of the machine learning algorithms depends on how you properly and intelligently utilize value and present your data. It is often agreed that the hidden treasure (that is, features or patterns) out of your data will directly stimulate the results of the predictive model.

Therefore, better features (that is, what you extract and select from the datasets) mean better results (that is, the results you will achieve from the model). However, please remember one thing before you generalize the earlier statement for your machine learning model, you need a great feature which is true nonetheless with the properties that describe the structures inherent in your data.

In summary, better features signify three pros: flexibility, tuning, and better results:

- **Better features (better flexibility)**: If you are successful in extracting and selecting the better features, you will get better results for sure, even if you choose a non-optimal or wrong model. In fact, optimal or most suitable models can be selected or picked up based on the good structure of the original data you have. In addition to this, good features will allow you to use less complex but efficient, faster, easily understandable, and easy to maintain models eventually.

- **Better features (better tuning)**: As we already stated, if you do not choose your machine learning model intelligently or if your features are not in good shape, you are more likely to get worse results out of the ML model. However, even if you choose some wrong parameters during building the model and if you do have some well-engineered features, still you can expect better results out of the model. Furthermore, you don't need to worry much or even work harder to choose the most optimal models and related parameters. The reason is simple, which is the good feature, you have actually understood the problem well and ready to use the better represented by all the data by characterizing the problem itself.
- **Better features (better results)**: You are most likely to get better results even if you spent most of your efforts in feature engineering towards better features selections.

We also suggest readers not to be overconfident with only the features. The preceding statements are often true; however, sometimes they are misleading. We would like to clear the preceding statements further. Actually, if you receive the best predictive results from a model, it is actually of three factors: the model you selected, the data you had, and the features you had prepared.

Therefore, if you have enough time and computational resources, always try to use the standard model since often the simplicity does not imply better accuracy. Nonetheless, better features will contribute the most out of these three factors. One thing you should know is that, unfortunately, even if you master feature engineering emanates with many hands-on practices, and research what others are doing well in the state of the arts, some machine learning projects fail at the very end.

Feature engineering and data exploration

Very often, an intelligent choice for both training and test samples out of better features leads to better solutions. Although in the previous section we argued that there are two tasks in the feature engineering: feature extraction from the raw data and feature selection. However, there is no definite or fixed path for feature engineering.

Conversely, the whole step in feature engineering is very much directed by the available raw data. If the data is well-structured you would be feeling lucky. Nonetheless, the reality is often that the raw data comes from diverse sources in multiple formats. Therefore, exploring this data is very important before you proceed to feature extraction and feature selection.

Tip

We suggest you to figure out the data skewness and kurtosis using the histogram and outliers using the box-plot and bootstrapping the data using Data Sidekick techniques (introduced by Abe Gong) in the literature (see at: https://curiosity.com/paths/abe-gong-building-for-resilience-solid-2014-keynote-oreilly/#abe-gong-building-for-resilience-solid-2014-keynote-oreilly).

The following questions need to be answered and known by means of data exploration before applying the feature engineering:

- What is the percentage of the total data being present or not having null or missing values for all the available fields? Then try to handle those missing values and interpret them well without losing the data semantics.
- What is the correlation between the fields? What is the correlation of each field with the predicted variable? What values do they take (that is, categorical or non-categorical, numerical or alpha-numerical, and so on)?
- Then find out if the data distribution is skewed or not. You can identify the skewness by seeing the outliers or long tail (slightly skewed to the right or positively skewed, slightly skewed to the left or negatively skewed, as shown in *Figure 1*). Now identify if the outliers contribute towards making the prediction or not.
- After that, observe the data kurtosis. More technically, check if your kurtosis is mesokurtic (less than but almost equal to 3), leptokurtic (more than 3), or platykurtic (less than 3). Note, the kurtosis of any univariate normal distribution is considered to be 3.
- Now play with the tail and observe (do the predictions get better?) what happens when you remove the long tail?

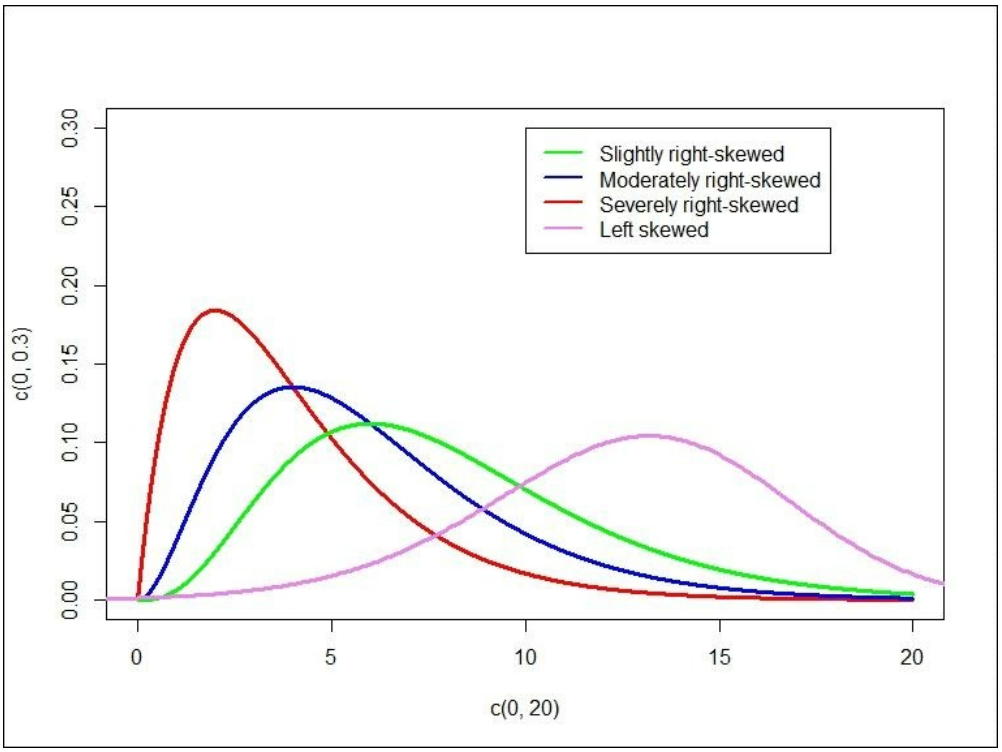


Figure 1: Skewness of the data distribution (x-axis = data, y-axis = density).

You can use simple visualization tools such as density plots for doing this, as explained by the following example.

Example 1. Suppose you are interested in fitness walking and you walked at a sports ground or countryside in the last four weeks (excluding the weekends). You spent the following time (in minutes to finish a 4 KM walking track):15, 16, 18, 17.16, 16.5, 18.6, 19.0, 20.4, 20.6, 25.15, 27.27, 25.24, 21.05, 21.65, 20.92, 22.61, 23.71, 35, 39, and 50. Now let's compute and interpret the skewness and kurtosis of these values using R.

Tip

We will show how to configure and work with SparkR in Chapter 10, *Configuring and Working with External Libraries* and show how to execute the same code on SparkR. The reason behind this is some plotting packages such as `ggplot2` are still not implemented in the current version of Spark used for SparkR directly. However, the `ggplot2` is available as a combined package named `ggplot2.SparkR` on GitHub, which can be installed and configured using the following command:

```
devtools::install_github("SKKU-SKT/ggplot2.SparkR")
```

However, there are numerous dependencies that need to be ensured before and during the configuration process. Therefore, we should resolve this issue in a later chapter instead. For the time being, we assume you have basic knowledge of using R and if you have R installed and configured on your computer then please use the following steps. However, a step-by-step example on how to install and configure SparkR using RStudio will be shown in Chapter 10, *Configuring and Working with External Libraries*.

Now just copy the following code snippets and try to execute to make sure you have the correct value of the Skewness and Kurtosis.

Install the `moments` package for calculating Skewness and Kurtosis:

```
install.packages("moments")
```

Use the `moments` package:

```
library(moments)
```

Make a vector for the time you have taken during the workout:

```
time_taken <- c (15, 16, 18, 17.16, 16.5, 18.6,
19.0, 20.4, 20.6, 25.15, 27.27, 25.24, 21.05,
21.65, 20.92, 22.61, 23.71, 35, 39, 50)
```

Convert the time into DataFrame:

```
df<- data.frame(time_taken)
```

Now calculate the `skewness`:

```
skewness(df)
[1]1.769592
```

Now calculate the `kurtosis`:

```
> kurtosis(df)
[1]5.650427
```

Interpretation of the result: The skewness of your workout time is 1.769592, which means your data is skewed to the right or positively skewed. The kurtosis, on the other hand, is 5.650427, which means the distribution of the data is leptokurtic. Now to check the outliers or tails check

the following histogram. Again, for simplicity, we will use R to plot the density plot that will interpret your workout time.

Install `ggplot2package` for plotting the histogram:

```
install.packages("ggplot2")
```

Use the `moments` package:

```
library(ggplot2)
```

Now plot the histogram using the `qplot()` method of `ggplot2`:

```
ggplot(df, aes(x = time_taken)) +
stat_density(geom="line", col=
"green", size = 1, bw = 4) + theme_bw()
```

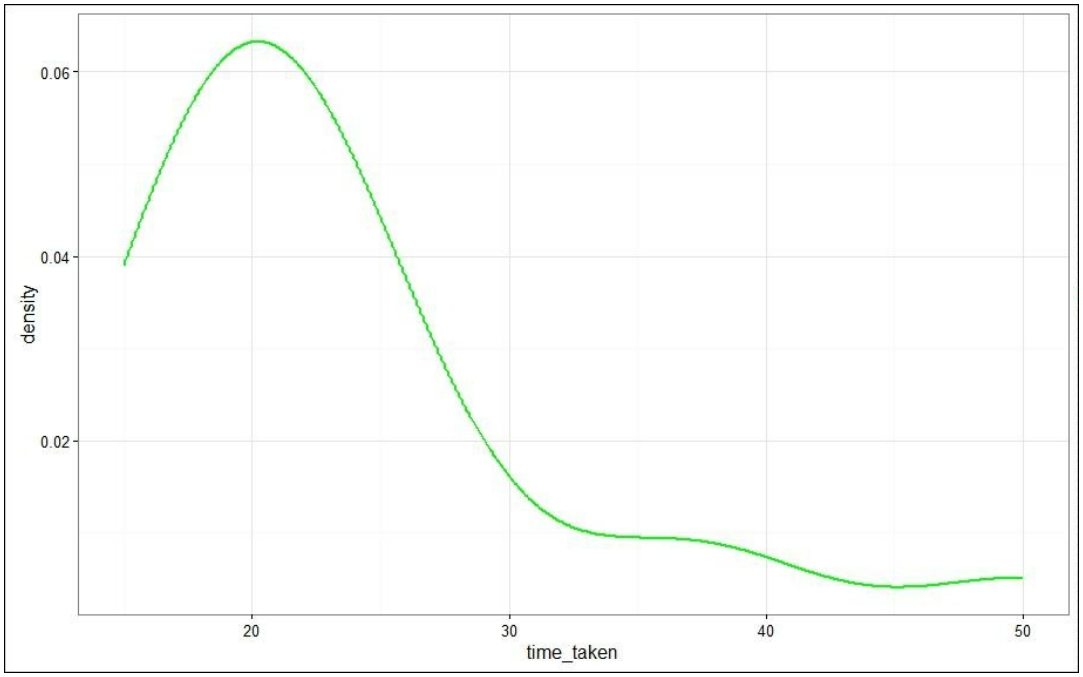


Figure 2. Histogram of the workout time (right-skewed).

The interpretation presented in *Figure 2* of the distribution of data (workout times) shows the density plot is skewed to the right so is leptokurtic. Besides the density plot, you can also look at the box-plots for each individual feature. Where the box plot displays the data distribution based on five-number summaries: **minimum**, **first quartile**, median, **third quartile**, and **maximum**, as shown in *Figure 3*, where we can look for outliers beyond three (3) **Inter-Quartile Range** (**IQR**):

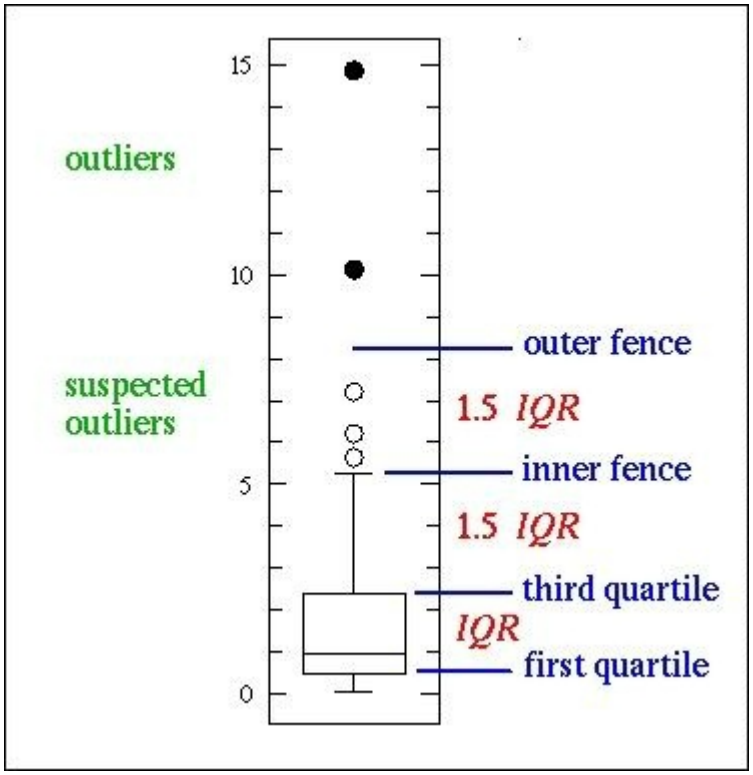


Figure 3. Histogram of the workout time (figure courtesy of Box Plot: Display of Distribution, http://www.physics.csbsju.edu/stats/box2.htmlhttp://www.physics.csbsju.edu/stats/box2.html).

Bootstrapping the datasets also sometimes offers insights on outliers. If the data volume is too large (that is, big data) doing the Data Sidekick, evaluations and predictions are also useful. The idea of Data Sidekick is to use a small part of the available data to figure out what insights can be concluded from the datasets and it is also commonly referred to as *using small data to multiply the value of big data*.

It is very useful for large-scale text analytics. For example, suppose you have a huge corpus of text, and of course you can use a small portion of it to test various sentiment analysis models and choose the one which gives the best results in terms of performance (computation time, memory usage, scalability, and throughput).

Now we would like to draw your attention to the other aspects of feature engineering. Moreover, converting continuous variables into categorical variables (with a certain combination of features) results in better predictor variables.

Tip

In statistical language, a variable in your data either represents measurements on some continuous scale, or on some categorical or discrete characteristics. For example, weight, height, and age of an athlete would represent the continuous variables. Alternatively, the

survival or failure in terms of time is also considered as continuous variables. A person's gender, occupation, or marital status, on the other hand, is categorical or discrete variables. Statistically, some variables could be considered in either way. For example, a movie viewer's rating of a move on a 10 point scale may be considered a continuous variable, or we may consider it as a discrete variable with 10 categories. Time series data or real-time streaming data are usually collected for continuous variables until a certain time.

In parallel, considering the square or cube or even using the non-linear models of the features can also provide better insights. Also, consider the forward selection or backwards selection wisely since both of them are computationally expensive.

Finally, when the number of features becomes significantly large it is a wise decision to use the **Principal Component Analysis** (**PCA**) or **Singular Value Decomposition** (**SVD**) technique to find the right combination of features.

Feature extraction – creating features out of data

Feature extraction is the automatic way of constructing new features from the raw data you have or will be collecting. During the feature extraction process, reducing the dimensionality of complex raw data is usually done by making the observation into a much smaller set automatically that can be modeled into later stages. Projection methods such as PCA and unsupervised clustering methods are used for tabular data in TXT, CSV, TSV, or RDB format. However, feature extraction from another data format is very complex. Specially parsing many data formats such as XML and SDRF is a tedious process if the number of fields to extract is huge.

For multimedia data such as image data, the most common type of technique includes line or edge detection or image segmentation. However, subject to the domain and image, video and audio observations advance themselves to many of the same types of **Digital Signal Processing** (**DSP**) methods where typically the analogue observations are stored in digital formats.

The most positive pros and the key to feature extraction are that the methods that are developed and available are automatic; therefore, thay can solve the problem of unmanageable high dimensional data. As we stated in Chapter 3, *Understanding the Problem by Understanding the Data* that more data exploration and better feature extraction eventually increases the performance of your ML model (since feature extraction also involves feature selection). The reality is more data will provide more

insights towards the performance of the predictive models eventually. However, the data has to be useful and dumping unwanted data will kill your valuable time; therefore, think of the meaning of the statement before collecting your data.

There are several steps involved in the feature extraction process; including the data transformation and feature transformation. As we stated several times, a machine learning model is likely to provide a better result if the model is well trained with better features out of the raw data. Optimized for learning and generalization is a key characteristic of good data. Therefore, the process of putting together the data in this optimal format is achieved through some data processing steps such as cleaning, missing values handling, and some intermediate transformation like from a text document to words transformation.

The methods that help to create new features as predictor variables are called feature transformation, which is actually a group of methods. Feature transformation is essentially required for the dimension reduction. Usually, when the transformed features have a descriptive dimension, it is likely to have better order compared to the original features.

Therefore, less descriptive features can be dropped from the training or test samples when building the machine learning models. The most common tasks included in the feature transformation are non-negative matrix factorization, principal component analysis, and factor analysis using scaling, decomposition, and aggregation operations.

Examples of feature extraction include the extraction of contours in images, extraction of diagrams from a text, extraction of phonemes from the recording of spoken text, and so on. Feature extraction involves a transformation of the features, which is often not reversible because some information is lost eventually in the process of dimensionality reduction.

Feature selection – filtering features from data

Feature selection is a process for preparing the training datasets or validation dataset for predictive modeling and analytics. Feature selection has practical implication in most of the machine learning problem types including classification, clustering, dimensionality reduction, collaborative filtering, regression, and so on.

Therefore, the ultimate goal is to select a subset from the large collection of features from the original data set. And often dimensionality reduction algorithms are applied, such as **Singular Value Decomposition** (**SVD**) and **Principal Component Analysis** (**PCA**).

An interesting power of the feature selection technique is that a minimal feature set can be applied to represent the maximum amount of variance in the available data. In other words, the minimal subset of the feature is enough to train your machine learning model quite efficiently.

This subset of features is used to train the model. There are two types of feature selection techniques, namely forward selection and backwards selection. The forward selection starts with the strongest feature and keeps adding more features. On the contrary, the backwards selection starts with all the features and removes the weakest features. However, both techniques are computationally expensive.

Importance of feature selection

Since not all the features are equally important; consequently, you will find some features with more importance than others for making the model more accurate. Therefore, those attributes can be treated as irrelevant to the problem. As a result, you need to remove those features before preparing the training and test sets. Sometimes, the same technique might be applied to the validation sets.

In parallel to importance, you will always find some features that will be redundant in the context of other features. Feature selection is not only involved with removing irrelevant or redundant features, it also serves other purposes that are important to increase the model's accuracy, as stated here:

- Feature selection increases the predictive accuracy of the model you are using by eliminating irrelevant, null/missing, and redundant features. It also deals with highly correlated features.
- Feature selection techniques make the model training process more robust and faster by decreasing the number of features.

Feature selection versus dimensionality reduction

Although by using the feature selection technique it is quietly possible to reduce the number of features by selecting certain features in the dataset. And later on, the subset is used to train the model. However, the entire process usually, cannot be used interchangeably with the term **dimensionality reduction**.

The reality is that the feature selection methods are used to extract a subset from the total set in the data without changing their underlying properties.

In contrast, the dimensionality reduction method, on the other hand, employs already engineered features that can transform the original

features into corresponding feature vectors by reducing the number of variables under certain considerations and requirements of the machine learning problem.

Thus, it actually modifies the underlying data, extracts the original features from raw and noisy features by compressing the data, but maintains the original structure and most of the time is irreversible. Typical examples of dimensionality reduction methods include **Principal Component Analysis** (**PCA**), **Canonical Correlation Analysis** (**CCA**), and **Singular Value Decomposition** (**SVD**).

Other feature selection techniques use the filter-based, wrapper methods and embedded methods feature selection by evaluating the correlation between each feature and the target attribute in a supervised context. These methods apply some statistical measures to assign a score to each feature also known as filter methods.

The features are then ranked based on the scoring system that can help to eliminate the specific features. Examples of such techniques are information gain, correlation coefficient scores, and Chi-squared test. An example of wrapper methods, which is a feature selection process as a search problem, is the recursive feature elimination algorithm. On the other hand, **Least Absolute Shrinkage and Selection Operator** (**LASSO**), Elastic Net, and Ridge Regression are typical examples of embedded methods of feature selection, which is also known as regularizations methods.

The current implementation of Spark MLlib provides the support for dimensionality reduction on the `RowMatrix` class only for the SVD and PCA. On the other hand, some typical steps from raw data collection to feature selection are feature extractions, feature transformation, and feature selection.

Tip

Interested readers are suggested to read the API documentation for the feature selection and dimensionality reduction at: http://spark.apache.org/docs/latest/mllib-dimensionality-reduction.html.

Best practices in feature engineering

In this section, we have figured out some good practices while performing the feature engineering on your available data. Some best practices of machine learning were described in Chapter 2, *Machine Learning Best Practices*. However, those were too general for the overall machine learning state of the arts. Those best practices, of course, would be useful in the feature engineering, too. Moreover, we will provide more concrete examples concerning feature engineering in the following sub-sections.

Understanding the data

Although the term feature engineering is more technical, however, it is an art that helps you to understand where the features come from. Now some vital questions evolve too, which need to be answered before understanding the data:

- What are the provenances of those features? Is the data real-time or coming from the static sources?
- Are the features continuous, discrete, or none?
- What is the distribution of the features? Does the distribution largely depend on what subset of examples is being considered?
- Do these features contain missing values (that is, NULL)? If so, is it possible to handle those values? Is it possible to eliminate them in the present, future, or upcoming data?
- Is there duplicate or redundant entries?
- Should we go for manual feature creation that proves to be useful? If so, how hard would it be to incorporate those features in the model training stage?
- Are there features that can be used as standard features?

Knowing the answers to the preceding questions is important. Since data provenance would help you to prepare your feature engineering techniques a bit faster. You need to know if your features are discrete or continuous or if the requests are a real-time response or not. Moreover, you need to know the data distribution along with their skewness and kurtosis to handle the outliers.

You need to be prepared for the missing or null values whether they could be removed or need to be filled with alternative values. Besides, you need to remove duplicates entries in the first place, which is extremely important, since duplicate data points might significantly affect the results of model validation if not properly excluded. Finally, you need to know your machine

learning problem itself since knowing the problem type would help you to label your data accordingly.

Innovative way of feature extraction

Be innovative while extracting and selecting the features. Here we provide eight tips altogether that will help you to generalize the same during your machine learning application development.

> **Tip**
>
> Create the input by rolling up existing data fields to a broader level or category.

To be more specific, let's give you some examples. Obviously, you can categorize your colleagues based on their title into strategic or tactical. For instance, you can code the employee with *Vice President or VP* or above as strategic and the *Director* and below could be encoded as tactical.

Collating several industries into a higher-level industry could be another example of such categorization. Collate oil and gas companies with commodity companies; gold, silver, or platinum as precious metal companies; high-tech giants and telecommunications industries as *technology*; define the companies with more than $1B revenue as *large* and *small* with net asset $1M for instance.

> **Tip**
>
> Split data into separate categories or bins.

To be more specific, let's give you some examples. Suppose you are doing some analytics on the companies with an annual that ranges from $50 M to over $1 B. Therefore, obviously, you can split the revenue into some sequential bins, such as $50-$200M, $201-$500M, $501M-$1B, and $1B+, for instance. Now how do you represent the features in a presentable format? It's so simple, try to put a value one whenever a company falls with the revenue bin; otherwise, the value is zero. There are now four new data fields created from the annual revenue field, right?

> **Tip**

Think of an innovative way to combine existing data fields into new ones.

To be more specific, let's give you some examples. In the very first tip, we argue how to create new inputs by rolling up existing fields into broader fields. Now, suppose if you want to create a Boolean flag that identifies whether someone falls in a VP or higher category with more than 10 years of experience. Therefore, in this case, you are actually creating new fields by multiplying, dividing, adding, or subtracting one data field by another.

Tip

Think about the problem at hand and be creative simultaneously.

In previous tips, suppose you have created enough bins and fields or inputs. Now, don't worry much about creating too many variables in the first place. It would be wise to just let the brainstorming flow a normal flow for the feature selection step.

Tip

Don't be a fool.

Be cautious about creating unnecessary fields; since creating too many features out of a small amount of data may overfit your model, which can lead to spurious results. When you face the data correlation, remember that correlation does not always imply causation. Our logic to this common point is that modeling observational data can only show us that two variables are related, but it cannot tell us the reason.

Research articles in the book *Freakonomics* (see also at *Steven D. Levitt, Stephen J. Dubner, Freakonomics: A Rogue Economist Explores the Hidden Side of Everything*, http://www.barnesandnoble.com/w/freakonomics-steven-d-levitt/1100550563http://www.barnesandnoble.com/w/freakonomics-steven-d-levitt/1100550563) has found that data from public school's test scores indicates that children living at home with a higher number of books have a tendency of having higher standardized test scores compared to those with a lower number of books at home.

Therefore, be cautious before creating and constructing unnecessary features, which implies that don't be a fool.

Tip

Don't over engineer.

It is trivial to judge the difference whether an iteration takes a few minutes or half a day during the feature engineering phase. Since the most productive time during the feature engineering phase is usually spent on the whiteboard. Therefore, the most productive way to make sure it is done right is to ask the right questions to your data. It's true that nowadays the term big data is taking over the term feature engineering. There is no room for hacking, so for the over engineering:

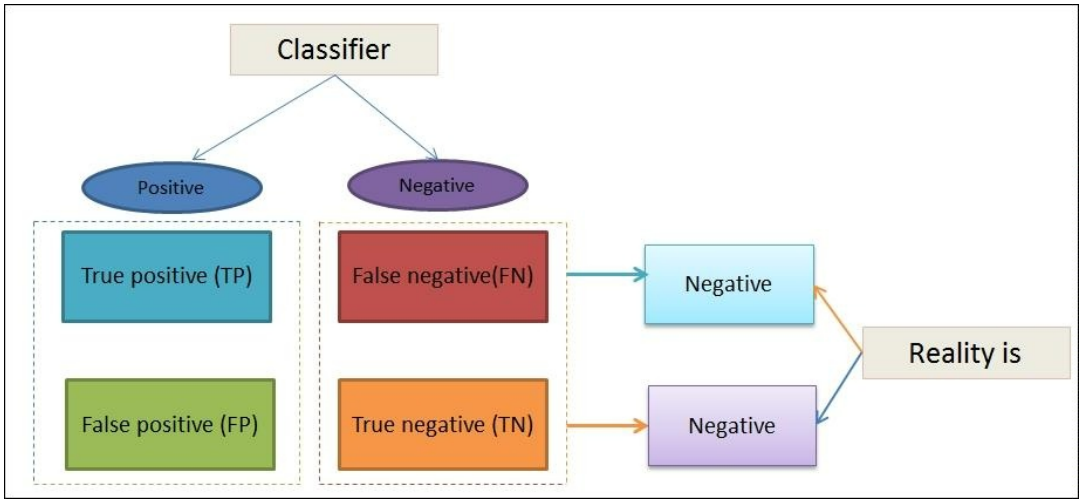


Figure 4: Real interpretation of false positive and false negative.

Tip

Beware of false positives and false negatives.

Another important aspect is comparing the false negatives and false positives. Depending on the problem, getting a higher accuracy on one or the other is important. For instance, if you are doing research in the healthcare section and trying to develop a machine learning model that will work towards the disease prediction, getting false positives might be better than getting the false negative results. Therefore, our suggestion in this regard would be to look at the confusion matrix that will help you to see the predictions made by a classifier in a visual way.

The rows indicate the true class of each observation while the columns correspond to the class predicted by the model itself, as shown in *Figure 4*. However, *Figure 5* would provide more insight. Note that the diagonal

elements, also called correct decision, are marked in bold. The last column, **Acc**, signifies the accuracy for each key as follows:

Target	Selected									Acc
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	
1	**137**	13	3	0	0	1	1	0	0	0.89
2	1	**55**	1	0	0	0	0	6	1	0.86
3	2	4	**84**	0	0	0	1	1	2	0.89
4	3	0	1	**153**	5	2	1	1	1	0.92
5	0	0	3	0	**44**	2	2	1	2	0.82
6	0	0	2	1	4	**35**	0	0	1	0.81
7	0	0	0	0	0	0	**61**	2	2	0.94
8	0	0	0	1	0	0	0	**69**	3	0.95
9	0	0	0	0	0	0	0	2	**26**	0.93
										0.89

Figure 5: A simple confusion matrix.

Tip

Think about precision and recall before selecting features.

Finally, two more important quantities to consider are the precision and recall. More technically, how often your classifier predicts a +ve outcome correctly is called recall. On the contrary, when your classifier predicts a +ve output and how often it is actually true is the precision.

It's true that it's really difficult to predict these two values. However, a careful feature selection would help you to get both the values better in the last place.

You will find more interesting and some excellent descriptions about the feature selection in a research paper written by *Matthew Shardlow* (see also at Matthew Shardlow, *An Analysis of Feature Selection Techniques*, https://studentnet.cs.manchester.ac.uk/pgt/COMP61011/goodProjects/Shardlow.pdf). Now let's have a journey to the realm of Spark's feature engineering features in the next section.

Feature engineering with Spark

Machine learning based on big data is a deep and broad area and it needs a new recipe and the ingredients would be feature engineering and stable optimization of the model out of the data. The optimized model can be called Big Models (see also at *S. Martinez*, *A. Chen*, *G. I. Webb*, and *N. A. Zaidi*, *Scalable learning of Bayesian network classifiers*, accepted to be published in *Journal of Machine Learning Research*) that can learn from big data and holds the key to a breakthrough other than big data.

Big model also signifies that your results out of diverse and complex big data would be with low bias (see at *D. Brain and G. I. Webb*, *The need for low bias algorithms in classification learning from small data sets*, *in PKDD*, *pp. 62, 73, 2002*) and out-of-core (see out-of-core learning defined at, https://en.wikipedia.org/wiki/Out-of-core_algorithm and https://en.wikipedia.org/wiki/Out-of-core_algorithm) using multi-class machine learning algorithms with minimal tuning parameters.

Spark introduces this big model for us to deploy our machine learning application at scale. In this section, we will describe how Spark developed machine learning libraries and Spark core to handle the advanced features of feature engineering for large-scale datasets and different data structures efficiently.

As we already stated, Spark's machine learning module contains two APIs including `spark.mllib` and `spark.ml`. The MLlib package is built on top of RDD, whereas the ML package is built on top of DataFrame and Dataset that provides a higher label API for constructing an ML pipeline. The next few sections will show you the details of the ML (MLlib will be discussed in Chapter 5 , *Supervised and Unsupervised Learning by Examples*) package with examples concluding with a practical machine learning problem.

Machine learning pipeline – an overview

Spark's ML package provides a uniform set of higher-level APIs that helps to create a practical machine learning pipeline. The main concept of this pipeline is to combine multiple algorithms of machine learning together to make a complete workflow. In the machine learning arena, it is often common practice to run a sequence of algorithms to process and learn from the available data.

For example, suppose you want to develop a text analytics machine learning application. The total process could be split into several stages for a collection of some simple text document. Naturally, the processing workflow might include several stages. In the very first step, you need to split the text into words from each document. Once you have the split

words, you should convert those words into numerical feature vectors for the words from each document.

Finally, you might want to learn a prediction model using the features vector you got in stage 2 and also want to label each vector to use supervised machine learning algorithms. In brief, these four stages can be summarised as follows. For each document, do the following:

- Split the texts=> words
- Convert words => numerical feature vectors
- Numerical feature vectors => labeling
- Build an ML model as a prediction model using vectors and labels

These four stages could be considered as a workflow. The Spark ML represents these kinds of workflows as pipelines that consists of a sequence of PipelineStages; where a Transformer and an Estimator contribute in each stage of the pipeline to be run in a certain order. The Transformer is actually an algorithm to transform one Dataset to another Dataset.

On the other hand, an Estimator is also an algorithm, which is liable for fitting on a Dataset to produce a Transformer. Technically, an Estimator implements a method called `fit()`, which accepts a Dataset and produces a model, which is a Transformer.

Tip

Interested readers should refer to this URL http://spark.apache.org/docs/latest/ml-pipeline.html for more details on the Transformer, an Estimator in pipelines.

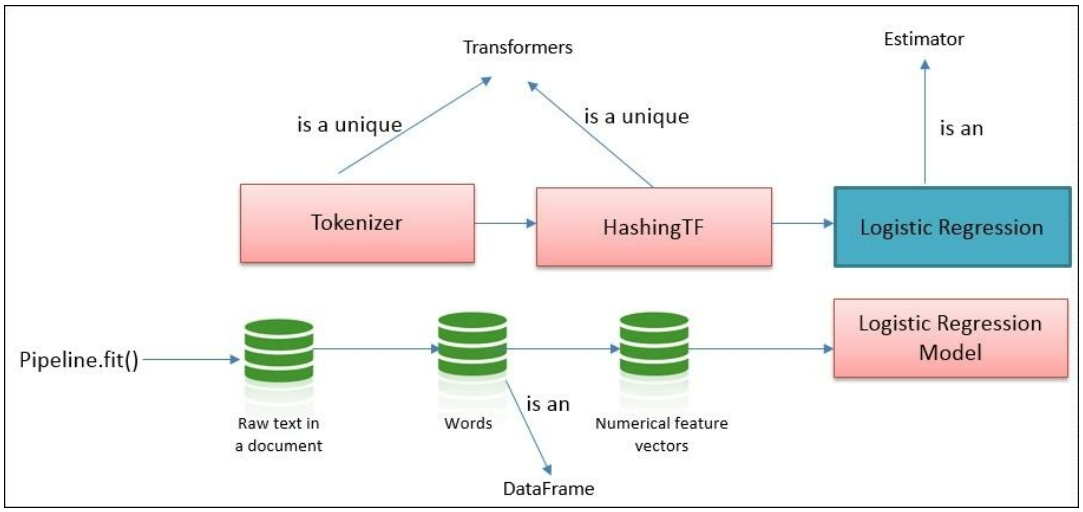


Figure 6: Pipeline is an Estimator.

To be more specific, let's draw an example, suppose a machine learning algorithm such as Logistic Regression (or the Linear Regression) is used as an Estimator. Now by calling the `fit()` method , which trains a **Logistic Regression Model** (which itself is a model, and hence a Transformer). Technically, a Transformer implements a method, namely `transform()`, which converts one Dataset into another.

During the conversion, one more column is depending upon the selection and column position. It is to be noted that the pipeline concept that Spark has developed is mostly inspired by the Scikit-learn project, which is a simple and efficient tool for data mining and data analysis (see also at Scikit-learn project, http://scikit-learn.org/stable/).

As discussed in Chapter 1, *Introduction to Data Analytics with Spark*, Spark has implemented RDD operation as **Directed Acyclic Graph** (**DAG**) style. The same fashion is also applicable on pipelining as well; wherein each DAG Pipeline, stages are specified as an ordered array. The text-processing pipeline we previously described as an example with four stages is actually a linear Pipeline; in which each stage consumes the data produced by the previous stage. It is also possible to create the non-linear pipelines as long as the data flow of the feature engineering graph forms and aligns in a DAG style.

It is to be noted that, if a Pipeline forms a DAG, then the stages need to be specified in topological order essentially. The pipeline we are talking about can be operated on top of Dataset including various file types, therefore run-time and compile-time checking from the pipelining consistencies is required. Unfortunately, the current implementation of Spark Pipeline does not provide the use compile-time type checking. However, Spark provides the run-time checking that is used by the Pipelines and PipelineModels, which is done using the Dataset schema.

Since the concept of RDD is immutable, that means once an RDD is created, it's not possible to change the contents of the RDD, similarly, uniqueness in Pipeline stages should be persistent (please refer to *Figure 6* and *Figure 7* for the clear view) with unique IDs. For simplicity, the preceding text processing workflow can be visualized as like Figure 5; where we have shown the text processing pipeline with three stages. The **Tokenizer** and **HashingTF** are two unique Transformers.

On the other hand, LogisticRegression is an Estimator. In the bottom row, a cylinder indicates a Dataset. The `fit()` method from pipeline is called on the original Dataset containing the documents of text with labels. Now the `Tokenizer.transform()` method splits the raw text documents into words and the `HashingTF.transform()` method on the other hand converts the words column into feature vectors.

Please note in each case, a column on the Dataset is added. Now the `LogisticRegression.fit()` method is called to produce a `LogisticRegressionModel`:

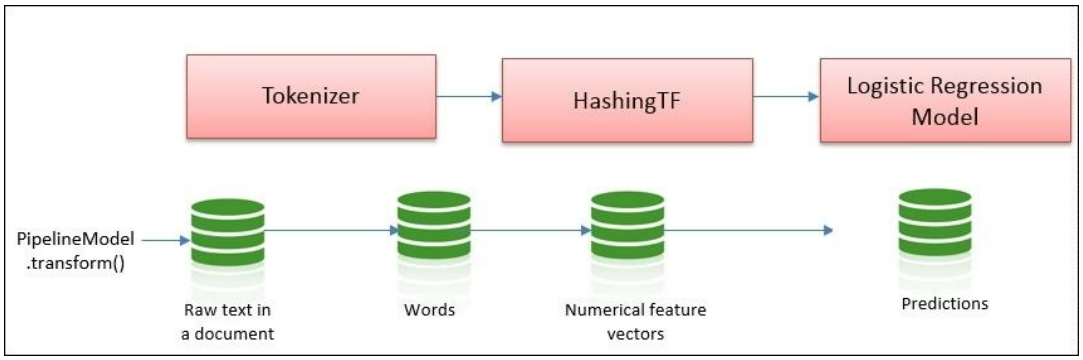


Figure 7: Pipeline is an Estimator.

In *Figure 7*, the PipelineModel has the same number of stages as the original Pipeline. However, in this case, all the Estimators from the original Pipeline need to be converted into Transformers.

When the `transform()` method from the **PipelineModel** is called on a test Dataset (that is, numeric feature vectors), the data is passed through the fitted pipeline in a certain order.

In a nutshell, Pipelines and PipelineModel help to ensure that training and test data go through identical feature processing steps. The following section shows a practical example of the preceding pipelining process we have described.

Pipeline – an example with Spark ML

This section will show a practical machine-learning problem called **Spam Filtering**, which was introduced in Chapter 3, *Understanding the Problem by Understanding the Data* with Spark's pipeline. We will use the `SMSSpamCollection` dataset downloaded from https://archive.ics.uci.edu/ml/datasets/SMS+Spam+Collection to show the feature engineering with Spark. The following code reads a sample dataset as a Dataset using a **Plain Old Java Object** (**POJO**) class (see more at https://en.wikipedia.org/wiki/Plain_Old_Java_Object). Note that the `SMSSpamHamLabelDocument` class contains the label (`label: double`) and SMS lines (`text: String`).

To run the code, just create a Maven project in your Eclipse IDE by specifying the master URL and dependencies on the provided `pom.xml` file under the Maven project and package the application as a jar file. Alternatively, run the example on Eclipse as a standalone Java application.

The code for Spark session creation is as follows:

```
  static SparkSession spark = SparkSession
      .builder().appName("JavaLDAExample")
      .master("local[*]")
      .config("spark.sql.warehouse.dir",
"E:/Exp/")
      .getOrCreate();
```

Here the Spark SQL warehouse is set to the `E:/Exp/` directory for Windows. Set your path accordingly based on the OS type.

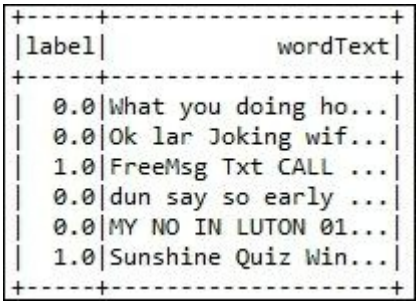
The code for the `smsspamdataset` sample is as follows:

```
public static void main(String[] args) {
 // Prepare training documents, which are
labelled.
 Dataset<Row> smsspamdataset =
spark.createDataFrame(Arrays.asList(
      new SMSSpamHamLabelDocument(0.0, "What you
doing how are you"),
      new SMSSpamHamLabelDocument(0.0, "Ok lar
Joking wif u oni"),
      new SMSSpamHamLabelDocument(1.0, "FreeMsg
Txt CALL to No 86888 & claim your reward of 3
hours talk time to use from your phone now
ubscribe6GBP mnth inc 3hrs 16 stop txtStop"),
      new SMSSpamHamLabelDocument(0.0, "dun say
so early hor U c already then say"),
      new SMSSpamHamLabelDocument(0.0, "MY NO IN
LUTON 0125698789 RING ME IF UR AROUND H"),
      new SMSSpamHamLabelDocument(1.0, "Sunshine
Quiz Win a super Sony DVD recorder if you canname
the capital of Australia Text MQUIZ to 82277 B")
    ), SMSSpamHamLabelDocument.class);
```

Now let's see the structure of the Dataset by calling the `show()` method as follows:

```
Smsspamdataset.show();
```

The output will look as follows:



label	wordText
0.0	What you doing ho...
0.0	Ok lar Joking wif...
1.0	FreeMsg Txt CALL ...
0.0	dun say so early ...
0.0	MY NO IN LUTON 01...
1.0	Sunshine Quiz Win...

The code for the POJO Class is as follows:

```
public class SMSSpamHamLabelDocument implements
Serializable {
    private double label;
    private String wordText;
    public SMSSpamHamLabelDocument(double label,
String wordText) {
      this.label = label;
      this.wordText = wordText;
    }
    public double getLabel() { return
this.label; }
    public void setLabel(double id) { this.label
= label; }
    public String getWordText() { return
this.wordText; }    public void
setWordText(String wordText) { this.wordText =
wordText; }
}  }
```

Now, let's split the dataset into `trainingData` (60%) and `testData` (40%) for the model training purpose.

The code for splits is as follows:

```
Dataset<Row>[] splits =
smsspamdataset.randomSplit(new double[] { 0.6,
0.4 });
Dataset<Row> trainingData = splits[0];
Dataset<Row> testData = splits[1];
```

The objective of the dataset is to build a predictive model using a classification algorithm, as we know from the dataset; there are two types of messages. One is spam, which is represented as 1.0, and another one is ham, represented as 0.0 labels. We can consider here the LogisticRegression or linear regression algorithm for training a model for the simplicity of training and using.

However, more complex classifiers using regression such as generalized regression will be discussed in Chapter 8, *Adapting Your Machine Learning*

Models. Consequently, our workflow or pipeline will be like the following, according to our dataset:

- Tokenize the text lines into words from the training data
- Extracting the features using the hashing technique
- Applying a LogisticRegression Estimator for building a model

The preceding three steps can be done easily by Spark's pipeline component. You can define all the stages into a single Pipeline class that will build a model in an efficient way. The following code shows the whole pipeline for building the predictive model. The Tokenizer class defines the input and output column (for example, `wordText` to words), the `HashTF` class defines how to extract the features from the words of the Tokenizer class.

The `LogisticRegression` class configures its parameter. Finally, you can see the Pipeline class that takes the preceding methods into a PipelineStage array and returns an Estimator. After applying the `fit()` method to the training set it will return the final model, which is ready for prediction. You can see the output of the test data after applying a model for predicting.

The code for Pipeline is as follows:

```
Tokenizer tokenizer = new Tokenizer()
      .setInputCol("wordText")
      .setOutputCol("words");
HashingTF hashingTF = new HashingTF()
      .setNumFeatures(100)
      .setInputCol(tokenizer.getOutputCol())
      .setOutputCol("features");
LogisticRegression logisticRegression = new
LogisticRegression()
      .setMaxIter(10)
      .setRegParam(0.01);
Pipeline pipeline = new Pipeline().setStages(new
PipelineStage[] {tokenizer, hashingTF,
logisticRegression});
    // Fit the pipeline to training documents.
PipelineModel model = pipeline.fit(trainingData);
Dataset<Row> predictions =
model.transform(testData);
for (Row r: predictions.select("label",
"wordText", "prediction").collectAsList()) {
  System.out.println("(" + r.get(0) + ", " +
r.get(1) + ") --> prediction=" + r.get(2));
    } }
```

The output is as follows:

```
(0.0, What you doing how are you)
--> prediction=0.0
```

```
(0.0, MY NO IN LUTON 0125698789 RING ME IF UR
AROUND H)
--> prediction=0.0
(1.0, Sunshine Quiz Win a super Sony DVD recorder
if you canname the capital of Australia Text
MQUIZ to 82277 B)
--> prediction=0.0
```

Feature transformation, extraction, and selection

The preceding section showed you the overall process of the pipeline. This pipeline or workflow is basically the collection of some operation such as transformation to one dataset of another data set, extracting the features, and selecting the features. These are the basic operators for feature engineering that we already described in previous sections. This section will show you the details about those operations using Spark machine learning packages. Spark provides some efficient APIs for feature engineering including MLlib and ML.

In this section, we will start with the ML package by continuing the Spam Filter examples. Let's read a large dataset from the text file as Dataset, which contains the lines starting with ham or spam words. The sample output of this Dataset is given here. Now we will use this dataset for feature extracting the features and building a model with Spark's APIs.

The code for `Input DF` is as follows:

```
Dataset<Row> df =
spark.read().text("input/SMSSpamCollection.txt");
df.show();
```

The output is as follows:



```
+--------------------+
|               value|
+--------------------+
|ham    Go until juro...|
|ham    Ok lar... Jok...|
|spam   Free entry i...|
|ham    U dun say so ...|
|ham    Nah I don't t...|
|spam   FreeMsg Hey ...|
|ham    Even my broth...|
|ham    As per your r...|
|spam   WINNER!! As ...|
|spam   Had your mob...|
|ham    I'm gonna be ...|
|spam   SIX chances ...|
|spam   URGENT! You ...|
|ham    I've been sea...|
|ham    I HAVE A DATE...|
|spam   XXXMobileMov...|
|ham    Oh k...i'm wa...|
|ham    Eh u remember...|
|ham    Fine if that...|
|spam   England v Ma...|
+--------------------+
only showing top 20 rows
```

Transformation – RegexTokenizer

From the preceding output, you can see that we have to transform it into two columns for identifying the spam and ham messages. For doing this, we can use the `RegexTokenizer` Transformer that can take input from a regular expression (`regex`) and transform it to a new dataset. This code produces `labelFeatured`. For example, refer the Dataset shown in the following output:

```
// Feature Transformers (RegexTokenizer)
RegexTokenizer regexTokenizer1 = new
RegexTokenizer()
        .setInputCol("value")
        .setOutputCol("labelText")
        .setPattern("\\t.*$");
Dataset<Row> labelTextDataFrame =
regexTokenizer1.transform(df);
RegexTokenizer regexTokenizer2 = new
RegexTokenizer()

 .setInputCol("value").setOutputCol("text").setPattern("\
\W");
Dataset<Row> labelFeatureDataFrame =
regexTokenizer2
        .transform(labelTextDataFrame);
for (Row r : labelFeatureDataFrame.select("text",
"labelText").collectAsList()) {
      System.out.println( r.getAs(1) + ": " +
r.getAs(0));
```

```
        }
```

Here is the output of `labelFeature` Dataset:

```
WrappedArray(ham): WrappedArray(ham, what, you,
doing, how, are, you)
WrappedArray(ham): WrappedArray(ham, ok, lar,
joking, wif, u, oni)
WrappedArray(ham): WrappedArray(ham, dun, say,
so, early, hor, u, c, already, then, say)
WrappedArray(ham): WrappedArray(ham, my, no, in,
luton, 0125698789, ring, me, if, ur, around, h)
WrappedArray(spam): WrappedArray(spam, freemsg,
txt, call, to, no, 86888, claim, your, reward,
of, 3, hours, talk, time, to, use, from, your,
phone, now, ubscribe6gbp, mnth, inc, 3hrs, 16,
stop, txtstop)
WrappedArray(ham): WrappedArray(ham, siva, is,
in, hostel, aha)
WrappedArray(ham): WrappedArray(ham, cos, i, was,
out, shopping, wif, darren, jus, now, n, i,
called, him, 2, ask, wat, present, he, wan, lor,
then, he, started, guessing, who, i, was, wif, n,
he, finally, guessed, darren, lor)
WrappedArray(spam): WrappedArray(spam, sunshine,
quiz, win, a, super, sony, dvd, recorder, if,
you, canname, the, capital, of, australia, text,
mquiz, to, 82277, b)
```

Now let's create a new Dataset from the `labelFeatured` Dataset that we just created by selecting the label text as follows:

```
Dataset<Row> newDF =
labelFeatureDataFrame.withColumn("labelTextTemp",
       labelFeatureDataFrame.col("labelText").cast(DataTypes.StringType))
       .drop(labelFeatureDataFrame.col("labelText"))
    .withColumnRenamed("labelTextTemp",
"labelText");
```

Now let's further explore the contents in the new Dataset by calling the `show()` method as follows:

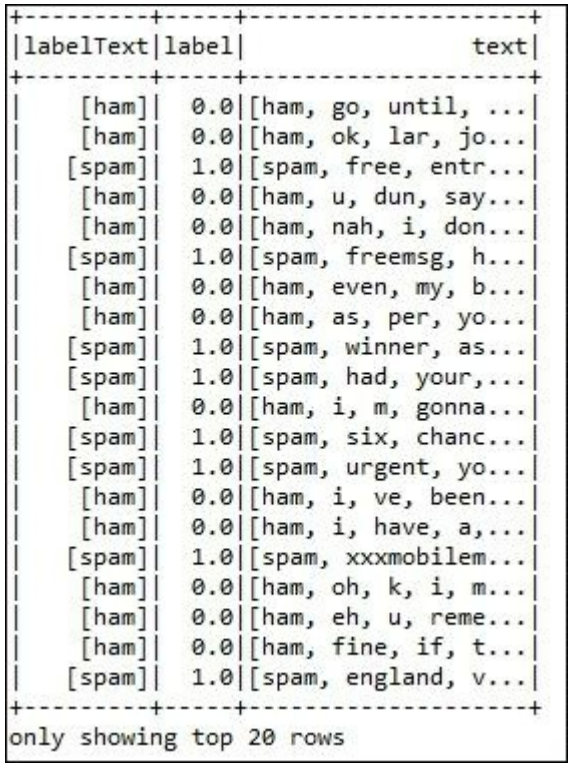
```
+--------------------+--------------------+---------+
|               value|                text|labelText|
+--------------------+--------------------+---------+
|ham    Go until juro...|[ham, go, until, ...|    [ham]|
|ham    Ok lar... Jok...|[ham, ok, lar, jo...|    [ham]|
|spam   Free entry i...|[spam, free, entr...|   [spam]|
|ham    U dun say so ...|[ham, u, dun, say...|    [ham]|
|ham    Nah I don't t...|[ham, nah, i, don...|    [ham]|
|spam   FreeMsg Hey ...|[spam, freemsg, h...|   [spam]|
|ham    Even my broth...|[ham, even, my, b...|    [ham]|
|ham    As per your r...|[ham, as, per, yo...|    [ham]|
|spam   WINNER!! As ...|[spam, winner, as...|   [spam]|
|spam   Had your mob...|[spam, had, your,...|   [spam]|
|ham    I'm gonna be ...|[ham, i, m, gonna...|    [ham]|
|spam   SIX chances ...|[spam, six, chanc...|   [spam]|
|spam   URGENT! You ...|[spam, urgent, yo...|   [spam]|
|ham    I've been sea...|[ham, i, ve, been...|    [ham]|
|ham    I HAVE A DATE...|[ham, i, have, a,...|    [ham]|
|spam   XXXMobileMov...|[spam, xxxmobilem...|   [spam]|
|ham    Oh k...i'm wa...|[ham, oh, k, i, m...|    [ham]|
|ham    Eh u remember...|[ham, eh, u, reme...|    [ham]|
|ham    Fine if that...|[ham, fine, if, t...|    [ham]|
|spam   England v Ma...|[spam, england, v...|   [spam]|
+--------------------+--------------------+---------+
only showing top 20 rows
```

Transformation – StringIndexer

The preceding output has the classification of ham and spam messages, but we have to make the ham and spam text as double values. The `StringIndexer` Transformer can do it easily. It can encode a string column of labels into indices in another column. The indices are ordered by label frequencies. `StringIndexer` produces two indices, 0.0 and 1.0 for our dataset:

```
// Feature Transformer (StringIndexer)
StringIndexer indexer = new
StringIndexer().setInputCol("labelText")
        .setOutputCol("label");
Dataset<Row> indexed =
indexer.fit(newDF).transform(newDF);
    indexed.select(indexed.col("labelText"),
indexed.col("label"),
indexed.col("text")).show();
```

The following is the output for the `indexed.show()` function:



```
+---------+-----+--------------------+
|labelText|label|                text|
+---------+-----+--------------------+
|    [ham]|  0.0|[ham, go, until, ...|
|    [ham]|  0.0|[ham, ok, lar, jo...|
|   [spam]|  1.0|[spam, free, entr...|
|    [ham]|  0.0|[ham, u, dun, say...|
|    [ham]|  0.0|[ham, nah, i, don...|
|   [spam]|  1.0|[spam, freemsg, h...|
|    [ham]|  0.0|[ham, even, my, b...|
|    [ham]|  0.0|[ham, as, per, yo...|
|   [spam]|  1.0|[spam, winner, as...|
|   [spam]|  1.0|[spam, had, your,...|
|    [ham]|  0.0|[ham, i, m, gonna...|
|   [spam]|  1.0|[spam, six, chanc...|
|   [spam]|  1.0|[spam, urgent, yo...|
|    [ham]|  0.0|[ham, i, ve, been...|
|    [ham]|  0.0|[ham, i, have, a,...|
|   [spam]|  1.0|[spam, xxxmobilem...|
|    [ham]|  0.0|[ham, oh, k, i, m...|
|    [ham]|  0.0|[ham, eh, u, reme...|
|    [ham]|  0.0|[ham, fine, if, t...|
|   [spam]|  1.0|[spam, england, v...|
+---------+-----+--------------------+
only showing top 20 rows
```

Transformation – StopWordsRemover

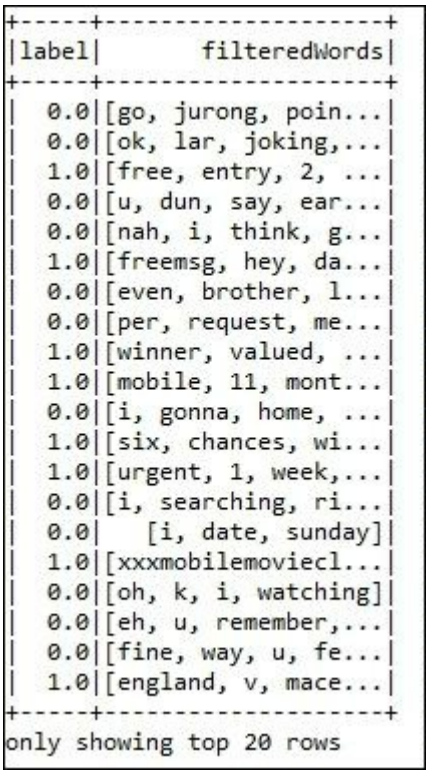
The preceding output contains words or tokens, but some words are not as important as features. Therefore, we need to remove those words. For making this task easier, Spark provides the list of stop words through the `StopWordsRemover` class that will be discussed more in Chapter 6, *Building Scalable Machine Learning Pipelines*.

We can use those words to filter unwanted words. Additionally, we will remove the ham and spam words from the text column. The `StopWordsRemover` class will transform the preceding Dataset into a filtered Dataset by removing the stop works from the features. The following output will show us the words without spam and ham word tokens:

```
// Feature Transformers (StopWordsRemover)
StopWordsRemover remover = new
StopWordsRemover();
String[] stopwords = remover.getStopWords();
String[] newStopworks = new
String[stopwords.length+2];
newStopworks[0]="spam";
newStopworks[1]="ham";
for(int i=2;i<stopwords.length;i++){
      newStopworks[i]=stopwords[i];}
remover.setStopWords(newStopworks).setInputCol("text").setOutputCol("filt
Dataset<Row> filteredDF =
remover.transform(indexed);
```

```
filteredDF.select(filteredDF.col("label"),
filteredDF.col("filteredWords")).show();
```

The output is as follows:



```
+-----+--------------------+
|label|       filteredWords|
+-----+--------------------+
|  0.0|[go, jurong, poin...|
|  0.0|[ok, lar, joking,...|
|  1.0|[free, entry, 2, ...|
|  0.0|[u, dun, say, ear...|
|  0.0|[nah, i, think, g...|
|  1.0|[freemsg, hey, da...|
|  0.0|[even, brother, l...|
|  0.0|[per, request, me...|
|  1.0|[winner, valued, ...|
|  1.0|[mobile, 11, mont...|
|  0.0|[i, gonna, home, ...|
|  1.0|[six, chances, wi...|
|  1.0|[urgent, 1, week,...|
|  0.0|[i, searching, ri...|
|  0.0|   [i, date, sunday]|
|  1.0|[xxxmobilemoviecl...|
|  0.0|[oh, k, i, watching]|
|  0.0|[eh, u, remember,...|
|  0.0|[fine, way, u, fe...|
|  1.0|[england, v, mace...|
+-----+--------------------+
only showing top 20 rows
```

Extraction – TF

Now we have the Dataset containing a label with a double value and filtered words or tokens. The next task is to vectorize (make numeric values) the features or extract the features from the words or tokens.

TF-IDF (`HashingTF` and `IDF`; also known as **Term Frequency-Inverse Document Frequency**) is a feature vectorization method widely used for extracting the features, which basically calculates the importance of a term to a document in the corpus.

`TF` counts the frequency of the terms in a document or line and `IDF` counts the document or line frequency, that is, number of document or lines containing a particular term. The following code explains the term frequency of the preceding dataset using the efficient `HashingTF` class of Spark. `HashingTF` is a Transformer that takes sets of terms; and converts those sets into fixed-length feature vectors. The output of the featured data is also shown:

```
// Feature Extractors (HashingTF transformer)
int numFeatures = 100;
```

```
HashingTF hashingTF = new
HashingTF().setInputCol("filteredWords")

 .setOutputCol("rawFeatures").setNumFeatures(numFeatures);
Dataset<Row> featurizedData =
hashingTF.transform(filteredDF);
    for (Row r :
featurizedData.select("rawFeatures",
"label").collectAsList()) {
Vector features = r.getAs(0); ////Problematic
line
Double label = r.getDouble(1);
System.out.println(label + "," + features);
    }
```

The output is as follows:

```
0.0,(100,[19],[1.0])
0.0,(100,[9,16,17,48,86,96],
[1.0,1.0,1.0,1.0,1.0,1.0])
0.0,(100,[17,37,43,71,99],[1.0,1.0,2.0,1.0,2.0])
0.0,(100,[4,41,42,47,92],[1.0,1.0,1.0,1.0,1.0])
1.0,(100,
[3,12,19,26,28,29,34,41,46,51,71,73,88,93,94,98],
[1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,2.0])
0.0,(100,[19,25,38],[1.0,1.0,1.0])
0.0,(100,
[8,10,16,30,37,43,48,49,50,55,76,82,89,95,99],
[1.0,4.0,2.0,1.0,1.0,2.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,2.0])
1.0,(100,[0,24,36,39,42,48,53,58,67,86,95,97,98],
[1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,2.0,1.0,1.0,2.0,1.0])
```

Extraction – IDF

Similarly, we can apply `IDF` on the featured data to count the document frequency. `IDF` is an Estimator that fits on the preceding dataset and produces an `IDFModel` that transforms to a rescaled dataset containing features and labels:

```
// Feature Extractors (IDF Estimator)
IDF idf = new
IDF().setInputCol("rawFeatures").setOutputCol("features");
IDFModel idfModel = idf.fit(featurizedData);
Dataset<Row> rescaledData =
idfModel.transform(featurizedData);
for (Row r : rescaledData.select("features",
"label").collectAsList()) {
Vector features = r.getAs(0);
Double label = r.getDouble(1);
System.out.println(label + "," + features);
    }
```

The output is as follows:

```
0.0,(100,[19],[0.8109302162163288])
0.0,(100,[9,16,17,48,86,96],
[1.5040773967762742,1.0986122886681098,1.0986122886681098,0.8109302162163
0.0,(100,[17,37,43,71,99],
[1.0986122886681098,1.0986122886681098,2.1972245773362196,1.0986122886681
0.0,(100,[4,41,42,47,92],
[1.5040773967762742,1.0986122886681098,1.0986122886681098,1.5040773967762
1.0,(100,
[3,12,19,26,28,29,34,41,46,51,71,73,88,93,94,98],
[1.5040773967762742,1.5040773967762742,0.8109302162163288,1.5040773967762
0.0,(100,[19,25,38],
[0.8109302162163288,1.5040773967762742,1.5040773967762742])
0.0,(100,
[8,10,16,30,37,43,48,49,50,55,76,82,89,95,99],
[1.5040773967762742,6.016309587105097,2.1972245773362196,1.50407739677627
1.0,(100,[0,24,36,39,42,48,53,58,67,86,95,97,98],
[1.5040773967762742,1.5040773967762742,1.5040773967762742,1.5040773967762
```

The preceding output extracts features from the raw texts. The very first entry is the label and the rest are the feature vector extracted.

Selection – ChiSqSelector

The preceding output is ready for training using a classification algorithm such as `LogisticRegression`. But we can use the more important feature from the categorical features. For doing this, Spark provides some feature selector APIs such as `ChiSqSelector`. The `ChiSqSelector` is called **Chi-Squared feature selection**.

It operates on labeled data with categorical features. It orders features based on a Chi-Squared test, which is independent from the class, and then filters the top features which the class label depends on the most. This selector is useful for improving the predictive power of a model. The following code will select the top three features from the feature vectors, along with the output given:

```
org.apache.spark.ml.feature.ChiSqSelector
selector = new
org.apache.spark.ml.feature.ChiSqSelector();
selector.setNumTopFeatures(3).setFeaturesCol("features")

 .setLabelCol("label").setOutputCol("selectedFeatures");
Dataset<Row> result =
selector.fit(rescaledData).transform(rescaledData);
    for (Row r :
result.select("selectedFeatures",
"label").collectAsList()) {
  Vector features = r.getAs(0);
```

```
Double label = r.getDouble(1);
System.out.println(label + "," + features);
  }
```

> **Tip**
>
> We will discuss more on the `ChiSqSelector`, `IDFModel`, `IDF`, `StopWordsRemover`, and `RegexTokenizer` classes in Chapter 6, *Building Scalable Machine Learning Pipelines*.

The output is as follows:

```
0.0,(3,[],[])
0.0,(3,[],[])
0.0,(3,[],[])
0.0,(3,[],[])
1.0,(3,[1,2],
[1.5040773967762742,2.1972245773362196])
0.0,(3,[],[])
0.0,(3,[],[])
1.0,(3,[0,2],
[1.5040773967762742,1.0986122886681098])
```

Now, you can apply `LogisticRegression` when building a model with the feature vectors. Spark provides lots of different APIs for feature engineering. However, we have not used the other machine learning of Spark (that is, Spark MLlib) for the brevity and page limitation. We will discuss the feature engineering using `spark.mllib` gradually with examples in future chapters.

Advanced feature engineering

In this section, we will discuss some advanced features that are also involved in the feature engineering process such as manual feature construction, feature learning, iterative process of feature engineering, and deep learning.

Feature construction

The best results come down to you through the manual feature engineering or feature construction. Therefore, manual construction is the process of creating new features from the raw data. Feature selection based on the feature's importance can inform you about the objective utility of features; however, those features have to come from somewhere else. In fact, sometimes, you need to manually create them.

In contrast to the feature selection, the feature construction technique requires spending a lot of effort and time with not the aggregation or picking the feature, but on the actual raw data so that new features can be constructive towards increasing the predictive accuracies of the model. Therefore, it also involves thinking of the underlying structure of the data along with the ML problem.

In this regard, to construct new features from the complex and high dimensional dataset, you need to know the overall structure of the data. In addition to this, how to use and apply them in predictive modeling algorithms. There will be three aspects in terms of tabular, textual, and multimedia datasets:

- Handling and manual creation from the tabular data often means a mixture of combining features to create new features. You might also need the decomposing or splitting of some original features to create new features.
- With textual data, it often means devising document or context-specific indicators relevant to the problem. For example, when you are applying text analytics on large raw data such as data from Twitter hashtags.
- With multimedia data such as image data, it can often mean enormous amounts of time are passed to pick out relevant structures in a manual way.

Unfortunately, the feature construction technique is not only manual, but the whole process is slower, requiring lots of research involvement from humans like you and us. However, it can make a big difference in the long run. In fact, feature engineering and feature selection are not mutually

exclusive; however, both of them are important in the realm of machine learning.

Feature learning

Is it possible to avoid the manual process of prescribing how to construct or extract features from raw data? Feature learning helps you to get rid of this. Therefore, feature learning is an advanced process; alternatively, an automatic identification and use of features from raw data. This is also referred to as representation learning that helps your machine learning algorithm to identify useful features.

The feature learning technique is commonly used in deep learning algorithms. As a result, recent deep learning techniques are achieving some success in this area. The auto-encoders and restricted Boltzmann machines are such an example where the concept of feature learning was used. The key idea behind feature learning is the automatic and abstract representations of the features in a compressed form using unsupervised or semi-supervised learning algorithms.

Speech recognition, image classification, and object recognition are some successful examples; where researchers have found supported state-of-the-art results. Further details could not have been represented in this book due to the brevity.

Unfortunately, Spark has not implemented any APIs for the automatic feature extraction or construction.

Iterative process of feature engineering

The whole process of feature engineering is not a standalone, but more or less iterative. Since you are actually interplaying with the form the data selection to model evaluation again and again until you are completely satisfied or you are running out of time. The iteration could be imagined as a four-step workflow that iteratively runs over time. When you are aggregating or collecting the raw data, you might not be doing enough brainstorming. However, when you start exploring the data, you are really getting into the problem into deeper.

After that you will be looking at a lot of data, studying the best technique of feature engineering and the related problems presented in the state of the arts and you will see how much you are able to steal. When you have done enough brainstorming, you will start devising the required features or extracting the features depending on your problem type or class. You might use the automatic feature extraction or manual feature construction (or both sometimes). If you are not satisfied with the performance you might redo

the feature extraction process for improvement. Please refer to *Figure 7* for a clear view of the iterative process of feature engineering:

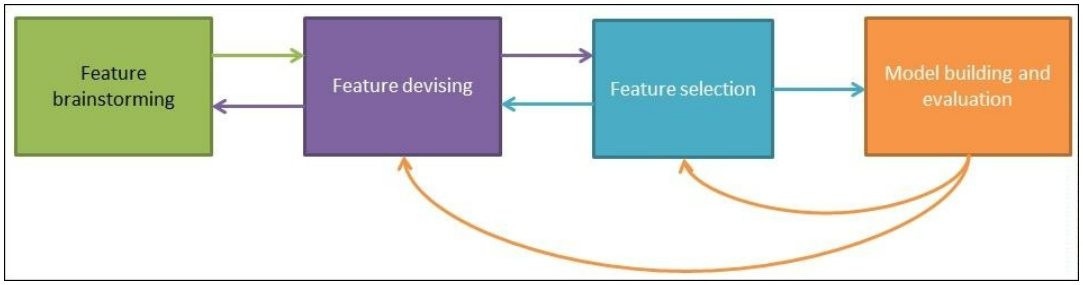


Figure 8: The iterative processing in feature engineering.

When you have devised or extracted the feature, you need to select the features. You might apply a different scoring or ranking mechanism based on feature importance. Similarly, you might iterate the same process such as devising the feature to improve the model. And finally, you will evaluate your model to estimate the model's accuracy on new data to make your model adaptive.

You also need a well-defined problem that will help you to stop the whole iteration. When finished, you can move on to try other models. There will be gain waiting for you in the future once you plateau on ideas or the accuracy delta out of your ML pipeline.

Deep learning

One of the most interesting and promising moves in data representation we would say is deep learning. It is very popular on the tensor computing application and the **Artificial Intelligent Neural Network** (**AINN**) system. Using the deep learning technique, the network learns how to represent data at different levels.

Therefore, you will have an exponential ability to represent the linear data you have. Spark can take this advantage and it can be used to improve deep learning. For more general discussion, please refer to the following URL at https://en.wikipedia.org/wiki/Deep_learning and to learn how to deploy pipelines on a cluster with TensorFlow, see https://www.tensorflow.org/.

A recent research and development at Databricks (also see https://databricks.com/) has shown that Spark can also be used to find the best set of hyperparameters for AINN training. The advantage is that Spark will do the computation 10X faster than a normal deep learning or neural network algorithm.

Consequently, your model training time will drastically reduce up to 10 times and the error rate will be 34% lower. Moreover, Spark can be applied

to a trained AINN model on a large amount of data so you can deploy your ML model at scale. We will discuss more on deep learning in later chapters as advanced machine learning.

Summary

Feature engineering, feature selection, and feature construction are the three most commonly used steps while preparing the training and test set towards building a machine learning model. Usually, the feature engineering is applied first to generate additional features from the available dataset. After that, the feature selection technique is applied to eliminate irrelevant, missing or null, redundant, or even highly correlated features so that high predictive accuracy can be availed.

In contrast, feature construction is an advanced technique applied to construct new features that are either absent or trivial in the raw dataset.

Note that it is not always necessary to perform feature engineering or feature selection. Whether to perform feature selection and construction depends on the data you have or collected, what kind of ML algorithm you have picked, and the objective of the experiment itself.

In this chapter, we have described all of the three steps in detail with practical Spark examples. In the next chapter, we will describe in detail some practical examples of supervised and unsupervised learning using two machine learning APIs: Spark MLlib and Spark ML.

Chapter 5. Supervised and Unsupervised Learning by Examples

In Chapter 2, *Machine Learning Best Practices* readers, learned some theoretical underpinnings of basic machine learning techniques. Whereas, Chapter 3, *Understanding the Problem by Understanding the Data,* describes the basic data manipulation using Spark's APIs such as RDD, DataFrame, and Datasets. Chapter 4, *Extracting Knowledge through Feature Engineering*, on the other hand, describes feature engineering from both the theoretical and practical point of view. However, in this chapter, the reader will learn the practical know-how needed quickly and powerfully to apply supervised and unsupervised techniques on the available data to the new problems through some widely used examples based on the understandings from the previous chapters. These examples we are talking about will be demonstrated from the Spark perspective. In a nutshell, the following topics will be covered throughout this chapter:

- Machine learning classes
- Supervised learning
- Unsupervised learning
- Recommender system
- Advanced learning and generalization

Machine learning classes

As stated in Chapter 1, *Introduction to Data Analytics with Spark* and Chapter 2, *Machine Learning Best Practices*, machine learning techniques can be categorized further into three major classes of algorithms: supervised learning, unsupervised learning, and the recommender system. Where classification and regression algorithms are widely used in the supervised learning application development, clustering, on the other hand, falls in the category of unsupervised learning. In this section, we will describe some examples of the supervised learning technique.

Then we will provide some example of the same example presented using Spark. On the other hand, an example of the clustering technique will be discussed in the section: *Unsupervised learning*, where a regression technique often models the past relationship between variables to predict their future changes (up or down). Here we show two real-life examples of classification and regression algorithms respectively. In contrast, a classification technique takes a set of data with known labels and learns how to label new records based on that information:

- **Example (classification)**: Gmail uses a machine learning technique called classification to designate if an e-mail is spam or not, based on the data of an e-mail.
- **Example (regression)**: As an example, suppose you are an online currency trader and you work on Forex or Fortrade. Right now you have two currency pairs in mind to buy or sell say: GBP/USD and USD/JPY. If you look at these two pairs carefully, USD is a common in these two pairs. Now if you look at the historical prices of USD, GBP, or JPY you can predict the future outcome of whether you should open the trade in buy or sell. These types of problems can be resolved with supervised learning techniques using regression analysis:

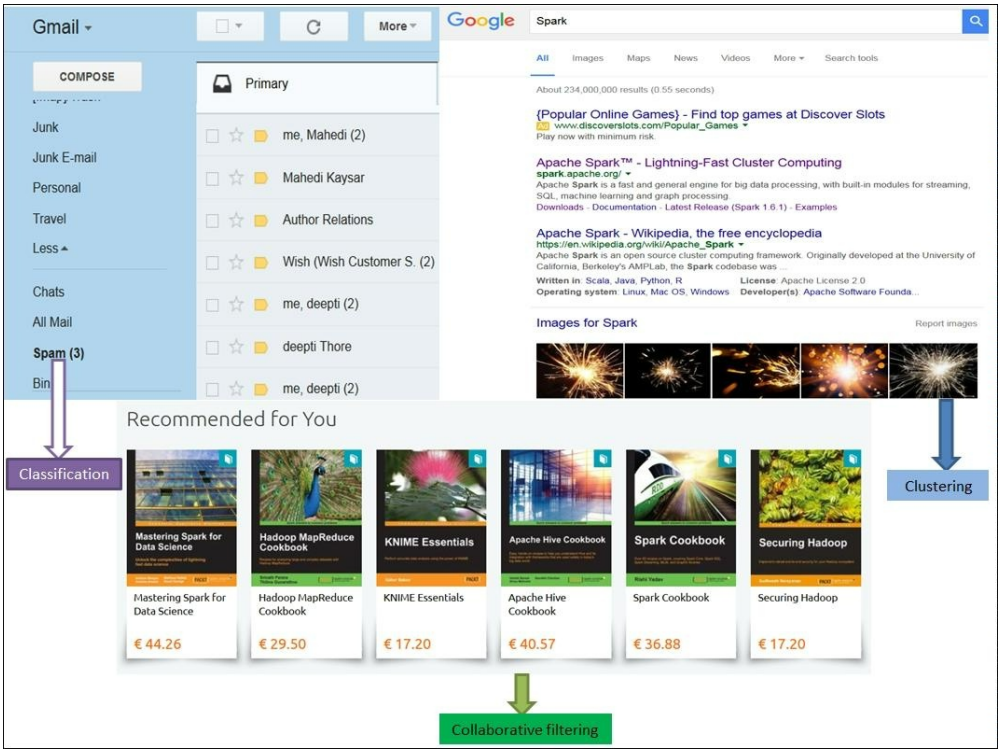


Figure 1: Classification, clustering, and collaborative filtering-the big picture

On the other hand, clustering and dimensionality reduction are commonly used for unsupervised learning. Here are some examples:

- **Example (clustering)**: Google News uses a technique called clustering to group news articles into different categories, based on title and content. Clustering algorithms discover groupings that occur in collections of data.
- **Example (collaborative filtering)**: The collaborative filtering algorithm is often used in the recommendation system development. Renowned companies such as Amazon and Netflix use a machine learning technique called collaborative filtering, to determine which

products users will like based on their history and similarity to other users.
- **Example (dimensionality reduction)**: Dimensionality reduction is often used to make the available dataset that is high dimensional. For example, suppose you have an image of size 2048x1920, and you would like to reduce the dimension to 1080x720 without sacrificing the quality much. In this case, popular algorithms such as **Principal Component Analysis** (**PCA**) or **Singular Value Decomposition** (**SVD**) can be used although you can also implement the SVD to implement the PCA. This is why SVD is more widely used.

Supervised learning

As already stated, a supervised learning application makes predictions based on a set of examples and the goal is to learn general rules that map inputs to outputs aligning with the real world. For example, a dataset for spam filtering usually contains spam messages as well as non-spam messages. Consequently, we could know which messages in the training set are spams or non-spam. Therefore, supervised learning is the machine learning technique of inferring a function from the labeled training data. The following steps are involved in supervised learning tasks:

- Train the ML model with the training dataset
- Use the test dataset to test the model performance

Therefore, the dataset for training the ML model, in this case, is labeled with the value of interest and a supervised learning algorithm looks for patterns in those value labels. After the algorithm has found the required patterns, those patterns can be used to make predictions for unlabeled test data.

A typical use of the supervised learning is diverse and commonly used in the bioinformatics, cheminformatics, database marketing, handwriting recognition, information retrieval, object recognition in computer vision, optical character recognition, spam detection, pattern recognition, speech recognition, and so on, and in these applications mostly the classification technique is used. On the other hand, supervised learning is a special case of downward causation in biological systems.

> **Tip**
>
> More on how the supervised learning technique works from the theoretical perspective can be found on these books: Vapnik, V. N. *The Nature of Statistical Learning Theory (2nd Ed.)*, Springer Verlag, 2000; and Mehryar M., Afshin R. Ameet T. (2012) Foundations of Machine Learning, The MIT Press ISBN 9780262018258.

Supervised learning example

Classification is a family of supervised machine learning algorithms that designate input as belonging to one of the several pre-defined classes. Some common use cases for classification include:

- Credit card fraud detection
- E-mail spam detection

Classification data is labeled, for example, as spam/non-spam or fraud/non-fraud. Machine learning assigns a label or class to new data. You classify something based on pre-determined features. Features are the *if questions* that you ask. The label is the answer to those questions. For example, if an object walks, swims, and quacks like a duck, then the label would be *duck*. Or suppose for a flight is delayed on to be a departure or arrival by more than say 1 hour, it would be a delay; otherwise not a delay.

Supervised learning with Spark - an example

We will demonstrate an example by analyzing an air-flight delay. The dataset named `On_Time_Performance_2016_1.csv` from the United Department of Transportation website at http://www.transtats.bts.gov/ will be used.

Air-flight delay analysis using Spark

We are using flight information for 2016. For each flight, we have the following information presented in *Table 1* (we have presented only a few fields out of 444,827 rows and 110 columns as of May 17, 2016):

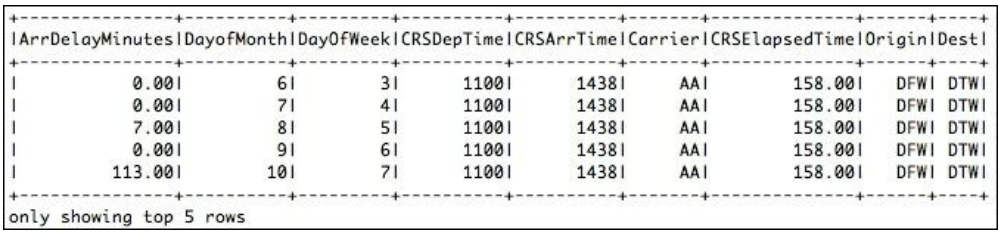
Data field	Description	Example value
`DayofMonth`	Day of month	2
`DayOfWeek`	Day of week	5
`TailNum`	Tail number for the plane	N505NK
`FlightNum`	Flight number	48
`AirlineID`	Airline ID	19805
`OriginAirportID`	Origin airport ID	JFK
`DestAirportID`	Destination airport ID	LAX
`Dest`	Destination airport code	1424

CRSDepTime	Schedule departure time	10:00
DepTime	Actual departure time	10:30
DepDelayMinutes	Departure delay in minutes	30
CRSArrTime	Schedule arrival time	22:45
ArrTime	Actual arrival time	23:45
ArrDelayMinutes	Arrival delay in minutes	60
CRSElapsedTime	Elapsed time	825
Distance	Total distance	6200

Table 1: Sample data from the "On Time On Time Performance 2016_1" dataset

In this scenario, we will build a tree to predict the label of delayed or not delayed based on the following features shown in the figure, which is small snapshot of an air flight dataset. Here ArrDelayMinutes is 113 which should be classified in delayed (1.0) and other rows are less than 60 minutes so the label should be 0.0 (not delayed). From this dataset, we will do some operation such as feature extraction, transformation, and selection. *Table 2* shows the top five rows related to the features we will be considering for this example as follows:

- **Label**: Delayed and not delayed - delayed if delay >60 minutes
- **Features**: {DayOfMonth, WeekOfday, CRSdeptime, CRSarrtime, Carrier, CRSelapsedtime, Origin, Dest, ArrDelayMinutes}



```
+---------------+----------+---------+----------+----------+-------+--------------+------+----+
|ArrDelayMinutes|DayofMonth|DayOfWeek|CRSDepTime|CRSArrTime|Carrier|CRSElapsedTime|Origin|Dest|
+---------------+----------+---------+----------+----------+-------+--------------+------+----+
|           0.00|         6|        3|      1100|      1438|     AA|        158.00|   DFW| DTW|
|           0.00|         7|        4|      1100|      1438|     AA|        158.00|   DFW| DTW|
|           7.00|         8|        5|      1100|      1438|     AA|        158.00|   DFW| DTW|
|           0.00|         9|        6|      1100|      1438|     AA|        158.00|   DFW| DTW|
|         113.00|        10|        7|      1100|      1438|     AA|        158.00|   DFW| DTW|
+---------------+----------+---------+----------+----------+-------+--------------+------+----+
only showing top 5 rows
```

Figure 2: Selected feature for air-flight delay prediction

Loading and parsing the Dataset

Before performing the feature extraction, we need to load and parse the dataset. This step also includes: loading packages and related dependencies, reading the dataset as a DataFrame, making the POJO or Bean class, and adding the new label column based on requirements.

Step 1: Load required packages and dependencies

For reading csv files, we used the csv reader provided by the Databricks:

```
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.spark.api.java.JavaRDD;
import
org.apache.spark.api.java.function.Function;
import org.apache.spark.ml.Pipeline;
import org.apache.spark.ml.PipelineModel;
import org.apache.spark.ml.PipelineStage;
import
org.apache.spark.ml.classification.DecisionTreeClassificationModel;
import
org.apache.spark.ml.classification.DecisionTreeClassifier;
import
org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator;
import org.apache.spark.ml.feature.IndexToString;
import org.apache.spark.ml.feature.LabeledPoint;
import org.apache.spark.ml.feature.StringIndexer;
import
org.apache.spark.ml.feature.StringIndexerModel;
import
org.apache.spark.ml.feature.VectorAssembler;
import org.apache.spark.ml.feature.VectorIndexer;
import
org.apache.spark.ml.feature.VectorIndexerModel;
import org.apache.spark.ml.linalg.Vector;
import org.apache.spark.rdd.RDD;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;
import scala.Tuple2;
```

Step 2: Create the Spark session

Here is the code to create a Spark session:

```
  static SparkSession spark = SparkSession
      .builder()
      .appName("JavaLDAExample")
      .master("local[*]")
      .config("spark.sql.warehouse.dir",
"E:/Exp/")
      .getOrCreate();
```

Step 3: Read and parse the csv file using Dataset

This dataset contains many columns that we will not include as a feature in this example. So we will select from the DataFrame only the features that we have mentioned previously. This DataFrame output has already been shown in *Figure 2*:

```
String csvFile =
"input/On_Time_On_Time_Performance_2016_1.csv";
Dataset<Row> df =
spark.read().format("com.databricks.spark.csv").option("header",
"true").load(csvFile);
RDD<Tuple2<String, String>> distFile =
spark.sparkContext().wholeTextFiles("input/test/*.txt",
2);
JavaRDD<Tuple2<String, String>> distFile2 =
distFile.toJavaRDD();
JavaRDD<Row> rowRDD = df.toJavaRDD();
Dataset<Row> newDF =
df.select(df.col("ArrDelayMinutes"),
df.col("DayofMonth"), df.col("DayOfWeek"),
df.col("CRSDepTime"), df.col("CRSArrTime"),
df.col("Carrier"),
df.col("CRSElapsedTime"), df.col("Origin"),
df.col("Dest"));
newDF.show(5);
```

Here is a the output of top 5 rows:

```
+---------------+----------+---------+----------+----------+-------+--------------+------+----+
|ArrDelayMinutes|DayofMonth|DayOfWeek|CRSDepTime|CRSArrTime|Carrier|CRSElapsedTime|Origin|Dest|
+---------------+----------+---------+----------+----------+-------+--------------+------+----+
|           0.00|         6|        3|      1100|      1438|     AA|        158.00|   DFW| DTW|
|           0.00|         7|        4|      1100|      1438|     AA|        158.00|   DFW| DTW|
|           7.00|         8|        5|      1100|      1438|     AA|        158.00|   DFW| DTW|
|           0.00|         9|        6|      1100|      1438|     AA|        158.00|   DFW| DTW|
|         113.00|        10|        7|      1100|      1438|     AA|        158.00|   DFW| DTW|
+---------------+----------+---------+----------+----------+-------+--------------+------+----+
only showing top 5 rows
```

Step 4: Making a POJO or Bean class

The POJO class we have developed is called `Flight` where the required features and label field will be defined with the corresponding setter and getter.

```
public class Flight implements Serializable {
  double label;
  double monthDay;
  double weekDay;
  double crsdeptime;
  double crsarrtime;
  String carrier;
  double crselapsedtime;
  String origin;
  String dest;

public Flight(double label, double monthDay,
double weekDay, double crsdeptime, double
crsarrtime, String carrier,
      double crselapsedtime, String origin,
String dest) {
    super();
    this.label = label;
    this.monthDay = monthDay;
    this.weekDay = weekDay;
    this.crsdeptime = crsdeptime;
    this.crsarrtime = crsarrtime;
    this.carrier = carrier;
    this.crselapsedtime = crselapsedtime;
    this.origin = origin;
    this.dest = dest;
  }
  public double getLabel() {
    return label;
  }
  public void setLabel(double label) {
    this.label = label;
  }
  public double getMonthDay() {
    return monthDay;
  }
  public void setMonthDay(double monthDay) {
    this.monthDay = monthDay;
  }
  public double getWeekDay() {
    return weekDay;
  }
  public void setWeekDay(double weekDay) {
    this.weekDay = weekDay;
  }
  public double getCrsdeptime() {
    return crsdeptime;
```

```
  }
  public void setCrsdeptime(double crsdeptime) {
    this.crsdeptime = crsdeptime;
  }
  public double getCrsarrtime() {
    return crsarrtime;
  }
  public void setCrsarrtime(double crsarrtime) {
    this.crsarrtime = crsarrtime;
  }
  public String getCarrier() {
    return carrier;
  }
  public void setCarrier(String carrier) {
    this.carrier = carrier;
  }
  public double getCrselapsedtime() {
    return crselapsedtime;
  }
  public void setCrselapsedtime(double
crselapsedtime) {
    this.crselapsedtime = crselapsedtime;
  }
  public String getOrigin() {
    return origin;
  }
  public void setOrigin(String origin) {
    this.origin = origin;
  }
  public String getDest() {
    return dest;
  }
  public void setDest(String dest) {
    this.dest = dest;
  }
  @Override
  public String toString() {
    return "Flight [label=" + label + ",
monthDay=" + monthDay + ", weekDay="
       + weekDay + ", crsdeptime="
        + crsdeptime + ", crsarrtime=" +
crsarrtime + ", carrier=" +
      carrier + ", crselapsedtime="
        + crselapsedtime + ", origin=" + origin +
", dest=" +
       dest + "]";
  }
```

We believe the preceding class is self-explanatory, which is used for setting and getting the feature values from the original dataset.

Step 5: Adding the new label column based on the delay column

If the delay is greater than 40 then the label should be 1 otherwise it should be 0. Create a new Dataset using the Flight bean class. This dataset can contain an empty string for the `ArrDelayMinutes` column. So before mapping we filtered the rows containing the empty string from the dataset:

```
JavaRDD<Flight> flightsRDD =
newDF.toJavaRDD().filter(new Function<Row,
Boolean>() {
          @Override
          public Boolean call(Row v1) throws
Exception {
            return !v1.getString(0).isEmpty();
          }
        }).map(new Function<Row, Flight>() {
          @Override
          public Flight call(Row r) throws
Exception {
            double label;
            double delay =
Double.parseDouble(r.getString(0));
            if (delay > 60)
              label = 1.0;
else
      label = 0.0;
double monthday =
Double.parseDouble(r.getString(1)) - 1;
double weekday =
Double.parseDouble(r.getString(2)) - 1;
double crsdeptime =
Double.parseDouble(r.getString(3));
double crsarrtime =
Double.parseDouble(r.getString(4));
String carrier = r.getString(5);
double crselapsedtime1 =
Double.parseDouble(r.getString(6));
String origin = r.getString(7);
String dest = r.getString(8);
Flight flight = new Flight(label, monthday,
weekday,crsdeptime, crsarrtime,
carrier,crselapsedtime1, origin, dest);
        return flight;
    }});
```

Now create a new Dataest from the RDD we created above as follows:

```
Dataset<Row> flightDelayData =
spark.sqlContext().createDataFrame(flightsRDD,Flight.class);
flightDelayData.printSchema();
```

Now show the top 5 rows from the data frame `flightDelayData` in following *Figure 3*:

```
flightDelayData.show(5);
```

[Output:]

```
+-------+----------+----------+--------------+----+-----+--------+------+-------+
|carrier|crsarrtime|crsdeptime|crselapsedtime|dest|label|monthDay|origin|weekDay|
+-------+----------+----------+--------------+----+-----+--------+------+-------+
|     AA|    1438.0|    1100.0|         158.0| DTW|  0.0|     5.0|   DFW|    2.0|
|     AA|    1438.0|    1100.0|         158.0| DTW|  0.0|     6.0|   DFW|    3.0|
|     AA|    1438.0|    1100.0|         158.0| DTW|  0.0|     7.0|   DFW|    4.0|
|     AA|    1438.0|    1100.0|         158.0| DTW|  0.0|     8.0|   DFW|    5.0|
|     AA|    1438.0|    1100.0|         158.0| DTW|  1.0|     9.0|   DFW|    6.0|
+-------+----------+----------+--------------+----+-----+--------+------+-------+
only showing top 5 rows
```

Figure 3:The DataFrame showing the new label column

Feature extraction

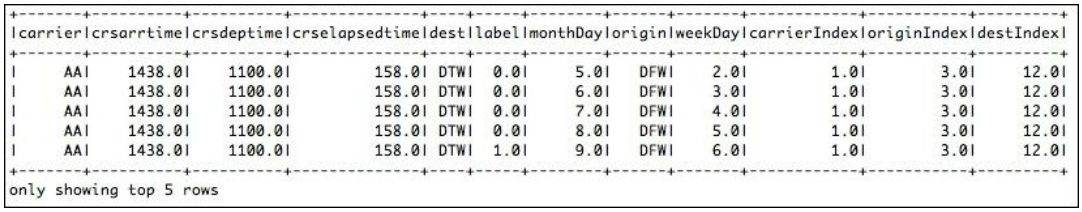
For extracting the feature, we have to make numerical values and if there are any text values, then we have to make a labeled vector for applying a machine learning algorithm.

Step 1 :Transformation towards feature extraction

Here we will transform the columns containing text into double values columns. Here we use `StringIndexer` for making a unique index for each unique text:

```
StringIndexer carrierIndexer = new
StringIndexer().setInputCol("carrier").setOutputCol("carrierIndex");
Dataset<Row> carrierIndexed =
carrierIndexer.fit(flightDelayData).transform(flightDelayData);
StringIndexer originIndexer = new
StringIndexer().setInputCol("origin").setOutputCol("originIndex");
Dataset<Row> originIndexed =
originIndexer.fit(carrierIndexed).transform(carrierIndexed);
StringIndexer destIndexer = new
StringIndexer().setInputCol("dest").setOutputCol("destIndex");
Dataset<Row> destIndexed =
destIndexer.fit(originIndexed).transform(originIndexed);
destIndexed.show(5);
```

[Output]:



carrier	crsarrtime	crsdeptime	crselapsedtime	dest	label	monthDay	origin	weekDay	carrierIndex	originIndex	destIndex
AA	1438.0	1100.0	158.0	DTW	0.0	5.0	DFW	2.0	1.0	3.0	12.0
AA	1438.0	1100.0	158.0	DTW	0.0	6.0	DFW	3.0	1.0	3.0	12.0
AA	1438.0	1100.0	158.0	DTW	0.0	7.0	DFW	4.0	1.0	3.0	12.0
AA	1438.0	1100.0	158.0	DTW	0.0	8.0	DFW	5.0	1.0	3.0	12.0
AA	1438.0	1100.0	158.0	DTW	1.0	9.0	DFW	6.0	1.0	3.0	12.0

only showing top 5 rows

Figure 4: Uunique indices for each unique text

Step 2: Making the feature vectors using the vector assembler

Make the feature vector with the vector assembler and transform it to the labelled vector for applying the machine learning algorithm (decision tree). Note, here we used the decision tree to show just an example since it shows better classification accuracies. Based on the algorithm and model selection and tuning, you will be further able to explore and use other classifiers:

```
VectorAssembler assembler = new
VectorAssembler().setInputCols(
        new String[] { "monthDay", "weekDay",
"crsdeptime",
            "crsarrtime", "carrierIndex",
"crselapsedtime",
            "originIndex",
"destIndex" }).setOutputCol(
        "assembeledVector");
```

Now transform the assembler into a Dataset of row as follows:

```
Dataset<Row> assembledFeatures =
assembler.transform(destIndexed);
```

Now convert the Dataset into `JavaRDD` for making the feature vectors as follows:

```
JavaRDD<Row> rescaledRDD =
assembledFeatures.select("label","assembeledVector").toJavaRDD();
```

Map the RDD for `LabeledPoint` as follows:

```
JavaRDD<LabeledPoint> mlData =
rescaledRDD.map(new Function<Row, LabeledPoint>()
{
          @Override
          public LabeledPoint call(Row row)
throws Exception {
            double label = row.getDouble(0);
            Vector v = row.getAs(1);
            return new LabeledPoint(label, v);
```

```
        }
    });
```

Now print the first five values as follows:

```
System.out.println(mlData.take(5));
```

[Output]:

```
+-----+--------------------+
|label|    assembeledVector|
+-----+--------------------+
|  0.0|[5.0,2.0,1100.0,1...|
|  0.0|[6.0,3.0,1100.0,1...|
|  0.0|[7.0,4.0,1100.0,1...|
|  0.0|[8.0,5.0,1100.0,1...|
|  1.0|[9.0,6.0,1100.0,1...|
+-----+--------------------+
only showing top 5 rows
```

Figure 5: The corresponding assembled vectors

Preparing the training and testing set

Here we will prepare the training dataset from the dataset of the labeled vector. Initially, we will make a training set where 15% of records will be non-delayed and 85% will be delayed records. Finally, the training and testing dataset will be prepared as 70% and 30% respectively.

Step 1: Make training and test set from the whole Dataset

First, create a new RDD by filtering the RDD based on the labels (that is, 1 and 0) we created previously as follows:

```
JavaRDD<LabeledPoint> splitedData0 =
mlData.filter(new Function<LabeledPoint,
Boolean>() {
          @Override
          public Boolean call(LabeledPoint r)
throws Exception {
              return r.label() == 0;
          }
        }).randomSplit(new double[] { 0.85,
0.15 })[1];

    JavaRDD<LabeledPoint> splitedData1 =
mlData.filter(new Function<LabeledPoint,
Boolean>() {
          @Override
          public Boolean call(LabeledPoint r)
throws Exception {
            return r.label() == 1;
```

```
            }
        });

    JavaRDD<LabeledPoint> splitedData2 =
splitedData1.union(splitedData0);
    System.out.println(splitedData2.take(1));
```

Now union the two RDDs using the `union()` method as follows:

```
JavaRDD<LabeledPoint> splitedData2 =
splitedData1.union(splitedData0);
System.out.println(splitedData2.take(1));
```

Now further convert the combined RDD into Dataset of Row as follows (max categories is set to be 4):

```
Dataset<Row> data =
spark.sqlContext().createDataFrame(splitedData2,
LabeledPoint.class);
data.show(100);
```

Now we need to do the vector indexer for the categorical variables as follows:

```
VectorIndexerModel featureIndexer = new
VectorIndexer()
          .setInputCol("features")
          .setOutputCol("indexedFeatures")
          .setMaxCategories(4)
          .fit(data);
```

Now that we have the feature indexer using the `VectorIndexerModel` estimator. Now the next task is to do the string indexing using the `StringIndexerModel` estimator as follows:

```
StringIndexerModel labelIndexer = new
StringIndexer()
          .setInputCol("label")
          .setOutputCol("indexedLabel")
          .fit(data);
```

Finally, split the Dataset of Row into training and test (70% and 30% respectively but you should adjust the values based on your requirements) set as follows:

```
Dataset<Row>[] splits = data.randomSplit(new
double[]{0.7, 0.3});
Dataset<Row> trainingData = splits[0];
```

```
Dataset<Row> testData = splits[1];
```

Well done! Now our dataset is ready to train the model, right? For the time being, we will naively select a classifier to say let's use the decision tree classifier to solve our purpose. You can try this with other multiclass classifiers based on examples provided in Chapter 6, *Building Scalable Machine Learning Pipelines*, Chapter 7, *Tuning Machine Learning Models*, and Chapter 8, *Adapting Your Machine Learning Models*.

Training the model

As shown in *Figure 2*, training and test data will be collected from the raw data. After the feature engineering process has been done, the RDD of feature vectors with labels or ratings will be used next to be processed by the classification algorithm before building the predictive model (as shown in *Figure 6*) and at the end the test data will be used for testing the model performance:

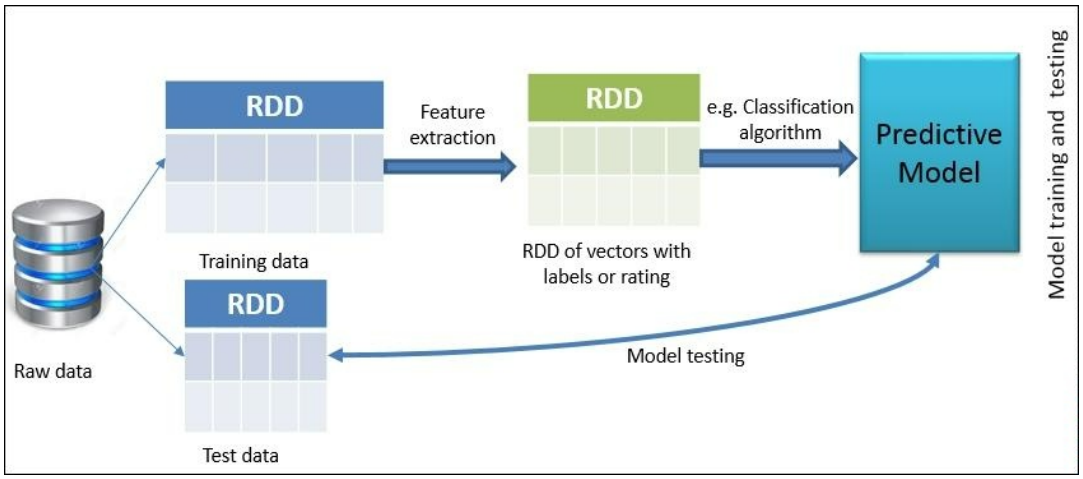


Figure 6: Supervised learning using Spark

Next, we prepare the values for the parameters that will be required for the decision tree. You might have wondered why we are talking about the decision tree. The reason is simple since using the decision tree (that is, the **binary decision tree**)we observed better prediction accuracy compared to the Naive Bayes approaches. Refer to *Table 2* which describes the categorical features and their significance as follows:

Categorical features	**Mapping**	**Significant**

categoricalFeaturesInfo	0 -> 31	Specifies that the feature index 0 (which represents the day of the month) has 31 categories [values {0, ..., 31}]
categoricalFeaturesInfo	1 -> 7	Represents days of the week, and specifies that the feature index 1 has seven categories
Carrier	0 -> N	N signifies the numbers from 0 to up to the number of distinct carriers

Table 2: Categorical features and their significance

Now we will describe the approach of the decision tree construction in brief. We will use the CategoricalFeaturesInfo that specifies which features are categorical and how many categorical values each of those features can take during the tree construction process. This is given as a map from the feature index to the number of categories for that feature.

However, the model is trained by making associations between the input features and the labeled output associated with those features. We train the model using the `DecisionTreeClassifier` method which returns a `DecisionTreeModel` eventually as shown in *Figure 7*. The detailed source code for constructing the tree will be shown later in this section:

```
    If (feature 7 in {16.0,20.0,26.0,29.0,30.0,31.0,33.0,34.0,39.0,41.0,47.0,49.0,54.0,58.0,76.0,78.0,88.0,90.0,92.0,100.0,103.0,106.0,108.0,1
     Predict: 0.0
    Else (feature 7 not in {16.0,20.0,26.0,29.0,30.0,31.0,33.0,34.0,39.0,41.0,47.0,49.0,54.0,58.0,76.0,78.0,88.0,90.0,92.0,100.0,103.0,106.0,1
     If (feature 7 in {32.0,35.0,36.0,38.0,45.0,50.0,52.0,53.0,55.0,56.0,61.0,68.0,69.0,70.0,71.0,72.0,73.0,74.0,77.0,79.0,84.0,85.0,86.0,89.0
      If (feature 6 in {8.0,16.0,24.0,29.0,34.0,35.0,36.0,55.0,56.0,60.0,63.0,65.0,88.0,104.0,115.0,121.0,9.0,13.0})
       If (feature 7 in {1.0,2.0,3.0,4.0,5.0,6.0,8.0,9.0,11.0,12.0,13.0,15.0,24.0,42.0,43.0,45.0,56.0,69.0,72.0})
        Predict: 0.0
       Else (feature 7 not in {1.0,2.0,3.0,4.0,5.0,6.0,8.0,9.0,11.0,12.0,13.0,15.0,24.0,42.0,43.0,45.0,56.0,69.0,72.0})
        Predict: 1.0
      Else (feature 6 not in {8.0,16.0,24.0,29.0,34.0,35.0,36.0,55.0,56.0,60.0,63.0,65.0,88.0,104.0,115.0,121.0,9.0,13.0})
       If (feature 2 <= 903.0)
        Predict: 0.0
       Else (feature 2 > 903.0)
        Predict: 1.0
     Else (feature 7 not in {32.0,35.0,36.0,38.0,45.0,50.0,52.0,53.0,55.0,56.0,61.0,68.0,69.0,70.0,71.0,72.0,73.0,74.0,77.0,79.0,84.0,85.0,86.
      If (feature 6 in {2.0,4.0,12.0,57.0,66.0,69.0,81.0,83.0,98.0,109.0,122.0,1.0})
       If (feature 7 in {0.0,23.0})
        Predict: 0.0
       Else (feature 7 not in {0.0,23.0})
        Predict: 0.0
      Else (feature 6 not in {2.0,4.0,12.0,57.0,66.0,69.0,81.0,83.0,98.0,109.0,122.0,1.0})
       If (feature 7 in {23.0,14.0})
        Predict: 1.0
       Else (feature 7 not in {23.0,14.0})
        Predict: 1.0
Else (feature 2 > 953.0)
 If (feature 0 in {29.0,25.0,26.0,27.0,0.0})
  If (feature 4 in {10.0,11.0,9.0,7.0,8.0,5.0,2.0,4.0,3.0,0.0})
   If (feature 6 in {44.0,46.0,51.0,61.0,67.0,73.0,77.0,81.0,84.0,94.0,95.0,98.0,105.0,107.0,109.0,113.0,115.0,117.0,119.0,124.0,127.0,129.0,1
    If (feature 7 in {2.0,3.0,5.0,7.0,8.0,11.0,12.0,13.0,16.0,17.0,18.0,19.0,21.0,22.0,25.0,26.0,28.0,29.0,31.0,33.0,38.0,39.0,40.0,47.0,49.0,
     If (feature 6 in {44.0,46.0,49.0,50.0,51.0,54.0,56.0,59.0,61.0,67.0,68.0,70.0,73.0,81.0,84.0,94.0,95.0,98.0,105.0,107.0,109.0,113.0,115.0
      Predict: 0.0
     Else (feature 6 not in {44.0,46.0,49.0,50.0,51.0,54.0,56.0,59.0,61.0,67.0,68.0,70.0,73.0,81.0,84.0,94.0,95.0,98.0,105.0,107.0,109.0,113.0
      If (feature 3 <= 1702.0)
       Predict: 0.0
      Else (feature 3 > 1702.0)
       If (feature 4 in {0.0})
        Predict: 0.0
```

Figure 7: The binary decision tree generated for the air-flight delay analysis (partially shown)

Step 1: Train the decision tree model

To train the decision tree classifier model, we need to have the necessary labels and features:

```
DecisionTreeClassifier dt = new
DecisionTreeClassifier()
      .setLabelCol("indexedLabel")
      .setFeaturesCol("indexedFeatures");
```

Step 2: Convert the indexed labels back to original labels

To create a decision tree pipeline, we need to have the original labels apart from the indexed labels. So, let's do it as follows:

```
IndexToString labelConverter = new
IndexToString()
      .setInputCol("prediction")
      .setOutputCol("predictedLabel")
        .setLabels(labelIndexer.labels());
```

Step 3: Chain the indexer and tree in a single pipeline

Create a new pipeline where the stages are as follows: `labelIndexer`, `featureIndexer`, `dt`, `labelConverter` as follows:

```
Pipeline pipeline = new Pipeline()
      .setStages(new
PipelineStage[]{labelIndexer,
        featureIndexer, dt, labelConverter});
```

Now fit the pipeline using the training set we created in *Step 8* as follows:

```
PipelineModel model = pipeline.fit(trainingData);
```

Testing the model

Here we will test the models as shown in the following steps:

Step 1: Make the prediction on the test dataset

Make the prediction on the test set by transforming the `PipelineModel` and show the performance parameters as follows:

```
Dataset<Row> predictions =
model.transform(testData);
predictions.select("predictedLabel", "label",
"features").show(5);
```

Step 2: Evaluate the model

Evaluate the model by the multiclass classification evaluator and print the accuracy and test error as follows:

```
MulticlassClassificationEvaluator evaluator = new
MulticlassClassificationEvaluator()
      .setLabelCol("indexedLabel")
      .setPredictionCol("prediction")
      .setMetricName("accuracy");
    double accuracy =
evaluator.evaluate(predictions);
    System.out.println("accuracy: "+accuracy);
    System.out.println("Test Error = " + (1.0 -
accuracy));
```

The preceding code segment produces the classification accuracy and test error as follows:

```
Accuracy: 0.7540472721385786
Test Error = 0.24595272786142142
```

Please note that since we randomly split the dataset into training and testing, you might get a different result. The classification accuracy is 75.40%, which is not good, we believe.

However, now it's your turn to use a different classifier and tune before deploying the model. A more details discussion will be carried out in Chapter 7, *Tuning Machine Learning Models,* regarding tuning the ML models.

Step 3: Print the decision tree

Here is the code to print the decision tree:

```
DecisionTreeClassificationModel treeModel =
      (DecisionTreeClassificationModel)
(model.stages()[2]);
System.out.println("Learned classification tree
model:\n" + treeModel.toDebugString());
```

This code segment produces a decision tree as shown in *Figure 7*.

Step 4: Stop the Spark session

Stop the Spark session using the `stop()` method of Spark as follows:

```
spark.stop();
```

This is a good practice that you initiate a Spark session and close or stop it properly to avoid the memory leak in your applications.

Unsupervised learning

In unsupervised learning, data points have no labels related to them; therefore, we need to put labels on them algorithmically. In other words, the correct classes of the training dataset in unsupervised learning are unknown.

Consequently, classes have to be inferred from the unstructured datasets which implies that the goal of an unsupervised learning algorithm is to pre-process the data in some structured ways by describing its structure. The main objective of the unsupervised learning algorithms or techniques is to explore the unknown patterns of the input data that are mostly unlabeled. In this way, it is closely related to the problem of density estimation used in theoretical and applied statistics.

Unsupervised learning, however, also comprehends many other techniques to summarize and explain the key features of the data including exploratory data analysis for finding these hidden patterns, even grouping the data points or features and applying the unsupervised learning technique based on data mining methods for the data pre-processing.

To overcome this obstacle in unsupervised learning, clustering techniques are used typically to group the unlabeled samples based on certain similarity measures, mining hidden patterns towards feature learning.

Tip

For in-depth theoretical knowledge, how the unsupervised algorithms work, please refer to these three books: Bousquet, O.; von Luxburg, U.; Raetsch, G., eds. (2004). *Advanced Lectures on Machine Learning*. Springer-Verlag. ISBN 978-3540231226. Or Duda, Richard O.; Hart, Peter E.; Stork, David G. (2001). *Unsupervised Learning and Clustering*. *Pattern Classification (2nd Ed.)*. Wiley. ISBN 0-471-05669-3 and Jordan, Michael I.; Bishop, Christopher M. (2004). *Neural Networks*. In Allen B. Tucker. Computer *Science Handbook, Second Edition (Section VII: Intelligent Systems)*. Boca Raton, FL: Chapman & Hall/CRC Press LLC. ISBN 1-58488-360-X.

Unsupervised learning example

In clustering, an algorithm groups objects into categories by analyzing similarities between input examples where similar objects or features are clustered and marked using circles around them.

Clustering uses include: **Search results grouping** such as grouping of customers, **anomaly detection** for suspicious pattern finding, **text categorization** for finding useful patterns in the tests, **social network analysis** for finding coherent groups, **data center computing clusters** for finding a way of putting related computers together to improve performance, **astronomic data analysis** for galaxy formation, and **real estate data analysis** for identifying neighborhoods based on similar features. Moreover, clustering uses unsupervised algorithms, which do not have the outputs in advance.

Clustering using the K-means algorithm begins by initializing all the coordinates to centroids. Note that Spark also supports other clustering algorithms such as Gaussian mixture, **Power Iteration Clustering** (**PIC**), **Latent Dirichlet Allocation** (**LDA**), Bisecting k-means, and Streaming k-means. Whereas, the Gaussian mixture is mainly used for expectation minimization as an optimization algorithm, the LDA, on the other hand, is used for the document classification and clustering. PIC is used for the clustering vertices of a graph given pairwise similarities as edge properties. Bisecting K-means is faster than the regular K-means, but it will generally produce a different clustering. Therefore, to keep the discussion simpler we will use the K-means algorithm for our purposes.

Interested readers should refer the Spark ML and Spark MLlib based clustering techniques at https://spark.apache.org/docs/latest/ml-clustering.html and https://spark.apache.org/docs/latest/mllib-clustering.html web pages respectively to get more insights. With every pass of the algorithm, each point is assigned to its nearest centroid based on some distance metric, usually the **Euclidean distance**.

Note that there are other ways to calculate the distance, for example, the **Chebyshev distance** is used to measure the distance by considering only the most significant dimensions. The **Hamming distance algorithm** is used to identify the difference bit by bit of two strings. The **Mahalanobis distance** is used to normalize the covariance matrix to make the distance metric scale-invariant.

The **Manhattan distance** is used to measure the distance following only axis-aligned directions. The **Minkowski distance algorithm** is used to make the Euclidean distance, Manhattan distance, and Chebyshev distance generalize. The **Haversine distance** is used to measure the great-circle distances between two points on a sphere from their longitudes and latitudes. Considering these distance measuring algorithms, it is clear that the Euclidean distance algorithm would be the most appropriate to solve our problem.

The centroids are then updated to be the centers of all the points assigned to it in that pass. This repeats until there is a minimum change in the

centers. The k-means algorithm is an iterative algorithm and works in two steps:

- **Cluster assignment step**: This algorithm will go through each data point and, depending upon which centroid it is nearer to, it will be assigned that centroid and, in turn, the cluster it represents
- **Move centroid step**: This algorithm will take each centroid and move it to the mean of the data points in the cluster

Unsupervised learning with Spark - an example

We will use the *Saratoga NY Homes* downloaded from the URL http://course1.winona.edu/bdeppa/Stat%20425/Datasets.html to demonstrate an example of clustering as an unsupervised learning technique using Spark in Java. The dataset contains several features as follows: Price, Lot Size, Waterfront, Age, Land Value, New Construct, Central Air, Fuel Type, Heat Type, Sewer Type, Living Area, Pct.College, Bedrooms, Fireplaces, Bathrooms, and the number of Rooms. However, among those columns, we have shown only some selected columns in *Table 3*. Note that the original dataset was downloaded and later on converted into a corresponding text file as a tab delimiter:

Price	Lot Size	Water Front	Age	Land Value	Rooms
132500	0.09	0	42	5000	5
181115	0.92	0	0	22300	6
109000	0.19	0	133	7300	8
155000	0.41	0	13	18700	5
86060	0.11	0	0	15000	3
120000	0.68	0	31	14000	8
153000	0.4	0	33	23300	8

170000	1.21	0	23	146000	9
90000	0.83	0	36	222000	8
122900	1.94	0	4	212000	6
325000	2.29	0	123	126000	12

Table 3: Sample data from the "Saratoga NY Homes" dataset

We further took only the first two features (that is, Price and Lot Size) using the Spark feature learning algorithm presented in the previous chapter for simplicity. Our target is to show an exploratory analysis based on these two features for possible neighborhoods of the house located in the same area. First, look at the basic scatter plot diagram based on the value in the dataset:



Figure 8: Cluster of the neighborhoods

It's clearly seen that there are four cluster on the plot marked as circles in *Figure 8*. However, finding a number of clusters is a tricky task. Here, we have the advantage of visual inspection, which is not available for data on hyperplanes or multidimensional data. Now we need to find the same result

using Spark. For simplicity, we will use the K-means clustering API of Spark. The use of raw data and finding feature vectors is shown in *Figure 9*:

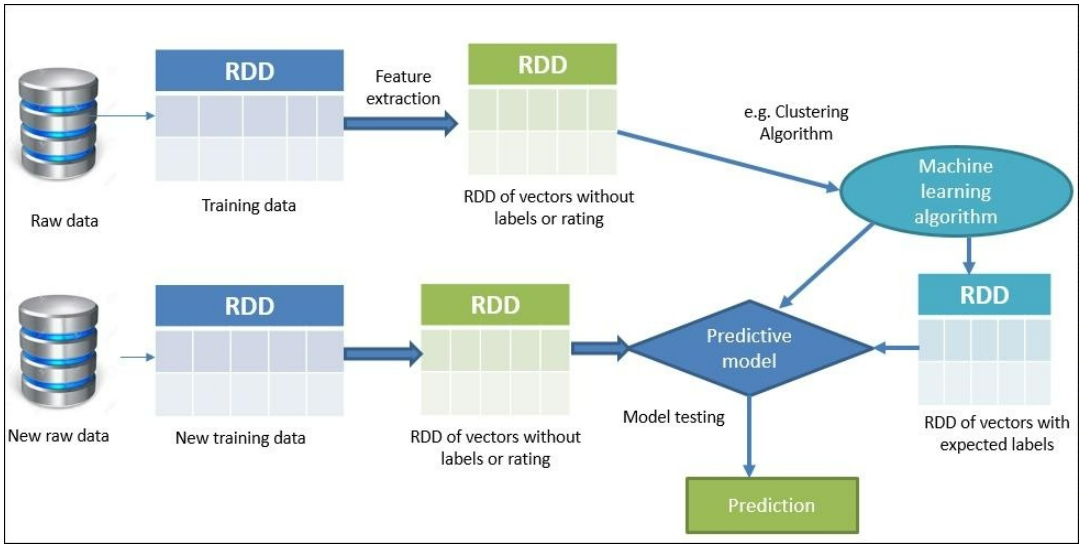


Figure 9: Unsupervised learning using Spark

K-means clustering of the neighborhood

Before performing the feature extraction, we need to load and parse the Saratoga NY Homes dataset. This step also includes: loading packages and related dependencies, reading the dataset as RDD, model training and prediction, collecting the local parsed data, and comparing clustering.

Step 1: Import statistics and related classes

Here is the code to import statistics and related classes:

```
import java.io.Serializable;
import java.util.List;
import org.apache.spark.api.java.JavaRDD;
import
org.apache.spark.api.java.function.Function;
import org.apache.spark.mllib.clustering.KMeans;
import
org.apache.spark.mllib.clustering.KMeansModel;
import org.apache.spark.mllib.linalg.Vector;
import org.apache.spark.mllib.linalg.Vectors;
import org.apache.spark.rdd.RDD;
import org.apache.spark.sql.SparkSession;
```

Step 2: Create the Spark session

Here is the code to create a Spark session:

```
static SparkSession spark = SparkSession
    .builder().appName("JavaLDAExample")
```

```
        .master("local[*]")
        .config("spark.sql.warehouse.dir",
"E:/Exp/")
        .getOrCreate();
```

Step 3: Load the Saratoga NY Homes.txt

Read, parse, and create RDDs from the dataset:

```
RDD<String> data =
spark.sparkContext().textFile("input/Saratoga_
NY_Homes.txt", 2);
```

Step 4:Transform the data into an RDD of dense vectors

If you follow the preceding step carefully, actually we have created the normal RDD. Therefore, that RDD has to be converted into the corresponding `JavaRDD` before mapping into a dense vector using Vector:

```
JavaRDD<Vector> parsedData =
data.toJavaRDD().map(new Function<String,
Vector>() {
      @Override
      public Vector call(String s) throws
Exception {
        String[] sarray = s.split(",");
        double[] values = new
double[sarray.length];
        for (int i = 0; i < sarray.length; i++)
          values[i] =
Double.parseDouble(sarray[i]);
        return Vectors.dense(values);
      }
    });
```

Step 5: Train the model

Train the model by specifying four clusters and five iterations. Just refer the following code for doing that:

```
int numClusters = 4;
int numIterations = 10;
int runs = 2;
KMeansModel clusters =
KMeans.train(parsedData.rdd(), numClusters,
numIterations, runs , KMeans.K_MEANS_PARALLEL());

Now estimate the cost to compute the clsuters as
follows:
double cost =
clusters.computeCost(parsedData.rdd());
```

```
System.out.println("Cost: " + cost);
```

You should receive the results as follows:

```
Cost: 3.60148995801542E12
```

Step 6: Show the cluster centers

```
Vector[] centers = clusters.clusterCenters();
System.out.println("Cluster Centers: ");
for (Vector center : centers)
{
  System.out.println(center);
}
```

The preceding code should produce the center of the clusters as follows:

```
[545360.4081632652,0.9008163265306122,0.1020408163265306,21.7346938775510
[134073.06845637583,0.3820000000000002,0.0026845637583892616,33.726174496
[218726.0625,0.5419711538461538,0.0,25.495192307692307,32579.647435897434
[332859.0580645161,0.6369354838709671,0.025806451612903226,19.80322580645
```

Step 7: Evaluate the model error rate

```
double WSSSE =
clusters.computeCost(parsedData.rdd());
System.out.println("Within Set Sum of Squared
Errors = " + WSSSE);
```

This should produce the result as follows:

```
Within Set Sum of Squared Errors =
3.60148995801542E12
```

Step 8: Predict the cluster for the second element

```
List<Vector> houses = parsedData.collect();
int prediction  =
clusters.predict(houses.get(18));
System.out.println("Prediction: "+prediction);
```

Output prediction: 0

Step 9: Stop the Spark session

Stop the Spark session using the `stop()` method as follows:

```
spark.stop();
```

Step 10: Cluster comparing

Now let's compare the cluster assignments by k-means versus the ones we have done individually. The k-means algorithm gives the cluster IDs starting from 0. Once you inspect the data, you find out the following mapping between the A to D cluster IDs we gave versus K-means in Table 4:

Cluster name	Cluster number	Cluster assignment
A	3	A=>3
B	1	B=>1
C	0	C=>0
D	2	D=>2

Table 4: Cluster assignment for the neighbourhood K-means clustering example

Now, let's pick some of the data from different parts of the chart and predict which cluster it belongs to. Let's look at the house (say 1 as an example) data, which has a lot size of 876 square feet and is priced at $665K:

```
int prediction  =
clusters.predict(houses.get(18));
    System.out.println("Prediction:
"+prediction);
```

[Output] Prediction: 2

That means the house with the preceding properties falls in the cluster 2. You can test the prediction capability with more data of course. Let's do some neighborhood analysis to see what meaning these clusters carry. We can assume that most of the houses in cluster 3 are near downtown. The cluster 2 houses are on hilly terrain for example.

In this example, we dealt with a very small set of features; common sense and visual inspection would also lead us to the same conclusions. However, if you want to acquire more accuracy, of course, you should construct more meaningful features by not only considering only the lot size and the house price but other features like the number of rooms, house age, land value, heating type, and so on.

However, it would not be wise to include the *Waterfront* as a meaningful feature since no house has a water garden in front of the house in this example. We will provide a detailed analysis towards the better accuracy of meaningful a prediction in next chapter, where we will demonstrate these considerations.

The beauty of the k-means algorithm is that it does the clustering on the data with an unlimited number of features. It is a great tool to use when you have a raw data and would like to know the patterns in that data. However, deciding pn the number of clusters prior to doing the experiment might not be successful but sometimes may lead to an overfitting problem or an underfitting problem.

Tip

To overcome the aformentioned limitation of the K-means, we have some more robust algorithms like **Markov Chain Monte Carlo** (**MCMC** , see also https://en.wikipedia.org/wiki/Markov_chain_Monte_Carlo) presented in Tribble, Seth D., *Markov chain Monte Carlo algorithms using completely uniformly distributed driving sequences*, Diss. Stanford University, 2007. Moreover, a more technical discussion can be found at the URL http://www.autonlab.org/tutorials/kmeans11.pdf too.

Recommender system

A recommender system is an original killer application which is a subclass of an information filtering system that looks to predict the rating or preference from the users that they usually provide to an item. The concept of recommender systems has become very common in recent years and has been subsequently applied in different applications. The most popular ones are probably products (for example, movies, music, books, research articles), news, search queries, social tags, and so on). Recommender systems can be typed into four categories as stated in Chapter 2, *Machine Learning Best Practices.* These are shown in *Figure 10*:

- **The collaborative filtering system**: This is the accumulation of a consumer's preferences and recommendations to other users based on likeness in behavioral patterns **Content-based systems**: Here the supervised machine learning is used to persuade a classifier to distinguish between interesting and uninteresting items for the users
- **Hybrid recommender systems**: This is a recent research and hybrid approach (that is, combining collaborative filtering and content-based filtering)
- **Knowledge-based systems**: Here knowledge about users and products are used to understand what fulfils a user's requirements, using a perception tree, decision support systems, and case-based reasoning:

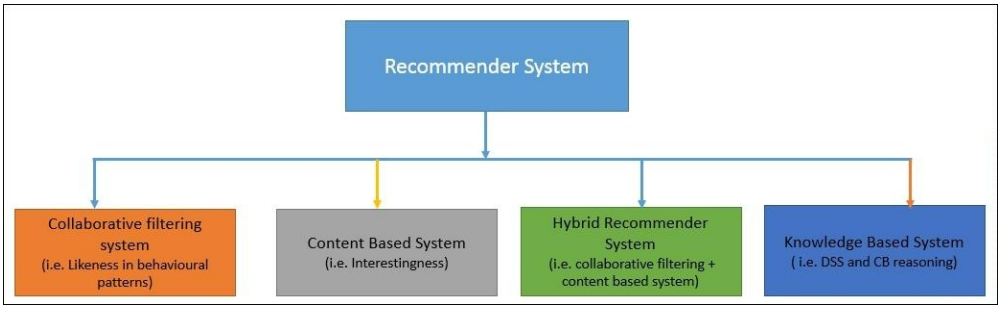


Figure 10: Hierarchy of the recommendation systems

From the technical viewpoint, we can further categorize them as follows:

- The item **hierarchy** is the weakest one where it is naively assuming that one item is correlated to another, for example, if you buy a printer, it is more likely that you will buy the ink. Previously this approach was used by **BestBuy**
- **Attribute-based recommendation**: Assumes that you like action movies starring Sylvester Stallone, therefore, you might be like the Rambo series. **Netflix** used to use this approach

- **Collaborative filtering** (U**ser-user similarity):** This assumes and exemplifies those people like you who brought baby milk also bought diapers. Target use this approach
- **Collaborative filtering** (**Item-item similarity)**: This assumes and exemplifies that people who like Godfather series also like Scarface. Netflix currently uses this approach
- **Social, interest and graph-based approach**: This assumes for example that your friend who likes Michel Jackson will also like *Just Beat It*. The tech giant like **LinkedIn** and **Facebook** use this approach
- **Model-based approach**: This uses an advanced algorithm such as **SVM**, **LDA**, and **SVD** based on the implicit features

As shown in *Figure 11*, the model-based recommender system that widely used advanced algorithms such as SVM, LDA, or SVD is the most robust approach in the recommender system class:

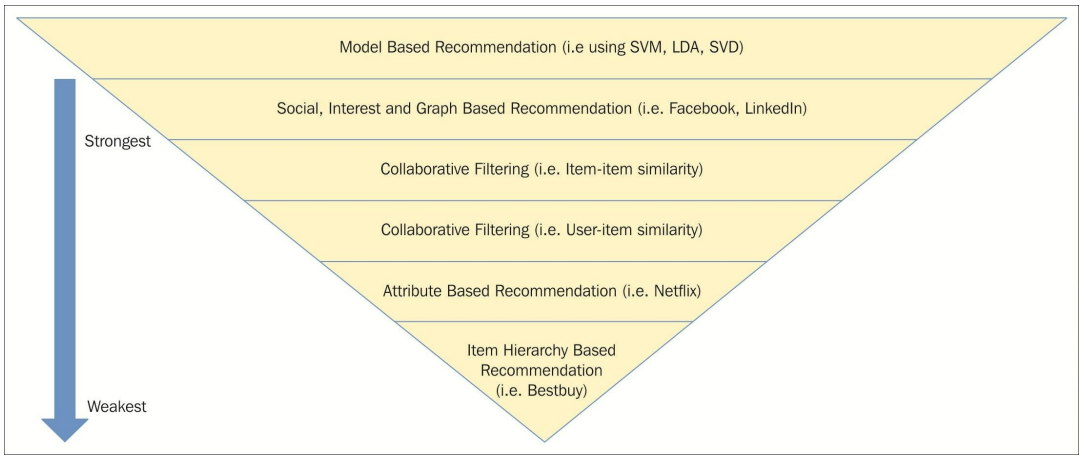


Figure 11: The recommender system from the technical point of view

Collaborative filtering in Spark

As already mentioned, the collaborative filtering techniques are commonly used for recommender systems. However, Spark MLlib currently supports model-based collaborative filtering only. Here, users and products are described by a small set of latent factors. The latent factors are later used for making the prediction of the missing entries. According to the Spark API reference for the collaborative filtering on http://spark.apache.org/docs/latest/mllib-collaborative-filtering.html: the **Alternating Least Squares** (**ALS**) (also known as non-linear least square,that is, NLS; see more at https://en.wikipedia.org/wiki/Non-linear_least_squares) algorithm is used to learn these latent factors by considering the following parameters:

- `numBlocks` is the number of blocks used for the parallelized computation using the native LAPACK

- `rank` is the number of latent factors during the machine learning model building
- `iterations` are the number of iterations needed to gain more accurate predictions
- `lambda` signifies the regularization parameter for the ALS algorithm
- `implicitPrefs` specifies which feedback to be used (explicit feedback ALS variant or one adapted for implicit feedback data)
- `alpha` specifies the baseline confidence in preference observations for the ALS algorithm

At first, the ALS, which is an iterative algorithm, is used to model the rating matrix as the multiplication of low-ranked users and product factors. After that, the learning task is done by using these factors by minimizing the reconstruction error of the observed ratings.

However, the unknown ratings can successively be calculated by multiplying these factors together. The approach for the move recommendation or any other recommendation based on the collaborative filtering technique used in the Spark MLlib has been proven a high performer with high prediction accuracy and is scalable for the billions of ratings on commodity clusters used by companies such as Netflix. In following this way, a company such as Netflix can recommend movies to its subscribers based on the predicted ratings. The ultimate target is to increase the sales and of course the customer satisfaction.

For brevity and page limitation, we will not show the movie recommendations using the collaborative filtering approach in this chapter. However, a step-by-step example using Spark will be shown in Chapter 9, *Advanced Machine Learning with Streaming and Graph Data*.

> **Tip**
>
> For the time being, interested readers are advised to visit the Spark website for the latest API and codes for the same at this URL: http://spark.apache.org/docs/latest/mllib-collaborative-filtering.html, where an example has been presented to show the sample movie recommendations using the ALS algorithm.

Advanced learning and generalizations

In this section, we will discuss some advanced aspects of learning, for example how we can generalize the supervised learning techniques for semi-supervised learning, active learning, structured prediction, and reinforcement learning. Moreover, reinforcement and semi-supervised learning will be discussed in brief.

Generalizations of supervised learning

There are several ways in which the standard supervised learning problem can be generalized:

- **Semi-supervised learning**: In this generalization technique, only the required output values for selected features are provided for a subset of the training data to build and evaluate the machine learning model. On the other hand, the remaining data is kept unchanged or unlabeled.
- **Active learning**: In contrast, in active learning, algorithms typically interactively collect new features by making queries to a human user instead of assuming all the training features are given. Consequently, the queries used here are based on unlabeled data. Interestingly, it is also an example that combines semi-supervised learning with active learning.
- **Structured prediction**: Sometimes the desired features need to be extracted or selected from complex objects like a parse tree or a labelled graph, and then the standard supervised or unsupervised methods must be improved towards adaptability for making it generalized. To be more precise, for example when a supervised machine learning technique tries to predict structured or unstructured texts such as translating NLP sentences into syntactic representation, structured prediction evolves that need to handle a large-scale parse tree. To make this task easier, often the structured SVMs or Markov logic networks or constrained conditional models are used that technically extend and update the classical supervised learning algorithms.
- **Learning to rank**: Machine-learned ranking is required when the input itself is a subset of objects and the desired output is a ranking of those objects, then the standard methods must be extended or improved similarly to the structure prediction technique.

Tip

Interested readers can refer to these two URLs: https://en.wikipedia.org/wiki/Learning_to_rank and https://en.wikipedia.org/wiki/Structured_prediction, where a more details discussion can be found.

Summary

We have discussed some supervised, unsupervised, and recommender systems from a theoretical and Spark's perspective. However, there are numerous examples for the supervised, unsupervised, reinforcement or recommendation systems too. Nevertheless, we have tried to present some simple examples for the sake of simplicity.

We will provide more insights on these examples in Chapter 6, *Building Scalable Machine Learning Pipelines*. More feature incorporation, extraction, selection using Spark ML and Spark MLlib pipelines, model scaling, and tuning will be discussed too. We also intend to provide some examples including data collection to model building and prediction.

Chapter 6. Building Scalable Machine Learning Pipelines

The ultimate goal of machine learning is to make a machine that can automatically build models from data without requiring tedious and time-consuming human involvement and interaction. Therefore, this chapter guides the readers through creating some practical and widely used machine learning pipelines and applications using Spark MLlib and Spark ML. Both APIs will be described in detail, and a baseline use case will also be covered for both. Then we will focus on scaling up the ML application so that it can cope with increasing data loads. After reading all the sections in this chapter, readers will be able to differentiate between both APIs and select the one which best fits their requirements. In a nutshell, the following topics will be covered throughout this chapter:

- Spark machine learning pipeline APIs
- Cancer-diagnosis pipeline with Spark
- Cancer-prognosis pipeline with Spark
- Market basket analysis with Spark Core
- OCR pipeline with Spark
- Topic modeling using Spark MLlib and ML
- Credit-risk-analysis pipeline with Spark
- Scaling the ML pipelines
- Tips and performance considerations

Spark machine learning pipeline APIs

MLlib's goal is to make practical machine learning (ML) scalable and easy. Spark introduces the pipeline API for the easy creation and tuning of practical ML pipelines. As discussed in Chapter 4, *Extracting Knowledge through Feature Engineering*, a practical ML pipeline involves a sequence of data collection, pre-processing, feature extraction, feature selection, model fitting, validation, and model evaluation stages. For example, classifying the text documents might involve text segmentation and cleaning, extracting features, and training a classification model with cross-validation toward tuning. Most ML libraries are not designed for distributed computation, or they do not provide native support for pipeline creation and tuning.

Dataset abstraction

As described in Chapter 1, *Introduction to Data Analytics with Spark*, when running SQL from within another programming language, the results return as a DataFrame. A DataFrame is a distributed collection of data organized into named columns. A Dataset, on the other hand, is an interface that tries to provide the benefits of RDDs out of the Spark SQL.

A Dataset can be constructed from JVM objects, which can be used both in Scala and Java. In the Spark pipeline design, a dataset is represented by Spark SQL's Dataset. An ML pipeline involves a number of the sequence of Dataset transformations and models. Each transformation takes an input dataset and outputs the transformed dataset, which becomes the input to the next stage.

Consequently, the data import and export are the start and end point of an ML pipeline. To make these easier, Spark MLlib and Spark ML provide import and export utilities of a Dataset, DataFrame, RDD, and model, for several application-specific types, including:

- LabeledPoint for classification and regression
- LabeledDocument for cross-validation and **Latent Dirichlet Allocation** (**LDA**)
- Rating and ranking for collaborative filtering

However, real datasets usually contain numerous types, such as user ID, item IDs, labels, timestamps, and raw records.

Unfortunately, the current utilities of Spark implementation cannot easily handle datasets consisting of these types, especially time-series datasets. If you recall the section *Machine learning pipeline - an overview*, in Chapter 4, *Extracting Feature through Feature Engineering*, feature transformation usually forms the majority of a practical ML pipeline. A feature transformation can be viewed as appending or dropping a new column created from existing columns.

In *Figure 1*, *Text processing for machine learning model*, you will see that the text tokenizer breaks a document into a bag of words. After that, the TF-IDF algorithm converts a bag of words into a feature vector. During the transformations, the labels need to be preserved for the model-fitting stage:

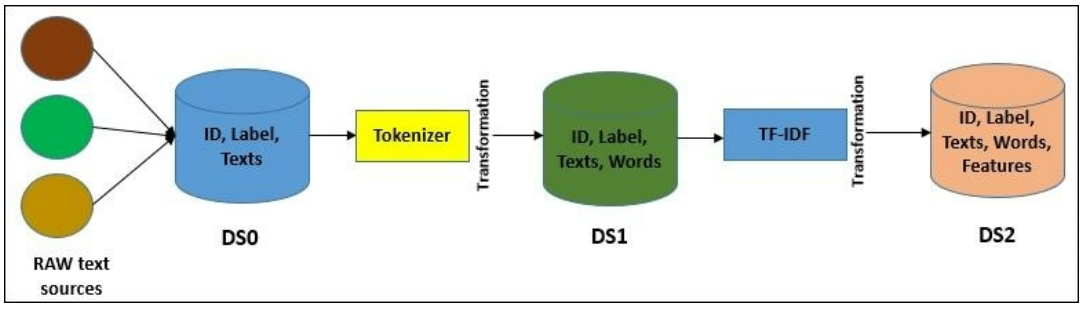


Figure 1: Text processing for machine learning model (DS indicates data sources)

If you recall *Figure 5* and *Figure 6* from Chapter 4, *Extracting Feature through Feature Engineering*, ID, text, and words are conceded during the transformations steps. They are useful in making predictions and model inspection. However, they are actually unnecessary for model fitting to state. According to a Databricks blog on ML Pipeline at https://databricks.com/blog/2015/01/07/ml-pipelines-a-new-high-level-api-for-mllib.html, it doesn't provide much information if the prediction dataset only contains the predicted labels.

Consequently, if you want to inspect the prediction metrics, such as the accuracy, precision, recall, weighted true positives, and weighted false positives, it is quite useful to look at the predicted labels along with the raw input text and tokenized words. The same recommendation also applies to other machine learning applications using Spark ML and Spark MLlib, too.

Therefore, an easy conversion between RDDs, Dataset, and DataFrames has been made possible for in-memory, disk, or external data sources such as Hive and Avro. Although creating new columns from existing columns is easy with user-defined functions, the manifestation of Dataset is a lazy operation.

In contrast, the Dataset supports only some standard data types. However, to increase the usability and for making a better fit for the machine learning model, Spark has also added the support for the Vector type as a user-defined type that supports both dense and sparse feature vectors under the `mllib.linalg.DenseVector` and `mllib.linalg.Vector`.

> **Tip**
>
> Complete DataFrame, Dataset, and RDD examples in Java, Scala, and Python can be found under the `examples/src/main/` folder under the Spark distribution. Interested readers can refer to Spark SQL's user guide at http://spark.apache.org/docs/latest/sql-programming-guide.html to learn more about DataFrame, Dataset, and the operations they support.

Pipeline

Spark provides the pipeline API under Spark ML. As previously stated, a pipeline is comprised of a sequence of stages consisting of transformers and estimators. There are two basic types of pipeline stages, called Transformer and Estimator.

A transformer takes a dataset as an input and produces an augmented dataset as the output so that the output can be fed to the next step. For example, **Tokenizer** and **HashingTF** are two transformers. Tokenizer transforms a dataset with text into a dataset with tokenized words. A HashingTF, on the other hand, produces the term frequencies. The concept of tokenization and HashingTF is commonly used in text mining and text analytics.

On the contrary, an estimator must be the first on the input dataset to produce a model. In this case, the model itself will be used as the transformer for transforming the input dataset into the augmented output dataset. For example, a **Logistic Regression** or linear regression can be used as an estimator after fitting the training dataset with corresponding labels and features.

After that, it produces a logistic or linear regression model. It implies that developing a pipeline is easy and simple. Well, all you need is to declare the required stages, then configure the related stage's parameters; finally, chain them in a pipeline object, as shown in *Figure 2*:

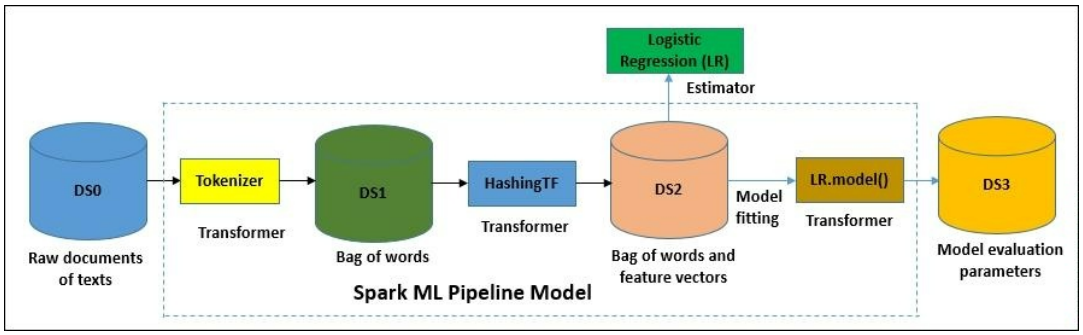


Figure 2: Spark ML pipeline model using logistic regression estimator (DS indicates data store and the steps inside the dashed line only happen during pipeline fitting)

If you look at *Figure 2*, the fitted model consists of a tokenizer, a hashingTF feature extractor, and a fitted logistic regression model. The fitted pipeline model acts as a transformer that can be used for prediction, model validation, model inspection, and finally, model deployment. However, increasing the performance in terms of prediction accuracy, the model itself needs to be tuned. We will discuss more about how to tune a machine learning model in Chapter 7, *Tuning Machine Learning Model*.

To show the pipelining technique more practically, the following section shows how to create a practical pipeline for cancer diagnosis using Spark ML and MLlib.

Cancer-diagnosis pipeline with Spark

In this section, we will look at how to develop a cancer-diagnosis pipeline with Spark ML and MLlib. A real dataset will be used to predict the probability of breast cancer, which is almost curable since the culprit genes for this cancer type have already been identified successfully. However, we would like to argue about this cancer type since in third world countries in Africa and Asia it is still a lethal disease.

> **Tip**
>
> We suggest the readers keep an open mind about the outcome or the status of this disease, as we will just show how the Spark ML API can be used to predict cancer by integrating and combining datasets from the Wisconsin Breast Cancer (original), **Wisconsin Diagnosis Breast Cancer** (**WDBC**), and **Wisconsin Prognosis Breast Cancer** (**WPBC**) datasets from the following website: http://archive.ics.uci.edu/ml.

Breast-cancer-diagnosis pipeline with Spark

In this subsection, we will develop a step-by-step cancer diagnosis pipeline. The steps include a background study of breast cancer, dataset collection, data exploration, problem formalization, and Spark-based implementation.

Background study

According to Salama et al. (*Breast Cancer Diagnosis on Three Different Datasets Using Multi-Classifiers, International Journal of Computer and Information Technology* (*2277 - 0764*) *Volume 01- Issue 01, September 2012*), breast cancer comes in fourth position after thyroid cancer, melanoma, and lymphoma, in women between 20 and 29 years.

Breast cancer develops from breast tissue that mutates due to several factors including sex, obesity, alcohol, family history, lack of physical exercise, and so on. Furthermore, according to statistics by **The Centre for Diseases Control and Prevention** (**TCDCP**) (https://www.cdc.gov/cancer/breast/statistics/), in 2013, a total of 230,815 women and 2,109 men were diagnosed with breast cancer across the USA. Unfortunately, 40,860 women and 464 men died from it.

Research has found that about 5-10% cases are due to some genetic inheritance from parents, including BRCA1 and BRCA2 gene mutations and so on. An early diagnosis could help to save thousands of breast cancer sufferers around the globe. Although the culprit genes have been identified, chemotherapy has not proven very effective. Gene silencing is becoming popular, but more research is required.

As mentioned previously, the learning tasks in machine learning depend heavily on classification, regression, and clustering techniques. Moreover, traditional data-mining techniques are being applied along with these machine learning techniques, which are the most essential and important task. Therefore, by integrating with Spark, these applied techniques are gaining wide acceptance and adoption in the area of biomedical data analytics. Furthermore, numerous experiments are being performed on biomedical datasets using multiclass and multilevel classifiers and feature-selection techniques toward cancer diagnosis and prognosis.

Dataset collection

The Cancer Genome Atlas (**TCGA**), **Catalogue of Somatic Mutations in Cancer** (**COSMIC**), **International Cancer Genome Consortium** (**ICGC**) is the most widely used cancer and tumor-related dataset for research purposes. These data sources have been curated from world-renowned institutes such as MIT, Harvard, Oxford, and others. However, the datasets that are available are unstructured, complex, and multidimensional. Therefore, we cannot use them directly to show how to apply large-scale machine learning techniques to them. The reason is that these datasets require lots of pre-processing and cleaning, which requires lots of pages.

After practising this application, we believe readers will be able to apply the same technique for any kind of biomedical dataset for cancer diagnosis. Due to the page limitation, we should use simpler datasets that are structured and manually curated for machine learning application development and of course, many of them show good classification accuracy.

For example, the Wisconsin Breast Cancer datasets from the UCI Machine Learning Repository available at http://archive.ics.uci.edu/ml contains data that was donated by researchers at the University of Wisconsin and includes measurements from digitized images of a fine-needle aspiration of a breast mass. The values represent characteristics of the cell nuclei present in the digital image described in the following subsection.

Tip

To read more about the Wisconsin breast cancer data, refer to the authors' publication: *Nuclear feature extraction for breast tumor diagnosis. IS&T/SPIE 1993 International Symposium on Electronic Imaging: Science and Technology, volume 1905, pp 861-870 by W.N. Street, W.H. Wolberg, and O.L. Mangasarian, 1993*.

Dataset description and preparation

As shown in the **Wisconsin Breast Cancer Dataset** (**WDBC**) manual available at https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/wdbc.names, the Clump thickness benign cells tend to be grouped in monolayers, while cancerous cells are often grouped in multilayers. Therefore, all the features and fields mentioned in the manual are important and before applying the machine learning technique since these features will help to identify if a particular cell is cancerous or not.

The breast cancer data includes 569 samples of cancer biopsies, each with 32 features. One feature is the identification number of the patient, another is the cancer diagnosis, labeled as benign or malignant, and the remainder are numeric-valued is called bio-assay that was identified in the molecular laboratory works. The diagnosis is coded as either M to indicate malignant or B to indicate benign with regard to the cancer diagnosis.

The Class distribution is as follows: Benign: 357 (62.74%) and Malignant: 212 (37.25%). The training and test dataset will be prepared following the dataset description given here. The 30 numeric measurements include the mean, standard error, and worst, which is the mean of the three largest values. Field 3 is the mean radius, 13 is the Radius SE, and 23 is the Worst Radius. The 10 real-valued features are computed for each cell nucleus by means of different characteristics of the digitized cell nuclei described in *Table 1, 10 real-valued features and their descriptions*:

No.	Value	Explanation
1	Radius	Mean of distances from center to points on the perimeter
2	Texture	Standard deviation of gray-scale values

3	Perimeter	The perimeter of the cell nucleus
4	Area	Area of the cell nucleus covering the perimeter
5	Smoothness	Local variation in radius lengths
6	Compactness	Calculated as follows: (Perimeter)^2 / area - 1.0
7	Concavity	Severity of concave portions of the contour
8	Concave points	Number of concave portions of the contour
9	Symmetry	Indicates if the cell structure is symmetrical
10	Fractal dimension	Calculated as: coastline approximation - 1

Table 1: 10 real-valued features and their descriptions

All feature values are recorded with four significant digits and there are no missing or NULL values. Therefore, we don't need to perform any data cleaning. However, from the previous description, it's really difficult for someone to get any good knowledge of the data. For example, you are unlikely to know how each field relates to benign or malignant masses unless you are an oncologist. These patterns will be revealed as we continue the machine learning process. A sample snapshot of the dataset is shown in *Figure 3*:

```
+--------------------------------------------------------------------------------------------------
|value
+--------------------------------------------------------------------------------------------------
|842302,M,17.99,10.38,122.8,1001,0.1184,0.2776,0.3001,0.1471,0.2419,0.07871,1.095,0.9053,8.589
|842517,M,20.57,17.77,132.9,1326,0.08474,0.07864,0.0869,0.07017,0.1812,0.05667,0.5435,0.7339,3
|84300903,M,19.69,21.25,130,1203,0.1096,0.1599,0.1974,0.1279,0.2069,0.05999,0.7456,0.7869,4.58
|84348301,M,11.42,20.38,77.58,386.1,0.1425,0.2839,0.2414,0.1052,0.2597,0.09744,0.4956,1.156,3.4
|84358402,M,20.29,14.34,135.1,1297,0.1003,0.1328,0.198,0.1043,0.1809,0.05883,0.7572,0.7813,5.4
|843786,M,12.45,15.7,82.57,477.1,0.1278,0.17,0.1578,0.08089,0.2087,0.07613,0.3345,0.8902,2.217
|844359,M,18.25,19.98,119.6,1040,0.09463,0.109,0.1127,0.074,0.1794,0.05742,0.4467,0.7732,3.18,
|84458202,M,13.71,20.83,90.2,577.9,0.1189,0.1645,0.09366,0.05985,0.2196,0.07451,0.5835,1.377,3
|844981,M,13,21.82,87.5,519.8,0.1273,0.1932,0.1859,0.09353,0.235,0.07389,0.3063,1.002,2.406,24
|84501001,M,12.46,24.04,83.97,475.9,0.1186,0.2396,0.2273,0.08543,0.203,0.08243,0.2976,1.599,2.
|845636,M,16.02,23.24,102.7,797.8,0.08206,0.06669,0.03299,0.03323,0.1528,0.05697,0.3795,1.187,
|84610002,M,15.78,17.89,103.6,781,0.0971,0.1292,0.09954,0.06606,0.1842,0.06082,0.5058,0.9849,3
|846226,M,19.17,24.8,132.4,1123,0.0974,0.2458,0.2065,0.1118,0.2397,0.078,0.9555,3.568,11.07,11
|846381,M,15.85,23.95,103.7,782.7,0.08401,0.1002,0.09938,0.05364,0.1847,0.05338,0.4033,1.078,2
|84667401,M,13.73,22.61,93.6,578.3,0.1131,0.2293,0.2128,0.08025,0.2069,0.07682,0.2121,1.169,2.
|84799002,M,14.54,27.54,96.73,658.8,0.1139,0.1595,0.1639,0.07364,0.2303,0.07077,0.37,1.033,2.8
|848406,M,14.68,20.13,94.74,684.5,0.09867,0.072,0.07395,0.05259,0.1586,0.05922,0.4727,1.24,3.1
|84862001,M,16.13,20.68,108.1,798.8,0.117,0.2022,0.1722,0.1028,0.2164,0.07356,0.5692,1.073,3.8
|849014,M,19.81,22.15,130,1260,0.09831,0.1027,0.1479,0.09498,0.1582,0.05395,0.7582,1.017,5.865
|8510426,B,13.54,14.36,87.46,566.3,0.09779,0.08129,0.06664,0.04781,0.1885,0.05766,0.2699,0.788
+--------------------------------------------------------------------------------------------------
only showing top 20 rows
```

Figure 3: Snapshot of the data (partial)

Problem formalization

Figure 4, *The breast cancer diagnosis and prognosis pipeline model*, describes the proposed breast cancer diagnosis model. The model consists of two phases, namely, the training and testing phases:

- The training phase includes four steps: data collection, pre-processing, feature extraction, and feature selection
- The testing phase includes the same four steps as the training phase with the addition of the classification step

In the data-collection step, first the pre-processing is done to check if there is an unwanted value or any values are missing. We have already mentioned that there are no missing values. However, it is always good practice to check, since even the unwanted value of a special character could halt the whole training process. After that, the feature engineering step is done through the feature extraction and selection process for determining the correct input vector for the subsequent logistic or linear regression classifier:

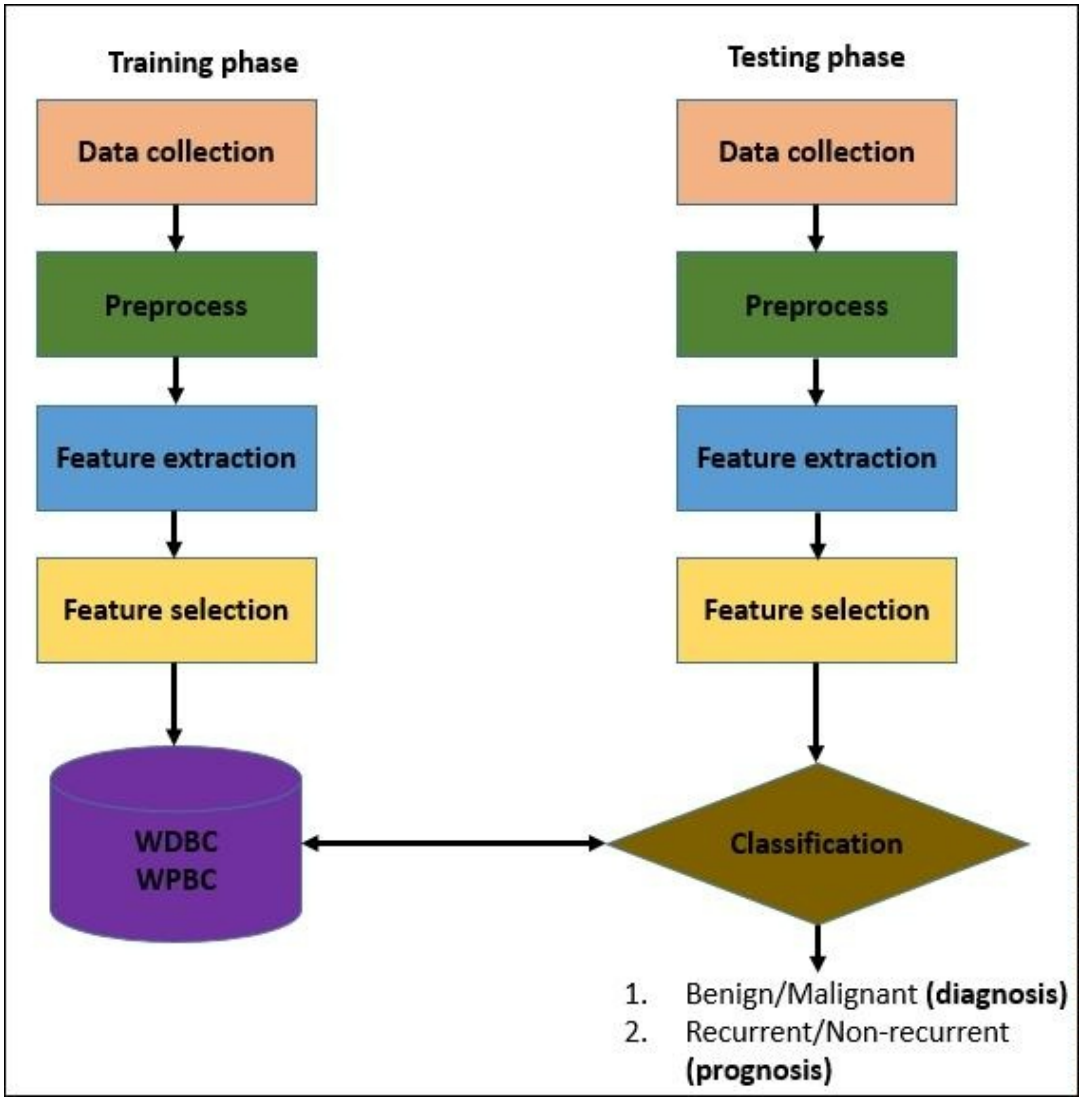


Figure 4: The breast cancer diagnosis and prognosis pipeline model

This helps to make a decision regarding the class associated to the pattern vectors. Based on either feature selection or feature extraction, the dimensionality reduction technique is accomplished. However, please note that we will not use any formal dimensionality reduction algorithms to develop this application. For more on dimensionality reduction, you can refer to the *Dimensionality reduction* section in Chapter 4, *Extracting Knowledge through Feature Engineering*.

In the classification step, a logistic regression classifier is applied to get the best result for the diagnosis and prognosis of the tumor.

Developing a cancer-diagnosis pipeline with Spark ML

As mentioned previously, the details of the attributes found in the WDBC dataset at https://archive.ics.uci.edu/ml/machine-learning-databases/breast-

cancer-wisconsin/breast-cancer-wisconsin.names include patient ID, diagnosis (M = malignant, B = benign), and 10 real-valued features are computed for each cell nucleus, as described in *Table 1*, *10 real-valued features and their description*.

These features are computed from a digitized image of a **fine needle aspiration** (**FNA**) of a breast mass, since we have enough knowing about the dataset. In this subsection, we will look at how to develop a breast cancer diagnosis machine learning pipeline step-by-step including taking the input of the dataset to prediction in the 10 steps described in *Figure 4*, as a data workflow.

Step 1: Import the necessary packages/libraries/APIs

Here is the code to import the packages:

```
import org.apache.spark.api.java.JavaRDD;
import
org.apache.spark.api.java.function.Function;
import org.apache.spark.ml.Pipeline;
import org.apache.spark.ml.PipelineModel;
import org.apache.spark.ml.PipelineStage;
import
org.apache.spark.ml.classification.LogisticRegression;
import org.apache.spark.ml.feature.LabeledPoint;
import org.apache.spark.ml.linalg.DenseVector;
import org.apache.spark.ml.linalg.Vector;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;
```

Step 2: Initialize Spark session

A Spark session can be initialized with the help of the following code:

```
static SparkSession spark = SparkSession
        .builder()

 .appName("BreastCancerDetectionDiagnosis")
       .master("local[*]")
       .config("spark.sql.warehouse.dir",
"E:/Exp/")
       .getOrCreate();
```

Here we set the application name as `BreastCancerDetectionDiagnosis`, and the master URL as `local`. The Spark Context is the entry point of the program. Please set these parameters accordingly.

Step 3: Take the breast cancer data as input and prepare JavaRDD out of the data

Here is the code to prepare `JavaRDD`:

```
  String path = "input/wdbc.data";
  JavaRDD<String> lines =
spark.sparkContext().textFile(path,
3).toJavaRDD();
```

To learn more about the data, please refer to *Figure 3*: *Snapshot of the data (partial*.

Step 4: Create LabeledPoint RDDs for regression

Create `LabeledPoint` RDDs for diagnosis (B = benign and M= Malignant):

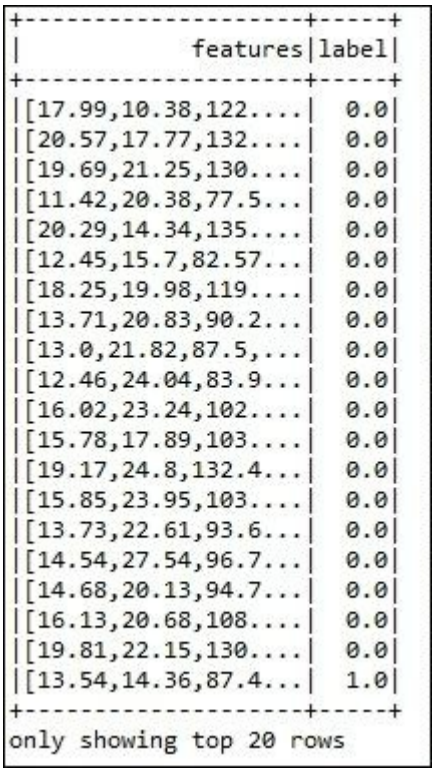
```
JavaRDD<LabeledPoint> linesRDD = lines
        .map(new Function<String, LabeledPoint>()
{
          public LabeledPoint call(String lines)
{
            String[] tokens = lines.split(",");
            double[] features = new double[30];
            for (int i = 2; i < features.length;
i++) {
              features[i - 2] =
Double.parseDouble(tokens[i]);
            }
            Vector v = new DenseVector(features);
            if (tokens[1].equals("B")) {
              return new LabeledPoint(1.0, v); //
benign
            } else {
              return new LabeledPoint(0.0, v); //
malignant
            }
          }
        });
```

Step 5: Create the Dataset of Row from the linesRDD and show the top features

Here is the code illustrated:

```
Dataset<Row> data =
spark.createDataFrame(linesRDD,LabeledPoint.class);
data.show();
```

The following figure shows the top features and their corresponding labels:



```
+--------------------+-----+
|            features|label|
+--------------------+-----+
|[17.99,10.38,122....|  0.0|
|[20.57,17.77,132....|  0.0|
|[19.69,21.25,130....|  0.0|
|[11.42,20.38,77.5...|  0.0|
|[20.29,14.34,135....|  0.0|
|[12.45,15.7,82.57...|  0.0|
|[18.25,19.98,119....|  0.0|
|[13.71,20.83,90.2...|  0.0|
|[13.0,21.82,87.5,...|  0.0|
|[12.46,24.04,83.9...|  0.0|
|[16.02,23.24,102....|  0.0|
|[15.78,17.89,103....|  0.0|
|[19.17,24.8,132.4...|  0.0|
|[15.85,23.95,103....|  0.0|
|[13.73,22.61,93.6...|  0.0|
|[14.54,27.54,96.7...|  0.0|
|[14.68,20.13,94.7...|  0.0|
|[16.13,20.68,108....|  0.0|
|[19.81,22.15,130....|  0.0|
|[13.54,14.36,87.4...|  1.0|
+--------------------+-----+
only showing top 20 rows
```

Figure 5: Top features and their corresponding labels

Step 6: Split the Dataset to prepare the training and test sets

Here we split the original data frame into training and test set as 60% and 40%, respectively. Here, `12345L` is the seed value. This value signifies that the split will be the same every time, so that the ML model produces the same result in each iteration. We follow the same conversion in each chapter for preparing the test and training set:

```
Dataset<Row>[] splits = data.randomSplit(new
double[] { 0.6, 0.4 }, 12345L);
Dataset<Row> trainingData = splits[0];
Dataset<Row> testData = splits[1];
```

To see a quick snapshot of these two sets just write `trainingData.show()` and `testData.show()` for training and test sets, respectively.

Step 7: Create a Logistic Regression classifier

Create a logistic regression classifier by specifying the max iteration and regression parameter:

```
LogisticRegression logisticRegression = new
LogisticRegression()
```

```
                                .setMaxIter(100)
                                   .setRegParam(0.01)

        .setElasticNetParam(0.4);
```

> **Tip**
>
> Logistic regression typically takes three parameters: the number of max iteration, the regression parameter, and the elastic-net regularization. See the following lines to get a clearer idea:

```
LogisticRegression lr = new
LogisticRegression().setMaxIter(100)
.setRegParam(0.01).setElasticNetParam(0.4);
```

> **Tip**
>
> The preceding statements create a logistic regression model `lr` with max iteration `100`, regression parameter `0.01`, and elastic net parameter `0.4`.

Step 8: Create and train the pipeline model

Here is the code illustrated:

```
Pipeline pipeline = new Pipeline().setStages(new
PipelineStage[] {logisticRegression});
PipelineModel model = pipeline.fit(trainingData);
```

Here we have created a pipeline whose stages are defined by the logistic regression stage, which is also an estimator we have just created. Note that you could try creating the Tokenizer and HashingTF stages if you are dealing with a text dataset.

However, in this cancer dataset, all of our values are numeric. Therefore, we don't create such stages to be chained to the pipeline.

Step 9: Create a Dataset, transform the model and prediction

Create a Dataset of type Row and transform the model to do the prediction based on the test dataset:

```
Dataset<Row>
predictions=model.transform(testData);
```

Step 10: Show the prediction with prediction precision

```
predictions.show();
long count = 0;
for (Row r : predictions.select("features",
"label", "prediction").collectAsList()) {
    System.out.println("(" + r.get(0) + ", " +
r.get(1) + r.get(2) + ", prediction=" +
r.get(2));
      count++;
    }
```

```
+--------------------+-----+--------------------+--------------------+----------+
|            features|label|       rawPrediction|         probability|prediction|
+--------------------+-----+--------------------+--------------------+----------+
|[6.981,13.43,43.7...|  1.0|[-10.707317184082...|[2.23800736454126...|       1.0|
|[8.196,16.84,51.7...|  1.0|[-7.5709897740070...|[5.14917006029287...|       1.0|
|[8.219,20.7,53.27...|  1.0|[-5.7650694094771...|[0.00312537912609...|       1.0|
|[8.598,20.98,54.6...|  1.0|[-6.0826303242234...|[0.00227696949283...|       1.0|
|[8.671,14.45,54.4...|  1.0|[-9.3127885577172...|[9.02543451561644...|       1.0|
|[8.95,15.76,58.74...|  1.0|[-7.8442635647809...|[3.91840580945936...|       1.0|
|[9.465,21.01,60.1...|  1.0|[-5.3869369611256...|[0.00455512415640...|       1.0|
|[9.731,15.34,63.7...|  1.0|[-6.7888569970452...|[0.00112498832782...|       1.0|
|[9.787,19.94,62.1...|  1.0|[-6.9646832573330...|[9.43770558758242...|       1.0|
|[10.51,20.19,68.6...|  1.0|[-7.2510504921270...|[7.08925815598015...|       1.0|
|[10.66,15.15,67.4...|  1.0|[-6.9907403957066...|[9.19518552611469...|       1.0|
|[10.75,14.97,68.2...|  1.0|[-5.4227257325096...|[0.00439568847536...|       1.0|
|[11.08,14.71,70.2...|  1.0|[-7.3299540470967...|[6.55174178703067...|       1.0|
|[11.15,13.08,70.8...|  1.0|[-6.1783567710851...|[0.00206954095637...|       1.0|
|[11.28,13.39,73.0...|  1.0|[-3.8979767014552...|[0.01987969039816...|       1.0|
|[11.31,19.04,71.8...|  1.0|[-3.5151402296256...|[0.02888450174100...|       1.0|
|[11.43,17.31,73.6...|  1.0|[-4.9060805032983...|[0.00734706262482...|       1.0|
|[11.45,20.97,73.8...|  1.0|[-4.2193435703871...|[0.01449509810021...|       1.0|
|[11.64,18.33,75.1...|  1.0|[-1.6885192483108...|[0.15597067264784...|       1.0|
|[11.75,20.18,76.1...|  1.0|[-2.4644602027662...|[0.07838751261233...|       1.0|
+--------------------+-----+--------------------+--------------------+----------+
only showing top 20 rows
```

Figure 6: Prediction with prediction precision

Figure 7 shows the prediction Dataset for the test set. The print method shown essentially generates output, much like the following:

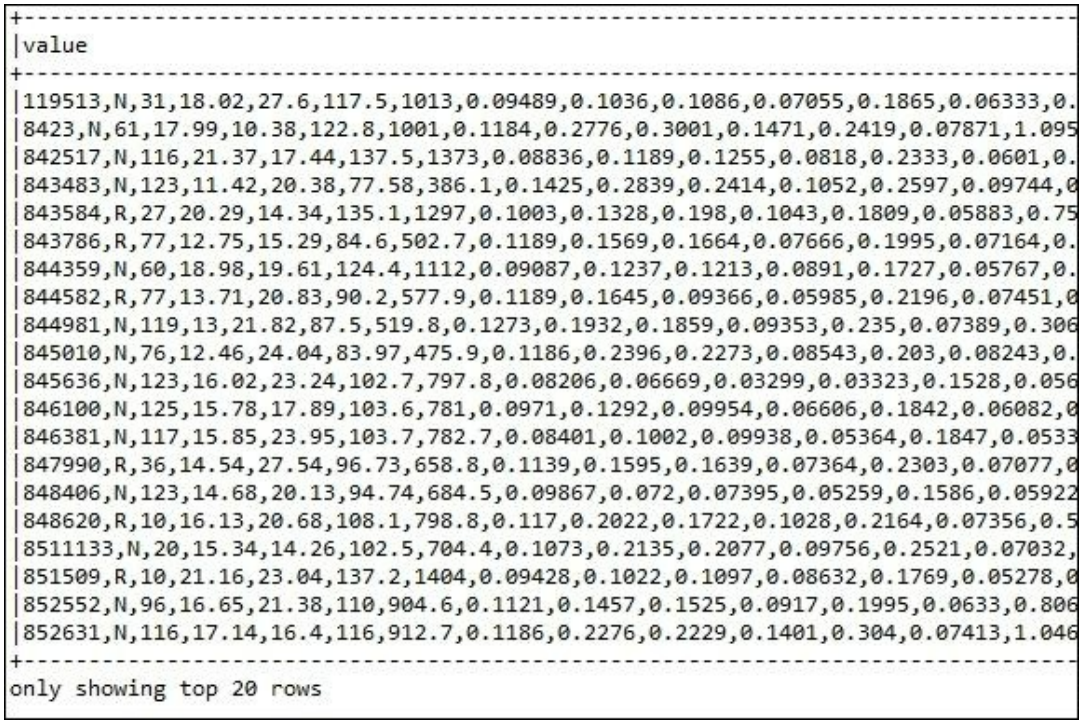


Figure 7: Sample output toward the prediction. The first value is the feature, the second is the label, and the final value is the prediction value

Now let's calculate the precision score. We do this by multiplying the counter by 100 and then dividing the value against how many predictions were done, as follows:

```
System.out.println("precision: " + (double)
(count * 100) / predictions.count());
Precision - 100.0
```

Therefore, the precision is 100%, which is fantastic. However, if you are still unsatisfied or have any confusion, the following chapter will demonstrate how you can still tune several parameters so that the prediction accuracy increases, as there might have been many false-negative predictions.

Furthermore, the result might vary on your platform due to the random-split nature and dataset processing on your side.

Cancer-prognosis pipeline with Spark

In the previous section, we showed how to develop a cancer diagnosis pipeline for predicting cancer based on two labels (Benign and Malignant). In this section, we will look at how to develop a cancer prognosis pipeline with Spark ML and MLlib APIs. The **Wisconsin Prognosis Breast Cancer** (**WPBC**) datasets will be used to predict the probability of breast cancer toward the prognosis for recurrent and non-recurrent tumor cells. Again, the dataset was downloaded from https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Prognostic). To understand the problem formalization, please refer to *Figure 1* once again as we will follow almost the same stages during the cancer-prognosis pipeline development.

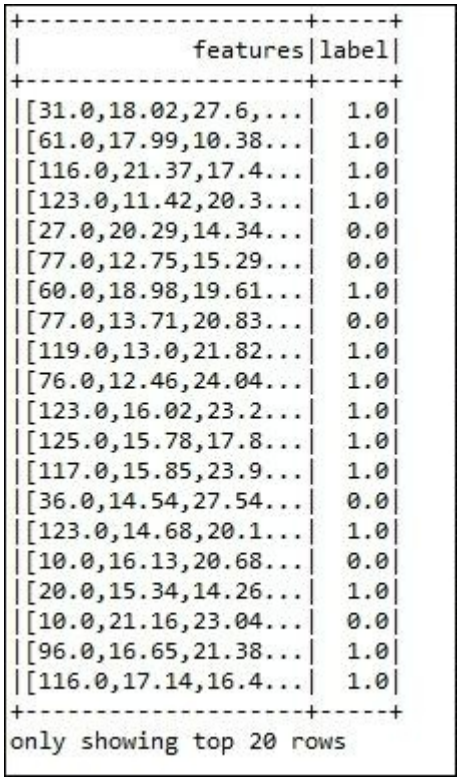
Dataset exploration

The details of the attributes found in the WPBC dataset in https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/wpbc.names are as follows:

- ID number
- Outcome (R = recurrent, N = non-recurrent)
- Time (recurrence time if field 2 => R, disease-free time if field 2 => N)
- 3 to 33: Ten real-valued features are computed for each cell nucleus: Radius, Texture, Perimeter, Area, Smoothness, Compactness, Concavity, Concave points, Symmetry, and Fractal dimension. Thirty-four is Tumor size and thirty-five is the Lymph node status, as follows:
    - Tumor size: Diameter of the excised tumor in centimeters
    - Lymph node status: The number of positive axillary lymph nodes

If you compare *Figure 3* and *Figure 9*, you will see that the diagnosis and prognosis have the same features, yet the prognosis has two additional features (mentioned previously as 34 and 35). Note that these are observed at the time of surgery from the year 1988 to 1995 and out of the 198 instances, 151 are non-recurring (N) and 47 are recurring (R), as shown in *Figure 8*.

Of course, a real cancer diagnosis and prognosis dataset today contains many other features and fields in a structured or unstructured way:



```
+--------------------+-----+
|            features|label|
+--------------------+-----+
|[31.0,18.02,27.6,...|  1.0|
|[61.0,17.99,10.38...|  1.0|
|[116.0,21.37,17.4...|  1.0|
|[123.0,11.42,20.3...|  1.0|
|[27.0,20.29,14.34...|  0.0|
|[77.0,12.75,15.29...|  0.0|
|[60.0,18.98,19.61...|  1.0|
|[77.0,13.71,20.83...|  0.0|
|[119.0,13.0,21.82...|  1.0|
|[76.0,12.46,24.04...|  1.0|
|[123.0,16.02,23.2...|  1.0|
|[125.0,15.78,17.8...|  1.0|
|[117.0,15.85,23.9...|  1.0|
|[36.0,14.54,27.54...|  0.0|
|[123.0,14.68,20.1...|  1.0|
|[10.0,16.13,20.68...|  0.0|
|[20.0,15.34,14.26...|  1.0|
|[10.0,21.16,23.04...|  0.0|
|[96.0,16.65,21.38...|  1.0|
|[116.0,17.14,16.4...|  1.0|
+--------------------+-----+
only showing top 20 rows
```

Figure 8: Snapshot of the data (partial)

Tip

For more detailed discussion and meaningful insights, interested readers can refer to the following research paper: *The Wisconsin Breast Cancer Problem: Diagnosis and DFS time prognosis using probabilistic and generalized regression neural classifiers Oncology Reports, special issue Computational Analysis and Decision Support Systems in Oncology, last quarter 2005 by Ioannis A. et al. found in the following link:* http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.65.2463&rep=rep1&type=pdf.

Breast-cancer-prognosis pipeline with Spark ML/MLlib

In this subsection, we will look at how to develop a breast cancer prognosis machine learning pipeline step-by-step, including taking the input of the dataset to prediction in 10 different steps that are described in *Figure 1*, as a data workflow.

Tip

Readers are advised to download the dataset and the project files, along with the `pom.xml` file for the Maven project configuration, from the Packt materials. We have advised how to make the code work in previous chapters, for example, Chapter 1, *Introduction to Data Analytics with Spark*.

Step 1: Import necessary packages/libraries/APIs

```
import org.apache.spark.api.java.JavaRDD;
import
org.apache.spark.api.java.function.Function;
import org.apache.spark.ml.Pipeline;
import org.apache.spark.ml.PipelineModel;
import org.apache.spark.ml.PipelineStage;
import
org.apache.spark.ml.classification.LogisticRegression;
import org.apache.spark.ml.feature.LabeledPoint;
import org.apache.spark.ml.linalg.DenseVector;
import org.apache.spark.ml.linalg.Vector;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;
```

Step 2: Initialize the necessary Spark environment

```
static SparkSession spark = SparkSession
        .builder()

 .appName("BreastCancerDetectionPrognosis")
       .master("local[*]")
       .config("spark.sql.warehouse.dir",
"E:/Exp/")
       .getOrCreate();
```

Here we set the application name as `BreastCancerDetectionPrognosis`, the master URL as `local[*]`. The Spark Context is the entry point of the program. Please set these parameters accordingly.

Step 3: Take the breast cancer data as input and prepare JavaRDD out of the data

```
String path = "input/wpbc.data";
JavaRDD<String> lines =
spark.sparkContext().textFile(path,
3).toJavaRDD();
```

Tip

To learn more about the data, please refer to *Figure 5* and its description and the Dataset exploration subsection.

Step 4: Create LabeledPoint RDDs

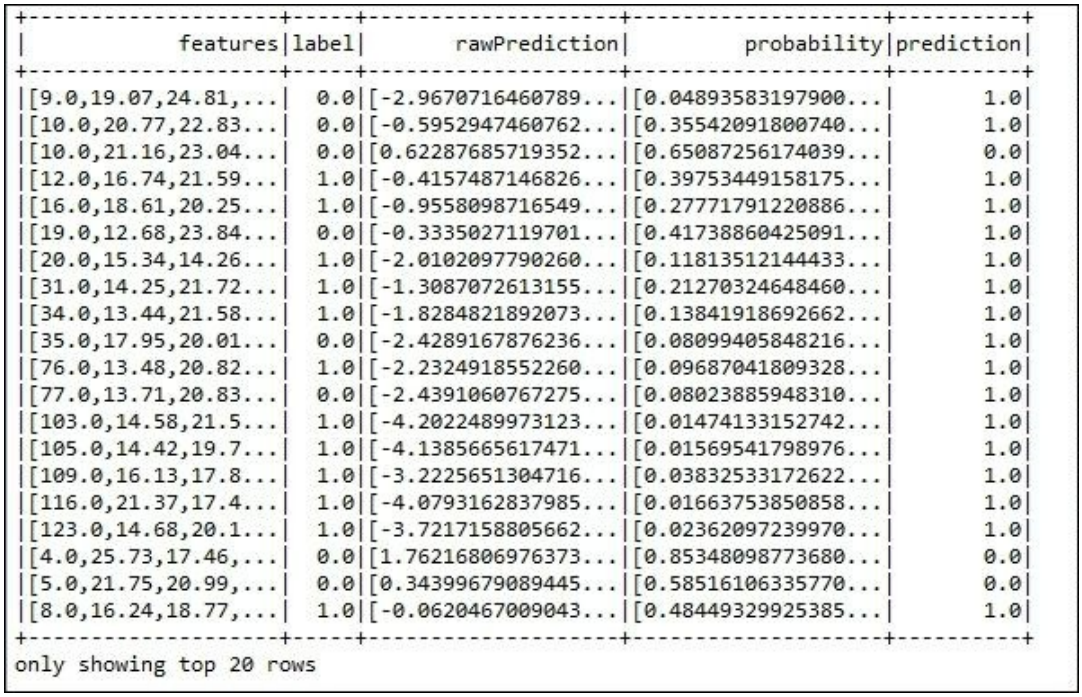
Create `LabeledPoint` RDDs for the prognosis for N = recurrent and R= non-recurrent, respectively, using the following code segments:

```
JavaRDD<LabeledPoint> linesRDD = lines.map(new
Function<String, LabeledPoint>() {
      public LabeledPoint call(String lines) {
        String[] tokens = lines.split(",");
        double[] features = new double[30];
        for (int i = 2; i < features.length; i++)
{
          features[i - 2] =
Double.parseDouble(tokens[i]);
        }
        Vector v = new DenseVector(features);
        if (tokens[1].equals("N")) {
          return new LabeledPoint(1.0, v); //
recurrent
        } else {
          return new LabeledPoint(0.0, v); //
non-recurrent
        }
      }
    });
```

Step 5: Create the Dataset from the lines RDD and show the top features

```
Dataset<Row> data =
spark.createDataFrame(linesRDD,LabeledPoint.class);
data.show();
```

The top features and their corresponding labels are shown in *Figure 9*:



features	label	rawPrediction	probability	prediction
[9.0,19.07,24.81,...	0.0	[-2.9670716460789...	[0.04893583197900...	1.0
[10.0,20.77,22.83...	0.0	[-0.5952947460762...	[0.35542091800740...	1.0
[10.0,21.16,23.04...	0.0	[0.62287685719352...	[0.65087256174039...	0.0
[12.0,16.74,21.59...	1.0	[-0.4157487146826...	[0.39753449158175...	1.0
[16.0,18.61,20.25...	1.0	[-0.9558098716549...	[0.27771791220886...	1.0
[19.0,12.68,23.84...	0.0	[-0.3335027119701...	[0.41738860425091...	1.0
[20.0,15.34,14.26...	1.0	[-2.0102097790260...	[0.11813512144433...	1.0
[31.0,14.25,21.72...	1.0	[-1.3087072613155...	[0.21270324648460...	1.0
[34.0,13.44,21.58...	1.0	[-1.8284821892073...	[0.13841918692662...	1.0
[35.0,17.95,20.01...	0.0	[-2.4289167876236...	[0.08099405848216...	1.0
[76.0,13.48,20.82...	1.0	[-2.2324918552260...	[0.09687041809328...	1.0
[77.0,13.71,20.83...	0.0	[-2.4391060767275...	[0.08023885948310...	1.0
[103.0,14.58,21.5...	1.0	[-4.2022489973123...	[0.01474133152742...	1.0
[105.0,14.42,19.7...	1.0	[-4.1385665617471...	[0.01569541798976...	1.0
[109.0,16.13,17.8...	1.0	[-3.2225651304716...	[0.03832533172622...	1.0
[116.0,21.37,17.4...	1.0	[-4.0793162837985...	[0.01663753850858...	1.0
[123.0,14.68,20.1...	1.0	[-3.7217158805662...	[0.02362097239970...	1.0
[4.0,25.73,17.46,...	0.0	[1.76216806976373...	[0.85348098773680...	0.0
[5.0,21.75,20.99,...	0.0	[0.34399679089445...	[0.58516106335770...	0.0
[8.0,16.24,18.77,...	1.0	[-0.0620467009043...	[0.48449329925385...	1.0

only showing top 20 rows

Figure 9: Top features and their corresponding labels

Step 6: Split the Dataset to prepare the training and test sets

Here we split the dataset to test and the training set as 60% and 40%, respectively. Please adjust these based on your requirements:

```
Dataset<Row>[] splits = data.randomSplit(new
double[] { 0.6, 0.4 }, 12345L);
Dataset<Row> trainingData = splits[0];
Dataset<Row> testData = splits[1];
```

To see a quick snapshot of these two sets, just write `trainingData.show()` and `testData.show()`, for training and test sets, respectively.

Step 7: Create a Logistic Regression classifier

Create a logistic regression classifier by specifying the max iteration and regression parameter:

```
LogisticRegression logisticRegression = new
LogisticRegression()
.setMaxIter(100)
.setRegParam(0.01)
.setElasticNetParam(0.4);
```

Step 8: Create a pipeline and train the pipeline model

```
Pipeline pipeline = new Pipeline().setStages(new
PipelineStage[]{logisticRegression});
```

```
PipelineModel model=pipeline.fit(trainingData);
```

Here, similarly to the diagnosis pipeline, we have created the prognosis pipeline whose stages are defined by only the logistic regression, which is again an estimator, and of course a stage.

Step 9: Create a Dataset and transform the model

Create a Dataset and do the transformation to make a prediction based on the test dataset:

```
Dataset<Row>
predictions=model.transform(testData);
```

Step 10: Show the prediction with prediction precision

```
predictions.show();
```

```
+--------------------+-----+--------------------+--------------------+----------+
|            features|label|       rawPrediction|         probability|prediction|
+--------------------+-----+--------------------+--------------------+----------+
|[7.0,20.29,21.49,...|  0.0|[-0.9140691613904...|[0.28616788319455...|       1.0|
|[8.0,18.65,17.6,1...|  0.0|[-1.2528495673355...|[0.22220725488150...|       1.0|
|[9.0,19.07,24.81,...|  0.0|[-2.6537079225322...|[0.06576084113577...|       1.0|
|[10.0,12.77,22.47...|  1.0|[-1.1562894148114...|[0.23934217148709...|       1.0|
|[10.0,21.16,23.04...|  0.0|[1.04918181571503...|[0.74061775439349...|       0.0|
|[12.0,16.74,21.59...|  1.0|[-1.2292783929854...|[0.22630774878993...|       1.0|
|[19.0,12.68,23.84...|  0.0|[-0.7655901021761...|[0.31743382683677...|       1.0|
|[27.0,20.29,14.34...|  0.0|[-1.7044878065449...|[0.15388004024379...|       1.0|
|[31.0,14.25,21.72...|  1.0|[-2.3454000247467...|[0.08743209814738...|       1.0|
|[34.0,13.44,21.58...|  1.0|[-1.3829415304139...|[0.20053699254342...|       1.0|
|[34.0,14.25,22.15...|  0.0|[-2.1600540691290...|[0.10339543888177...|       1.0|
|[35.0,17.95,20.01...|  0.0|[-1.8884324543727...|[0.13142330332236...|       1.0|
|[36.0,14.54,27.54...|  0.0|[-1.7908302248192...|[0.14297096565331...|       1.0|
|[53.0,19.27,26.47...|  1.0|[-0.8145738755736...|[0.30691668790662...|       1.0|
|[62.0,14.71,21.59...|  1.0|[-1.8617174922979...|[0.13450298981185...|       1.0|
|[105.0,14.42,19.7...|  1.0|[-3.1830623918478...|[0.03980811373365...|       1.0|
|[106.0,14.78,23.9...|  1.0|[-2.4966496618951...|[0.07609338518495...|       1.0|
|[108.0,18.66,17.1...|  1.0|[-2.2288663231462...|[0.09718806726643...|       1.0|
|[109.0,16.13,17.8...|  1.0|[-2.2221061420877...|[0.09778284011957...|       1.0|
|[111.0,13.17,18.6...|  1.0|[-2.8396209469675...|[0.05522031003533...|       1.0|
+--------------------+-----+--------------------+--------------------+----------+
only showing top 20 rows
```

Figure 10: Prediction with prediction precision

```
long count = 0;
for (Row r : predictions.select("features",
"label", "prediction").collectAsList()) {
      System.out.println("(" + r.get(0) + ", " +
r.get(1) + r.get(2) + ", prediction=" +
r.get(2));
      count++;
    }
```

This code segment will produce an output similar to that shown in *Figure 7*, with different features, labels, and predictions:

```
System.out.println("precision: " + (double)
(count * 100) / predictions.count());
Precision: 100.0
```

Therefore, the precision is almost 100%, which is fantastic. However, depending upon the data preparation, you might receive different results.

If you have any confusion, the following chapter demonstrates how to tune parameters so that the prediction accuracy increases, as they might have many false-negative predictions.

> **Tip**
>
> In their book titled *Machine Learning with R, Packt Publishing, 2015*, Brett Lantz at el. argue that it's possible to eliminate the false negatives completely by classifying every mass as malignant, benign, recurrent, or non-recurrent. Obviously, this is not a realistic strategy. Still, it illustrates the fact that prediction involves striking a balance between the false-positive rate and the false-negative rate.

If you are still unsatisfied, we will be tuning several parameters in Chapter 7, *Tuning Machine Learning Models*, so that the prediction accuracy increases toward more sophisticated methods for measuring predictive accuracy that can be used to identify places where the error rate can be optimized depending on the costs of each type of error.

Market basket analysis with Spark Core

In this section, we will look at how to develop a large-scale machine learning pipeline in terms of market basket analysis. Other than using the Spark ML and MLlib, we will demonstrate how to use Spark Core to develop such an application.

Background

In an early paper, *An Efficient Market Basket Analysis Technique with Improved MapReduce Framework on Hadoop: An E-commerce Perspective* (available at http://onlinepresent.org/proceedings/vol6_2012/8.pdf), the authors have argued that the **market basket analysis** (**MBA**) technique is of substantial importance to everyday business decision, since customers' purchase rules can be extracted from the association rules by discovering what items they are buying frequently and together. Consequently, purchase rules can be revealed for frequent shoppers based on these association rules.

You might still be wondering why we need market basket analysis, why it is important, and why it is computationally expensive. Well, if you could identify highly specific association rules like, for example, if a customer prefers mango or orange jam along with their milk or butter, you need to have large-scale transactional data to be analyzed and processed. Moreover, some massive chain retailers or supermarkets, for example, E-mart (UK), HomePlus (Korea), Aldi (Germany), or Dunnes Stores (Ireland) use databases of many millions, or even billions, of transactions in order to find the associations among particular items with regard to brand, color, origin, or even flavor, to increase the probability of sales and profit.

In this section, we will look at an efficient approach for large-scale market basket analysis with Spark libraries. After reading and practising this, you will be able to show how the Spark framework lifts the existing single-node pipeline to a pipeline usable on a multi-node data-mining cluster. The result is that our proposed association-rules mining algorithm can be reused in parallel with the same benefits.

We use the acronym SAMBA, for Spark-based Market Basket Analysis, *min_sup* for minimum support, and *min_conf* for minimum confidence. We also use the terms frequent patterns and frequent itemset interchangeably.

Motivations

Traditional main memory or disk-based computing and RDBMS are not capable of handling ever-increasing large transactional data. Furthermore, as discussed in Chapter 1, *Introduction to Data Analytics with Spark*, MapReduce has several issues with the I/O operation, algorithmic complexity, low-latency, and fully disk-based operation. Therefore, finding the null transactions and later eliminating them from the future scheme is the initial part of this approach.

It is quite possible to find all the null transactions by identifying those transactions that do not appear against at least one frequent 1-itemset. As already mentioned, Spark caches the intermediate data into memory and provides an abstraction of **Resilient Distributed Datasets** (**RDDs**), which can be used to overcome these issues by making a huge difference, achieving tremendous success in the last three years for handling large-scale data in distributed computing systems. These successes are promising and motivating examples to explore this research work to applying Spark in market basket analysis.

Exploring the dataset

Please download the grocery dataset for the market basket analysis from https://github.com/stedy/Machine-Learning-with-R-datasets/blob/master/groceries.csv. The first five rows of the raw `grocery.csv` data are as follows in *Figure 11*. These lines indicate 10 separate grocery-store transactions. The first transaction includes four items: citrus fruit, semi-finished bread, margarine, and ready soups. In comparison, the third transaction includes only one item, whole milk:

```
+--------------------------------------------------------------------------------+
|value                                                                           |
+--------------------------------------------------------------------------------+
|citrus fruit,semi-finished bread,margarine,ready soups                          |
|tropical fruit,yogurt,coffee                                                    |
|whole milk                                                                      |
|pip fruit,yogurt,cream cheese ,meat spreads                                     |
|other vegetables,whole milk,condensed milk,long life bakery product             |
|whole milk,butter,yogurt,rice,abrasive cleaner                                  |
|rolls/buns                                                                      |
|other vegetables,UHT-milk,rolls/buns,bottled beer,liquor (appetizer)            |
|pot plants                                                                      |
|whole milk,cereals                                                              |
|tropical fruit,other vegetables,white bread,bottled water,chocolate             |
|citrus fruit,tropical fruit,whole milk,butter,curd,yogurt,flour,bottled water,dishes|
|beef                                                                            |
+--------------------------------------------------------------------------------+
```

Figure 11: A snapshot of the groceries dataset

Problem statements

We believe we have enough motivations and reasons for why we need to analyze the market basket using transactional or retail datasets. Now, let us

discuss some background studies, which are needed to apply our Spark-based market basket analysis technique.

Suppose you have a set of distinct items *I = {i1, i2...in}* and *n* is the number of distinct items. A transactional database *T = {t1, t2...tN}* is a set of *N* transactions and |*N*| is the number of total transactions. A set *X* $X \subseteq I$ is called a pattern or itemset. We assume that input is given as a sequence of transactions, where items are separated by a comma, as shown in *Table 1*.

For the sake of simplicity to describe the background study, the same transactions are presented with a single character in *Table 2*:

Transaction 1	crackers, ice-cream, coke, orange,
Transaction 2	beef, pizza, coke, bread
Transaction 3	baguette, soda, shampoo, crackers, pepsi
Transaction 4	burger, cream cheese, diapers, milk
...	...

Table 1. Sample transactions made by a customer

TID	**Itemset (Sequence of items)**
10	A, B, C, F
20	C, D, E
30	A, C, E, D
40	A
50	D, E, G

60	B, D
70	B
80	A, E, C
90	A, C, D
100	B, E, D

Table 2. A transactional database

If $X \subseteq t$, it is said that *X* occurs in *t* or *t* contains *X*. The support count is the frequency of occurrence of an itemset in all transactions, which can be described as follows:

$$Support(X) = \frac{Count(X)}{N} \ldots\ldots (i)$$

In other words, if the *support* $(X) \geq min_sup$, , we say that *X* is a frequent itemset. For example, in *Table 2*, the occurrences of itemsets *CD*, *DE,* and *CDE* are *3*, *3*, and *2*, respectively, and if the *min_sup* is *2*, all of these are frequent itemsets.

On the other hand, association rules are statements of form $X \rightarrow Y$, or more formally:

$$X \rightarrow Y, where\ X, Y \in I, and\ X \cap Y = \phi \ldots\ldots (ii)$$

Therefore, we can say that an association rule is a pattern that states when *X* occurs, then *Y* occurs with a certain probability. Confidence for the association rule defined in equation 1 can be expressed as how often items in *Y* appear in transactions that also contain *X*, as follows:

$$Confidence(X \rightarrow Y) = \frac{support(X,Y)}{support(X)} \ldots\ldots (iii)$$

Now we need to introduce a new parameter, called `lift`, which as a metric is a measure of how much more likely one item is to be purchased relative to its typical purchase rate, given that you know another item has been purchased. This is defined by the following equation:

$$lift(X \rightarrow Y) = \frac{confidence(X \rightarrow Y)}{support(Y)} \ldots\ldots (iv)$$

In a nutshell, given a transactional database, now the problem of market basket analysis is to find the complete set of a customer's purchase rules by means of association rules from the frequent itemsets whose support and confidence are no less than the *min_sup* and *min_conf* threshold, respectively.

Large-scale market basket analysis using Spark

As shown in *Figure 12*, we assume that transactional databases are stored in a distributed way in a cluster of DB servers. A DB server is a computing node with large storage and main memory. Therefore, it can store large datasets, so it can compute any task assigned to it. The Driver PC is also a computing node, which mainly works as a client and controls the overall process.

Obviously, it needs to have a large memory for processing and holding the Spark codes to send across the computing nodes. The codes consist of a DB server ID, minimum support, minimum confidence, and mining algorithm:

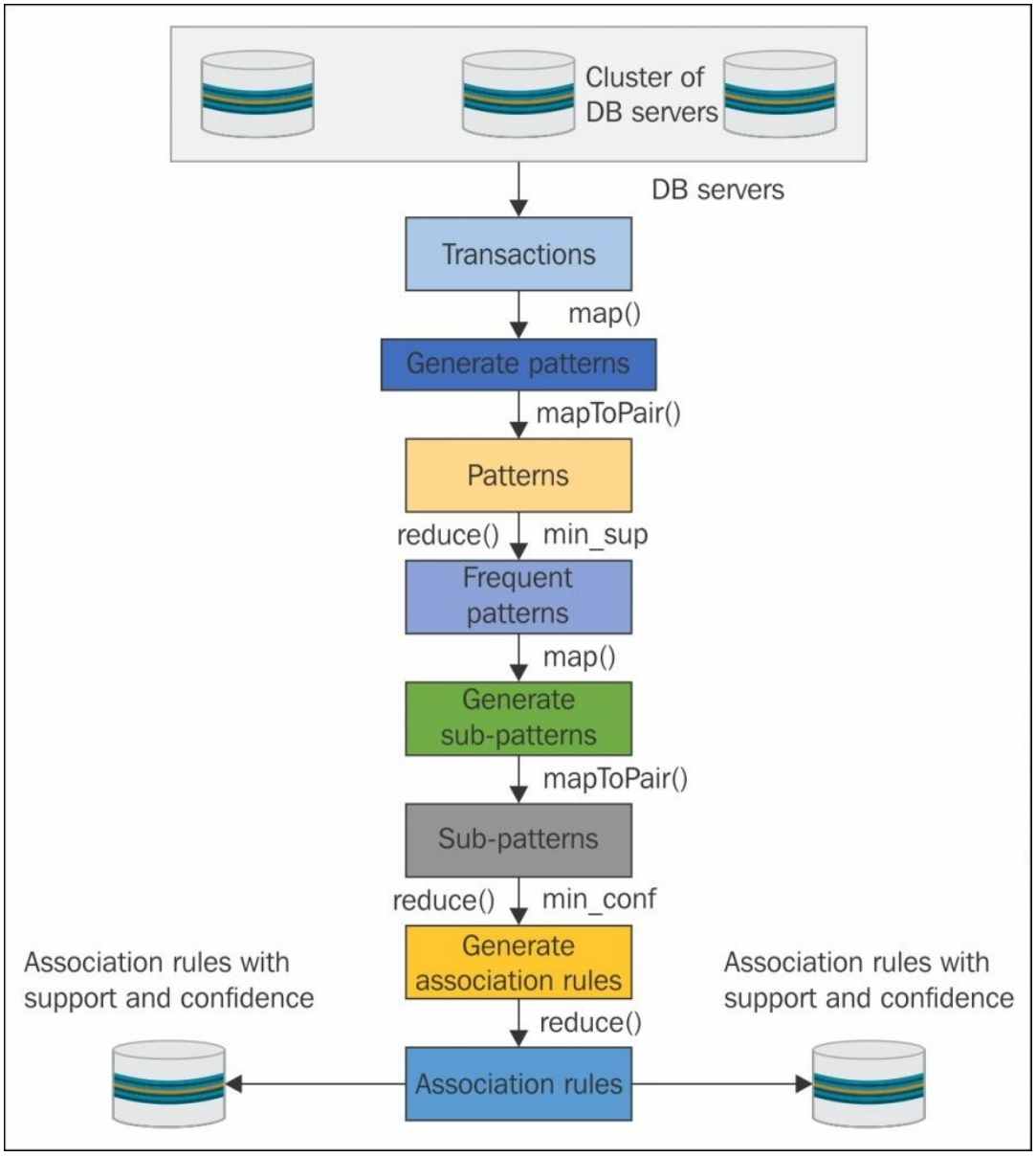


Figure 12: Workflow of the SAMBA algorithm using Spark

From the patterns, frequent patterns are generated using reduce phase 1, which satisfies the constraints *min_sup*. The map phase is applied on the computed frequent patterns to generate the sub-patterns that eventually help to generate the association rules. From the sub-patterns, reduce phase 2 is applied to generate the association rules that satisfy the constraints *min_conf*.

The incorporation of two Map and Reduce phases is possible because of the Spark ecosystem toward the Spark core and associated APIs. The final results are the complete set of association rules with their respective support count and confidence.

These store keepers have the full form to place their items, based on the association between items, to increase sales to frequent and non-frequent

shoppers. Due to space constraints, we cannot show a step-by-step example for the sample transactional database presented in *Table 2*.

However, we believe that the workflow and the pseudo codes will suffice to understand the total scenario. A DB server takes the input of codes sent from the Driver PC and starts the computation. From an environment variable Spark session, we create some initial data reference or RDD objects. Then, the initial RDD objects are transformed to create more and brand new RDD objects in the DB server. At first, it reads the dataset as a plain text (or other supported format) and null transactions using narrow/wide transformations (that is, `flatMap`, `mapToPair`, and `reduceByKey`).

Thereby, the filter join RDD operation provides a data segment without null transactions. Then the RDD objects are materialized to dump the RDD into the DB server's storage as filtered datasets. Spark's inter RDD join operation allows for the combining of the contents of multiple RDDs within a single data node. In summary, we follow the steps given here before getting the filtered dataset:

1. Set the system property of the distributed processing model and cluster manager (that is, Mesos) as true. This value can be saved on your application development as standard Spark code.
2. Set SparkConf, AppName, Master URL, Spark local IP, Spark driver host IP, Spark executor memory, and Spark driver memory.
3. Create `JavaSparkContext` using the `SparkConf`.
4. Create `JavaRDD` and read the dataset as plain text, as transactions, and perform necessary partitioning.
5. Perform a `flatMap` operation over the RDD to split the transactions as items.
6. Perform the `mapToPair` operation to ease finding the key/value pairs of the items.
7. Perform the filter operation to remove all the null transactions.

When we have the filtered databases, we materialize an action inter-RDD join operation to save the dataset on a DB server or partition if it does not have enough storage for a single machine, or cache if there's not enough memory.

Figure 12 shows the complete workflow of getting association rules as the final results using Spark's APIs. On the other hand, Figure 13 shows the pseudo-code of the algorithm, namely, **Spark-Based Market Basket Analysis** (**SAMBA**). There are actually two Map and Reduce operations associated, as outlined here:

- **Map/Reduce phase 1**: Mappers read the transactions from the HDFS servers and convert the transactions to patterns. Reducers, on the other hand, find the frequent patterns.

- **Map/Reduce phase 2**: Mappers convert the frequent patterns into sub-patterns. On the other hand, a reducer generates the association rules based on the given constraints (`min_conf` and `lift`):

Algorithm 1: SAMBA Algorithm

```
Input: 1st input TDB, min_sup; 2nd input FTDB (filtered TDB) and min_conf
Output: The complete set of association rules with their confidence
Entry point: Create Spark context object;
Procedure filterTDB (TDB)  //removes null transactions
    for (i = 1; i<TDB.length; i++) do
        Read all the transactions on HDFS or HDD;
        Create JavaRDD from the transactions;
        Find all transactions that don't appear against at least one frequent 1-itemset;
    end
    Return filteredTransactions;
EndProcedure
Procedure generateFrequentPatterns (FTDB, min_sup) //generates frequent patterns
    for (i = 1; i<FTDB.length; i++) do
        Read all the transactions from the filtered TDB on HDFS or HDD;
        Create new JavaRDD from the filtered transactions;
        Generate all combination of patterns from each transaction from RDDs; //Map 1
        Generate patterns X from the RDDs; //Reduce 1
        if (Sup(X) >= min_sup) then
            freqList.add(X);
        end
    end
    Return freqList;
EndProcedure
Procedure generateSubPatterns (freqList) //generates sub-patterns for rules computaiton
    for (i = 1; i< freqList.length; i++) do
        Read all the frequent patterns on HDFS or HDD;
        Generate all combination of sub-patterns; //Map 2
    end
    Return subPatternList;
EndProcedure
Procedure generateAssociationRules (subPatternList, min_conf) //generates ass. rules
    for (i = 1; i< subPatternList.length; i++) do
        Read all the sub-patterns on HDFS or HDD;
        Combine and generate all sub-patterns of the form X=>Y //Reduce 2
        if (Conf(X =>) >= min_conf) then
            freqList.add(X);
        end
    end
    Return associationRulesLitst;
EndProcedure
```

Figure 13: The SAMBA algorithm

After that, the SAMBA algorithm reads the **filtered database** (**FTDB**) and applies map phase 1 to generate all the possible combinations of the patterns. Then the `mapToPair()` methods them as patterns with their respective supports.

The algorithm solution using Spark Core

Here we will look at how to do the market basket analysis using Spark Core. Please note, we will not use the Spark ML or MLlib since although MLlib provides a technique of calculating the association rules, however, it does not show how to calculate some other parameters such as calculating confidence, support, and lift that are very needed for a complete analysis of groceries dataset. Therefore, we will show a complete example, step-by-step, from data exploration to association rules generation.

Step 1: Import the necessary packages and APIs

Here is the code to import packages and APIs:

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import
org.apache.spark.api.java.function.Function;
import
org.apache.spark.api.java.function.Function2;
import
org.apache.spark.api.java.function.PairFlatMapFunction;
import org.apache.spark.rdd.RDD;
import org.apache.spark.sql.SparkSession;
import scala.Tuple2;
import scala.Tuple4;
```

Step 2: Create the entry point by specifying the Spark session

The entry point can be created with the help of the following code:

```
SparkSession spark = SparkSession
.builder()
.appName("MarketBasketAnalysis")
.master("local[*]")
.config("spark.sql.warehouse.dir", "E:/Exp/")
.getOrCreate();
```

Step 3: Create the Java RDD for the transactions

Java RDD for the transactions can be created with the help of the following code:

```
String transactionsFileName =
"Input/groceries.data";
```

```
RDD<String> transactions =
spark.sparkContext().textFile(transactionsFileName,
1);
transactions.saveAsTextFile("output/transactions");
```

Step 4: Create a method for creating the list

Create a method named `toList` from the created transactions RDDs, which will add all the items in the transactions:

```
  static List<String> toList(String transaction)
{
    String[] items =
transaction.trim().split(",");
    List<String>list = new ArrayList<String>();
    for (String item :items) {
      list.add(item);
    }
    returnlist;
  }
```

Step 5: Remove infrequent items and null transactions

Create a method named `removeOneItemAndNullTransactions` to remove infrequent items and null transactions:

```
static List<String>
removeOneItemAndNullTransactions(List<String>list,
int i) {
    if ((list == null) || (list.isEmpty())) {
      returnlist;
    }
    if ((i< 0) || (i> (list.size() - 1))) {
      returnlist;
    }
    List<String>cloned = new
ArrayList<String>(list);
    cloned.remove(i);
    return cloned;
  }
```

Step 6: Flat mapping and 1-itemsets creation (map phase 1)

Do the `flatmap` and create 1-itemsets. Finally, save the patterns:

```
JavaPairRDD<List<String>, Integer> patterns =
transactions.toJavaRDD()
        .flatMapToPair(new
PairFlatMapFunction<String, List<String>,
Integer>() {
          @Override
```

```
  public Iterator<Tuple2<List<String>, Integer>>
call(String transaction) {
  List<String> list = toList(transaction);
  List<List<String>> combinations =
Combination.findSortedCombinations(list);
  List<Tuple2<List<String>, Integer>> result =
new ArrayList<Tuple2<List<String>, Integer>>();
for (List<String> combList : combinations) {
  if (combList.size() > 0) {
  result.add(new Tuple2<List<String>,
Integer>(combList, 1));
              }
            }
    return result.iterator();
          }
        });
    patterns.saveAsTextFile("output/1itemsets");
```

Note

Note that the last saving of the patterns RDD is for optional reference purposes and so that you can see the contents of the RDDs.

The following is a screenshot of the 1-itemsets:

```
1([citrus fruit],1)
2([semi-finished bread],1)
3([margarine],1)
4([ready soups],1)
5([citrus fruit, semi-finished bread],1)
6([citrus fruit, margarine],1)
7([citrus fruit, ready soups],1)
8([margarine, semi-finished bread],1)
9([ready soups, semi-finished bread],1)
0([margarine, ready soups],1)
1([citrus fruit, margarine, semi-finished bread],1)
2([citrus fruit, ready soups, semi-finished bread],1)
3([citrus fruit, margarine, ready soups],1)
4([margarine, ready soups, semi-finished bread],1)
5([citrus fruit, margarine, ready soups, semi-finished bread],1)
6([tropical fruit],1)
7([yogurt],1)
```

Figure 14: 1-itemsets

Step 7: Combine and reduce frequent patterns (reduce phase 1)

Combine and reduce all the frequent patterns, and save them:

```
JavaPairRDD<List<String>, Integer> combined =
patterns.reduceByKey(new Function2<Integer,
Integer, Integer>() {
      public Integer call(Integer i1, Integer i2)
{
        int support = 0;
```

```
            if (i1 + i2 >= 2) {
              support = i1 + i2;
            }
            // if(support >= 2)
            return support;
          }
        });

    combined.saveAsTextFile("output/frequent_patterns");
```

The following is a snapshot of the frequent patterns with their respective support (frequency in *Figure 15)*:

```
([curd, pasta, rolls/buns, whole milk],1)
([canned vegetables, misc. beverages, shopping bags, white bread],1)
([beef, liquor, whole milk],1)
([butter milk, fruit/vegetable juice, other vegetables, soda],1)
([butter milk, fruit/vegetable juice, other vegetables, pip fruit, soda, white bread],1)
([pastry, pork, rolls/buns, waffles, whole milk],1)
([chocolate, pastry, pork, ready soups, rolls/buns, waffles, whole milk],1)
([salty snack],1)
([fruit/vegetable juice, other vegetables, pip fruit, pork, white bread],1)
([frozen meals, pork, rolls/buns, shopping bags, whole milk],1)
([beef, brown bread, liquor],1)
([canned vegetables, shopping bags, sliced cheese],1)
([whole milk, yogurt],2)
([liquor, newspapers, other vegetables, yogurt],1)
([brown bread, frozen meals, ice cream, tropical fruit, yogurt],1)
([pet care, rolls/buns, white bread],1)
([chicken, margarine],1)
([other vegetables, pip fruit, pork, white bread],1)
([chicken, citrus fruit, onions, pet care, pot plants],1)
([curd, napkins, pastry, rolls/buns, whole milk],1)
([chicken, pet care, yogurt],1)
([brown bread, newspapers, yogurt],1)
```

Figure 15: Frequent patterns with their respective support (frequency)

Step 8: Generate all the candidate frequent patterns (map phase 2)

Generate all the candidate frequent patterns or sub-patterns by removing the 1-itemsets from the frequent patterns, and finally, save the candidate patterns:

```
JavaPairRDD<List<String>, Tuple2<List<String>,
Integer>> candidate-patterns =
combined.flatMapToPair(
new PairFlatMapFunction<Tuple2<List<String>,
Integer>, List<String>, Tuple2<List<String>,
Integer>>() {
          @Override
public Iterator<Tuple2<List<String>,
Tuple2<List<String>, Integer>>> call(
Tuple2<List<String>, Integer> pattern) {
List<Tuple2<List<String>, Tuple2<List<String>,
Integer>>> result = new
ArrayList<Tuple2<List<String>,
Tuple2<List<String>, Integer>>>();
  List<String> list = pattern._1;
  frequency = pattern._2;
```

```
  result.add(new Tuple2(list, new Tuple2(null,
frequency)));
            if (list.size() == 1) {
              return result.iterator();
            }

  // pattern has more than one item
  // result.add(new Tuple2(list, new
Tuple2(null,size)));
    for (int i = 0; i < list.size(); i++) {
    List<String> sublist = removeOneItem(list,
i);
              result.add(new Tuple2<List<String>,
Tuple2<List<String>, Integer>>(sublist,
                  new Tuple2(list, frequency)));
            }
            return result.iterator();
          }
        });
candidate-
patterns.saveAsTextFile("output/sub_patterns");
```

The following is a snapshot of the sub-patterns:

```
([curd, pasta, rolls/buns, whole milk],(null,1))
([pasta, rolls/buns, whole milk],([curd, pasta, rolls/buns, whole milk],1))
([curd, rolls/buns, whole milk],([curd, pasta, rolls/buns, whole milk],1))
([curd, pasta, whole milk],([curd, pasta, rolls/buns, whole milk],1))
([curd, pasta, rolls/buns],([curd, pasta, rolls/buns, whole milk],1))
([canned vegetables, misc. beverages, shopping bags, white bread],(null,1))
([misc. beverages, shopping bags, white bread],([canned vegetables, misc. beverages, shopping bags, white bread],1))
([canned vegetables, shopping bags, white bread],([canned vegetables, misc. beverages, shopping bags, white bread],1))
([canned vegetables, misc. beverages, white bread],([canned vegetables, misc. beverages, shopping bags, white bread],1))
([canned vegetables, misc. beverages, shopping bags],([canned vegetables, misc. beverages, shopping bags, white bread],1))
([beef, liquor, whole milk],(null,1))
([liquor, whole milk],([beef, liquor, whole milk],1))
([beef, whole milk],([beef, liquor, whole milk],1))
([beef, liquor],([beef, liquor, whole milk],1))
```

Figure 16: Sub-patterns for the items

Step 9: Combine all the sub-patterns

Combine all the sub-patterns and save them on disk or persist on memory:

```
JavaPairRDD<List<String>,
Iterable<Tuple2<List<String>, Integer>>>rules =
candidate_patterns.groupByKey();
rules.saveAsTextFile("Output/combined_subpatterns");
```

The following is a screenshot of the candidate patterns (sub-patterns) in combined form:

```
([salty snack],[(null,1), ([finished products, salty snack],1), ([onions, salty snack],1)])
([chicken, margarine],[(null,1), ([chicken, margarine, soda],1), ([chicken, frozen meals, margarine],1)])
([newspapers, oil],[(null,1), ([newspapers, oil, soda],1)])
([chicken, margarine, soda],[(null,1), ([chicken, frozen meals, margarine, soda],1)])
([canned beer],[(null,2), ([canned beer, frozen meals],1), ([candy, canned beer],1)])
([UHT-milk, frankfurter, pork],[(null,1), ([UHT-milk, condensed milk, frankfurter, pork],1)])
([oil, rice, root vegetables, whole milk],[(null,1), ([chicken, oil, rice, root vegetables, whole milk],1)])
([frozen meals, margarine, soda],[(null,1), ([chicken, frozen meals, margarine, soda],1)])
```

Figure 17: Candidate patterns (sub-patterns) in combined form

Step 10: Generate association rules

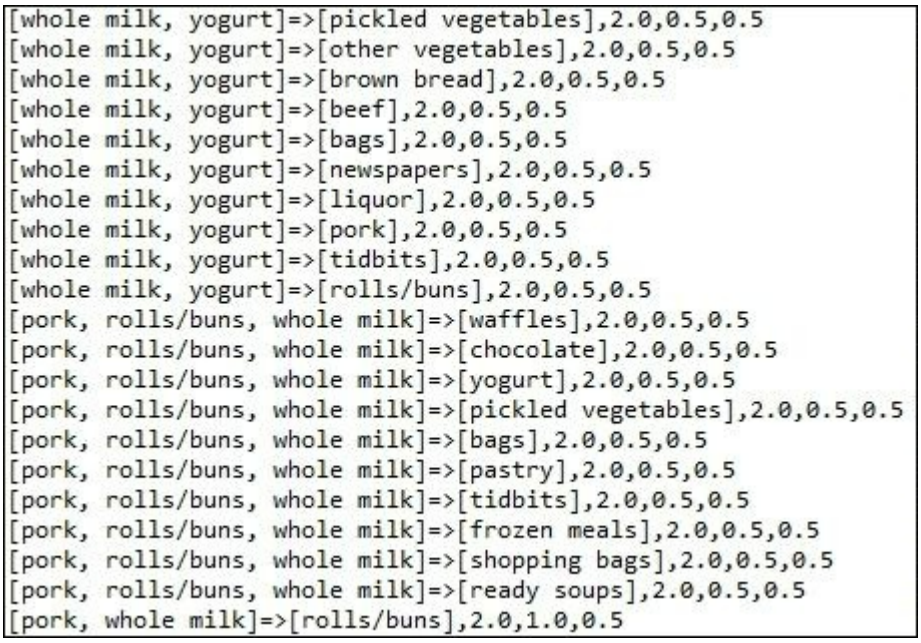
Generate all the association rules from the sub-patterns (reduce phase 2) by specifying the `confidence` and `lift`:

```
JavaRDD<List<Tuple4<List<String>, List<String>,
Double, Double>>> assocRules = rules.map(
        new Function<Tuple2<List<String>,
Iterable<Tuple2<List<String>, Integer>>>,
List<Tuple4<List<String>, List<String>, Double,
Double>>>() {
          @Override
public List<Tuple4<List<String>, List<String>,
Double, Double>> call(
Tuple2<List<String>,
Iterable<Tuple2<List<String>, Integer>>> in)
throws Exception {

List<Tuple4<List<String>, List<String>, Double,
Double>> result = new
ArrayList<Tuple4<List<String>, List<String>,
Double, Double>>();
  List<String> fromList = in._1;
  Iterable<Tuple2<List<String>, Integer>> to =
in._2;
  List<Tuple2<List<String>, Integer>> toList =
new ArrayList<Tuple2<List<String>, Integer>>();
Tuple2<List<String>, Integer> fromCount = null;
      for (Tuple2<List<String>, Integer> t2 : to)
{
        // find the "count" object
      if (t2._1 == null) {
                fromCount = t2;
              } else {
                toList.add(t2);
              }
            }
            if (toList.isEmpty()) {
              return result;
            }
for (Tuple2<List<String>, Integer> t2 : toList) {
  double confidence = (double) t2._2 / (double)
fromCount._2;
double lift = confidence / (double) t2._2;
double support = (double) fromCount._2;
List<String> t2List = new
ArrayList<String>(t2._1);
t2List.removeAll(fromList);
if (support >= 2.0 && fromList != null &&
t2List != null) {
```

```
  result.add(new Tuple4(fromList, t2List,
support, confidence));
System.out.println(fromList + "=>" + t2List + ","
+ support + "," + confidence + "," + lift);
                }
              }
              return result;
            }
          });
assocRules.saveAsTextFile("output/association_rules_with_conf_lift");
```

The following is the output of the association rules including their confidence and lift. For more details on support, confidence, and lift, refer to the problem statement section.

[Antecedent=>Consequent], Support, Confidence, Lift:



```
[whole milk, yogurt]=>[pickled vegetables],2.0,0.5,0.5
[whole milk, yogurt]=>[other vegetables],2.0,0.5,0.5
[whole milk, yogurt]=>[brown bread],2.0,0.5,0.5
[whole milk, yogurt]=>[beef],2.0,0.5,0.5
[whole milk, yogurt]=>[bags],2.0,0.5,0.5
[whole milk, yogurt]=>[newspapers],2.0,0.5,0.5
[whole milk, yogurt]=>[liquor],2.0,0.5,0.5
[whole milk, yogurt]=>[pork],2.0,0.5,0.5
[whole milk, yogurt]=>[tidbits],2.0,0.5,0.5
[whole milk, yogurt]=>[rolls/buns],2.0,0.5,0.5
[pork, rolls/buns, whole milk]=>[waffles],2.0,0.5,0.5
[pork, rolls/buns, whole milk]=>[chocolate],2.0,0.5,0.5
[pork, rolls/buns, whole milk]=>[yogurt],2.0,0.5,0.5
[pork, rolls/buns, whole milk]=>[pickled vegetables],2.0,0.5,0.5
[pork, rolls/buns, whole milk]=>[bags],2.0,0.5,0.5
[pork, rolls/buns, whole milk]=>[pastry],2.0,0.5,0.5
[pork, rolls/buns, whole milk]=>[tidbits],2.0,0.5,0.5
[pork, rolls/buns, whole milk]=>[frozen meals],2.0,0.5,0.5
[pork, rolls/buns, whole milk]=>[shopping bags],2.0,0.5,0.5
[pork, rolls/buns, whole milk]=>[ready soups],2.0,0.5,0.5
[pork, whole milk]=>[rolls/buns],2.0,1.0,0.5
```

Figure 18: Association rules including their confidence and lift

Tuning and setting the correct parameters in SAMBA

Note that if you attempt to use the default parameter settings as support = 0.1 and confidence = 0.6, you might end up with null rules, or technically, no rules, to be generated. You might be wondering why. Actually, the default support of 0.1 means that in order to generate an association rule, an item must have appeared in at least *0.1 * 9385 = 938.5* transactions or 938.5 times (for the dataset we are using, |N| = 9385).

However, in this regard, in their book titled *Machine Learning with R, Packt Publishing, 2015*, Brett Lantz at el. argue that there is one way to tackle this

issue while setting support. They suggest considering the minimum number of transactions needed before you would consider a pattern's interestingness. Moreover, for example, you could also argue that if an item is purchased twice a day (which is approximately 60 times a month), then it may be non-trivial to consider that transaction.

From this perspective, it is possible to estimate how to set the value of support needed to find only rules matching at least that many transactions. Consequently, you can set the value of minimum support as 0.006, because 60 out of 9,835 equals 0.006; we'll try setting the support there first.

On the other hand, setting the minimum confidence also requires a tricky balance and in this regard, again, we would like to refer you to the book by Brett Lantz et al. titled *Machine Learning with R, Packt Publishing, 2015*. If confidence is too low, obviously we might be incredulous with a pretty large of unreliable rules false positive results.

As a result, the optimum value of the minimum confidence threshold depends heavily on the goals of your analysis. Consequently, if you start with conservative values, you can always reduce them to broaden the search if you aren't finding actionable intelligence. If you set the minimum confidence threshold at 0.25, it means that in order to be included in the results, the rule has to be correct at least 25 percent of the time. This will eliminate the most unreliable rules while allowing some room for us to modify behavior with targeted promotions of the product.

Now, let's talk about the third parameter, the `lift`. Before suggesting how to set the value of the `lift`, let's see a practical example of how it might affect the generation of the association rules in the first place. For the third time, we refer to the book by Brett Lantz et al. titled *Machine Learning with R, Packt Publishing, 2015*.

For example, suppose at a supermarket store, many people often purchase milk and bread together. Therefore, naturally, you would expect to find many transactions that contain both milk and bread. However, if `lift` (milk => bread) is greater than 1, this implies that the two items are found together more often than one would expect by chance. As a consequence, a large `lift` value is, therefore, a strong indicator that a rule is important, and reflects a true connection between the items in the transactions.

In summary, we need to set the values of these parameters carefully, by considering the preceding examples. However, as a standalone model the algorithms might take hours to finish. So, run the application with enough time. Alternatively, reduce the long transaction to reduce the time overhead.

OCR pipeline with Spark

Image processing and computer vision are two classical but still-emerging research areas that often make proper utilization of many types of machine learning algorithms. There are several use cases where the relationships of linking the patterns of image pixels to higher concepts are extremely complex and hard to define, and of course, computationally extensive, too.

From a practical point of view, it's relatively easier for a human being to recognize if an object is a face, a dog, or letters or characters. However, defining these patterns under certain circumstances is difficult. Additionally, image-related datasets are often noisy.

In this section, we will develop a model similar to those used at the core of the **Optical Character Recognition** (**OCR**) used as document scanners. This kind of software helps to process paper-based documents by converting printed or handwritten text into an electronic form to be saved in a database.

When OCR software first processes a document, it divides the paper, or any object, into a matrix such that each cell in the grid contains a single glyph (also known as different graphical shapes), which is just an elaborate way of referring to a letter, symbol, or number, or any contextual information from the paper or the object.

To demonstrate the OCR pipeline, we will assume that the document contains only alpha characters in English that match glyphs to one of the 26 letters, A to Z. We will use the OCR letter dataset from the UCI Machine Learning Data Repository (http://archive.ics.uci.edu/ml). The dataset was donated by W. Frey and D. J. Slate et al. To explore the dataset, we have found that the dataset contains 20,000 examples of 26 English alphabet capital letters printed using 20 different randomly reshaped and distorted black-and-white fonts as glyphs of different shapes.

> **Tip**
>
> For more information about these data, refer to *Letter recognition using Holland-style adaptive classifiers, Machine Learning, Vol. 6, pp. 161-182, by W. Frey and D.J. Slate (1991)*.

The image shown in *Figure 19* was published by Frey and Slate and provides an example of some of the printed glyphs. Distorted in this way, the letters are challenging for a computer to identify, yet are easily

recognized by a human being. The statistical attributes for the top 20 rows are shown in *Figure 20*:



Figure 19: Some of the printed glyphs [courtesy of the article titled Letter recognition using Holland-style adaptive classifiers, Machine Learning, Vol. 6, pp. 161-182, by W. Frey and D.J. Slate (1991)]

Exploring and preparing the data

According to the documentation provided by Frey and Slate, when the glyphs are scanned using an OCR reader to the computer they are automatically converted into pixels. Consequently, the 16 statistical attributes mentioned are recorded to the computer too.

Note that the concentration of black pixels across the various areas of the box where the character is indicated should provide a way to differentiate among the 26 letters of the alphabet using an OCR or a machine learning algorithm to be trained.

Tip

To follow along with this example, download the `letterdata.data` file from the Packt Publishing website and save it to your project directory working on one or the other directory.

Before reading the data from the Spark working directory, we confirm that we have received the data with the 16 features that define each example of the letter class. As expected, the letter has 26 levels, as shown in *Figure 20*:

letter	xbox	ybox	width	height	onpix	xbar	ybar	x2bar	y2bar	xybar	x2ybar	xy2bar	xedge	xedgey	yedge	yedgex
T	2	8	3	5	1	8	13	0	6	6	10	8	0	8	0	8
I	5	12	3	7	2	10	5	5	4	13	3	9	2	8	4	10
D	4	11	6	8	6	10	6	2	6	10	3	7	3	7	3	9
N	7	11	6	6	3	5	9	4	6	4	4	10	6	10	2	8
G	2	1	3	1	1	8	6	6	6	6	5	9	1	7	5	10
S	3	2	3	3	2	8	8	7	5	7	5	7	2	8	9	8
O	6	11	7	8	5	7	6	9	6	7	5	9	4	8	5	5
J	3	6	4	4	2	6	6	4	4	14	8	12	1	6	1	6
C	6	11	7	8	3	7	8	7	11	4	7	14	1	7	4	8
M	7	11	11	8	9	3	8	4	5	10	11	10	10	9	5	7
W	12	14	12	8	5	9	10	4	3	5	10	7	10	12	2	6
H	6	9	8	7	6	8	6	6	7	7	7	9	6	8	4	8
G	3	6	4	4	2	6	6	5	5	6	6	9	2	8	4	8
L	2	3	3	4	1	0	1	5	6	0	0	6	0	8	0	8
L	1	3	3	1	1	6	4	1	7	8	3	10	0	7	2	9
X	8	12	8	6	4	3	10	4	7	12	11	9	3	7	3	4
B	5	9	7	7	10	9	8	4	4	6	8	6	6	11	8	7
M	6	9	9	7	6	5	6	3	5	10	9	9	8	5	2	7
G	4	7	6	5	3	6	6	6	8	6	5	9	3	10	4	8
O	4	7	5	5	3	7	7	8	6	7	6	8	3	8	3	8

only showing top 20 rows

Figure 20: A snapshot of the dataset shown as Data Frame

Recall that SVM, Naive Baseyan-based classifier, or any other classifier algorithms, along with their associated learners, require all the features to be numeric. Moreover, each feature is scaled to a fairly small interval.

Also, SVM works well on dense vectorized features and consequently, will perform poorly against the sparse vectorized features. In our case, every feature is an integer. Therefore, we do not need to convert any values into numbers. On the other hand, some of the ranges for these integer variables appear fairly wide.

In practical cases, it might require that we normalize the data against all few features points.

OCR pipeline with Spark ML and Spark MLlib

Because of its accuracy and robustness, let's see whether the SVM is up to the task. As you can see in *Figure 17*, we have a multiclass OCR dataset (with 26 classes, to be more precise); therefore, we need to have a

multiclass classification algorithm, for example, the logistic regression model, since the current implementation of liner SVM in Spark does not support the multi-class classification.

> **Tip**
>
> Please refer to the following URL for more details: http://spark.apache.org/docs/latest/mllib-linear-methods.html#linear-support-vector-machines-svms.

Step 1: Import necessary packages/libraries/APIs

The following is the code to import the necessary packages:

```
import java.util.HashMap;
import java.util.Map;
import org.apache.spark.api.java.JavaRDD;
import
org.apache.spark.api.java.function.Function;
import
org.apache.spark.mllib.classification.LogisticRegressionWithLBFGS;
import
org.apache.spark.mllib.evaluation.MulticlassMetrics;
import
org.apache.spark.mllib.evaluation.MultilabelMetrics;
import org.apache.spark.mllib.linalg.DenseVector;
import org.apache.spark.mllib.linalg.Vector;
import
org.apache.spark.mllib.regression.LabeledPoint;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;
import scala.Tuple2;
```

Step 2: Initialize necessary Spark environment

The following is the code to initialize a Spark environment:

```
  static SparkSession spark = SparkSession
        .builder()
        .appName("OCRPrediction")
            .master("local[*]")
            .config("spark.sql.warehouse.dir",
"E:/Exp/").
            getOrCreate();
```

Here we set the application name as `OCRPrediction`, and the master URL as `local`. The Spark session is the entry point of the program. Please set these parameters accordingly.

Step 3: Read the data file and create a corresponding Dataset and show the first 20 rows

The following is the code to read the data file:

```
String input = "input/letterdata.data";
Dataset<Row> df =
spark.read().format("com.databricks.spark.csv").option("header",
"true").load(input);
  df.show();
```

For the first 20 rows, please refer to *Figure 5*. As we can see, there are 26 characters presented as single characters that need to be predicted; therefore, we need to assign each character a random double value to align the value to the other features. Therefore, in the next step, that is what we'll do.

Step 4: Create a dictionary for assigning each character a double value randomly

The following code is to create a dictionary for assigning each character a double value randomly:

```
final Map<String, Integer>alpha = newHashMap();
    intcount = 0;
    for(chari = 'A'; i<= 'Z'; i++){
      alpha.put(i + "", count++);
      System.out.println(alpha);
    }
```

And here is the mapping output generated from the preceding code segment:

```
{A=0}
{A=0, B=1}
{A=0, B=1, C=2}
{A=0, B=1, C=2, D=3}
{A=0, B=1, C=2, D=3, E=4}
{A=0, B=1, C=2, D=3, E=4, F=5}
{A=0, B=1, C=2, D=3, E=4, F=5, G=6}
{A=0, B=1, C=2, D=3, E=4, F=5, G=6, H=7}
{A=0, B=1, C=2, D=3, E=4, F=5, G=6, H=7, I=8}
{A=0, B=1, C=2, D=3, E=4, F=5, G=6, H=7, I=8, J=9}
{A=0, B=1, C=2, D=3, E=4, F=5, G=6, H=7, I=8, J=9, K=10}
{A=0, B=1, C=2, D=3, E=4, F=5, G=6, H=7, I=8, J=9, K=10, L=11}
{A=0, B=1, C=2, D=3, E=4, F=5, G=6, H=7, I=8, J=9, K=10, L=11, M=12}
{A=0, B=1, C=2, D=3, E=4, F=5, G=6, H=7, I=8, J=9, K=10, L=11, M=12, N=13}
{A=0, B=1, C=2, D=3, E=4, F=5, G=6, H=7, I=8, J=9, K=10, L=11, M=12, N=13, O=14}
{A=0, B=1, C=2, D=3, E=4, F=5, G=6, H=7, I=8, J=9, K=10, L=11, M=12, N=13, O=14, P=15}
{A=0, B=1, C=2, D=3, E=4, F=5, G=6, H=7, I=8, J=9, K=10, L=11, M=12, N=13, O=14, P=15, Q=16}
{A=0, B=1, C=2, D=3, E=4, F=5, G=6, H=7, I=8, J=9, K=10, L=11, M=12, N=13, O=14, P=15, Q=16, R=17}
{A=0, B=1, C=2, D=3, E=4, F=5, G=6, H=7, I=8, J=9, K=10, L=11, M=12, N=13, O=14, P=15, Q=16, R=17, S=18}
{A=0, B=1, C=2, D=3, E=4, F=5, G=6, H=7, I=8, J=9, K=10, L=11, M=12, N=13, O=14, P=15, Q=16, R=17, S=18, T=19}
{A=0, B=1, C=2, D=3, E=4, F=5, G=6, H=7, I=8, J=9, K=10, L=11, M=12, N=13, O=14, P=15, Q=16, R=17, S=18, T=19, U=20}
{A=0, B=1, C=2, D=3, E=4, F=5, G=6, H=7, I=8, J=9, K=10, L=11, M=12, N=13, O=14, P=15, Q=16, R=17, S=18, T=19, U=20, V=21}
{A=0, B=1, C=2, D=3, E=4, F=5, G=6, H=7, I=8, J=9, K=10, L=11, M=12, N=13, O=14, P=15, Q=16, R=17, S=18, T=19, U=20, V=21, W=22}
{A=0, B=1, C=2, D=3, E=4, F=5, G=6, H=7, I=8, J=9, K=10, L=11, M=12, N=13, O=14, P=15, Q=16, R=17, S=18, T=19, U=20, V=21, W=22, X=23}
{A=0, B=1, C=2, D=3, E=4, F=5, G=6, H=7, I=8, J=9, K=10, L=11, M=12, N=13, O=14, P=15, Q=16, R=17, S=18, T=19, U=20, V=21, W=22, X=23, Y=24}
{A=0, B=1, C=2, D=3, E=4, F=5, G=6, H=7, I=8, J=9, K=10, L=11, M=12, N=13, O=14, P=15, Q=16, R=17, S=18, T=19, U=20, V=21, W=22, X=23, Y=24, Z=25}
```

Figure 21: Mapping assignments

Step 5: Creating the labeled point and the feature vector

Create the labeled points and feature vectors for the features combined from the 16 features (that is, 16 columns). Also, save them as Java RDD and dump or cache on disk or memory, and show the sample output:

```
JavaRDD<LabeledPoint> dataRDD =
df.toJavaRDD().map(new Function<Row,
LabeledPoint>() {
      @Override
      public LabeledPoint call(Row row) throws
Exception {

        String letter = row.getString(0);
        double label = alpha.get(letter);
        double[] features= new double
[row.size()];
        for(int i = 1; i < row.size(); i++){
          features[i-1] =
Double.parseDouble(row.getString(i));
        }
        Vector v = new DenseVector(features);

        return new LabeledPoint(label, v);
      }
    });

dataRDD.saveAsTextFile("Output/dataRDD");
System.out.println(dataRDD.collect());
```

If you look carefully at the preceding code segments, we have created an array named features for 16 features altogether and created a dense vector representation, since the dense vector representation is a more compact representation where the contents can be shown as in the following screenshot:

```
1(19.0,[2.0,8.0,3.0,5.0,1.0,8.0,13.0,0.0,6.0,6.0,10.0,8.0,0.0,8.0,0.0,8.0])
2(8.0,[5.0,12.0,3.0,7.0,2.0,10.0,5.0,5.0,4.0,13.0,3.0,9.0,2.0,8.0,4.0,10.0])
3(3.0,[4.0,11.0,6.0,8.0,6.0,10.0,6.0,2.0,6.0,10.0,3.0,7.0,3.0,7.0,3.0,9.0])
4(13.0,[7.0,11.0,6.0,6.0,3.0,5.0,9.0,4.0,6.0,4.0,4.0,10.0,6.0,10.0,2.0,8.0])
5(6.0,[2.0,1.0,3.0,1.0,1.0,8.0,6.0,6.0,6.0,6.0,5.0,9.0,1.0,7.0,5.0,10.0])
6(18.0,[4.0,11.0,5.0,8.0,3.0,8.0,8.0,6.0,9.0,5.0,6.0,6.0,0.0,8.0,9.0,7.0])
7(1.0,[4.0,2.0,5.0,4.0,4.0,8.0,7.0,6.0,6.0,7.0,6.0,6.0,2.0,8.0,7.0,10.0])
8(0.0,[1.0,1.0,3.0,2.0,1.0,8.0,2.0,2.0,2.0,8.0,2.0,8.0,1.0,6.0,2.0,7.0])
9(9.0,[2.0,2.0,4.0,4.0,2.0,10.0,6.0,2.0,6.0,12.0,4.0,8.0,1.0,6.0,1.0,7.0])
0(12.0,[11.0,15.0,13.0,9.0,7.0,13.0,2.0,6.0,2.0,12.0,1.0,9.0,8.0,1.0,1.0,8.0])
1(23.0,[3.0,9.0,5.0,7.0,4.0,8.0,7.0,3.0,8.0,5.0,6.0,8.0,2.0,8.0,6.0,7.0])
2(14.0,[6.0,13.0,4.0,7.0,4.0,6.0,7.0,6.0,3.0,10.0,7.0,9.0,5.0,9.0,5.0,8.0])
3(6.0,[4.0,9.0,6.0,7.0,6.0,7.0,8.0,6.0,2.0,6.0,5.0,11.0,4.0,8.0,7.0,8.0])
4(12.0,[6.0,9.0,8.0,6.0,9.0,7.0,8.0,6.0,5.0,7.0,5.0,8.0,8.0,9.0,8.0,6.0])
5(17.0,[5.0,9.0,5.0,7.0,6.0,6.0,11.0,7.0,3.0,7.0,3.0,9.0,2.0,7.0,5.0,11.0])
```

Figure 22: The Java RDD for the corresponding label and features as vectors

Step 6: Generating the training and test set

Here is the code for generating the test set:

```
JavaRDD<LabeledPoint>[] splits =
dataRDD.randomSplit(new double[] {0.7, 0.3},
12345L);
JavaRDD<LabeledPoint> training = splits[0];
JavaRDD<LabeledPoint> test = splits[1];
```

If you wish to see the snaps of the training or test datasets, you should dump or cache them. The following is a sample code for that:

```
training.saveAsTextFile("Output/training");
test.saveAsTextFile("Output/test");
```

> **Tip**
>
> We have randomly generated the training and the test set for the model to be trained and tested. In our case, it was 70% and 30%, respectively, and 11L as the long seed. Readjust the values based on your dataset. Note that if you add a seed to a Random, you get the same results each time you run your code and these come as primes up to 1062348.

Step 7: Train the model

As you can see, we have a multiclass dataset with 26 classes; therefore, we need to have a multiclass classification algorithm, for example, the logistic regression model:

```
Boolean useFeatureScaling= true;
final LogisticRegressionModel model = new
LogisticRegressionWithLBFGS()

 .setNumClasses(26).setFeatureScaling(useFeatureScaling)
  .run(training.rdd());
```

The preceding code segment builds a model using the training datasets by specifying the number of classes (that is, `26`) and the feature scaling as `Boolean true`. As you can see, we have used the RDD version of the training datasets using `training.rdd()`, since the training datasets are in normal vector format.

Tip

Spark has the support of the multiclass logistic regression algorithm that supports the **Limited-Memory-Broyden-Fletcher-Goldfarb-Shanno** (**LBFGS**) algorithm. In numerical optimization, the **Broyden-Fletcher-Goldfarb-Shanno** (**BFGS**) algorithm is an iterative method for solving unconstrained nonlinear optimization problems.

Step 8: Compute the raw scores on the test dataset

Here is the code to compute the raw scores:

```
JavaRDD<Tuple2<Object, Object>>
predictionAndLabels = test.map(
    new Function<LabeledPoint, Tuple2<Object,
Object>>() {
    public Tuple2<Object, Object>
call(LabeledPoint p) {
    Double prediction =
model.predict(p.features());
    return new Tuple2<Object, Object>(prediction,
p.label());
          }
        }
      );
predictionAndLabels.saveAsTextFile("output/prd2");
```

If you look at the preceding code carefully, you will see that we are actually calculating the predicted features out of the model we created in *Step 7* by making them Java RDD.

Step 9: Predict the outcome for label 8.0 (that is, I) and get the evaluation metrics

The following code illustrates how to predict the outcome:

```
MulticlassMetrics metrics = new
MulticlassMetrics(predictionAndLabels.rdd());
MultilabelMetrics(predictionAndLabels.rdd());
System.out.println(metrics.confusionMatrix());
double precision =
metrics.precision(metrics.labels()[0]);
double recall =
metrics.recall(metrics.labels()[0]);
double tp = 8.0;
double TP = metrics.truePositiveRate(tp);
double FP = metrics.falsePositiveRate(tp);
double WTP = metrics.weightedTruePositiveRate();
```

```
double WFP =
metrics.weightedFalsePositiveRate();
System.out.println("Precision = " + precision);
System.out.println("Recall = " + recall);
System.out.println("True Positive Rate = " + TP);
System.out.println("False Positive Rate = " +
FP);
System.out.println("Weighted True Positive Rate =
" + WTP);
System.out.println("Weighted False Positive Rate
= " + WFP);
```

```
16/06/29 20:28:33 INFO DAGScheduler: ResultStage 119 (collectAsMap at MulticlassMetrics.scala:54)
16/06/29 20:28:33 INFO DAGScheduler: Job 115 finished: collectAsMap at MulticlassMetrics.scala:54,
Precision = 0.7553648068669528
Recall = 0.7553648068669528
True Positive Rate = 0.7396694214876033
False Positive Rate = 0.010746910263299301
Weighted True Positive Rate = 0.755364806866953
Weighted False Positive Rate = 0.009803805809192706
16/06/29 20:28:33 INFO SparkContext: Invoking stop() from shutdown hook
16/06/29 20:28:33 INFO SparkUI: Stopped Spark web UI at http://10.2.17.28:4040
```

Figure 23: Performance metrics for precision and recall

Therefore, the precision is 75%, which is obviously not satisfactory. However, if you are still unsatisfied, the following chapter will look at how to tune parameters so that the prediction accuracy increases.

Tip

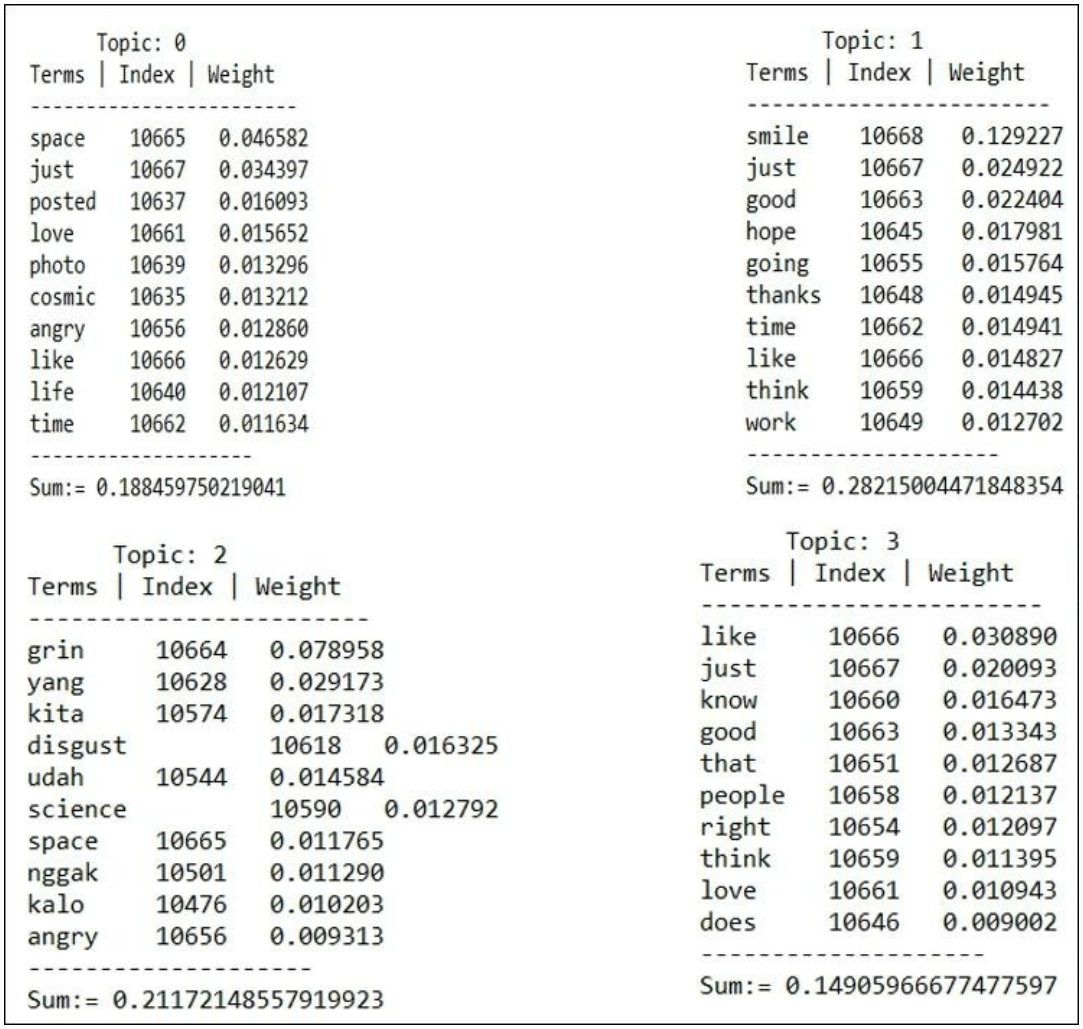
To get an idea about how to calculate the precision, recall, true positive rate, and true negative rate, please refer to the Wikipedia page at https://en.wikipedia.org/wiki/Sensitivity_and_specificity, which discusses the sensitivity and the specificity elaborately. You can also refer to *Powers, David M W (2011). Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness & Correlation(PDF). Journal of Machine Learning Technologies 2 (1): 37-63*.

Topic modeling using Spark MLlib and ML

The topic modeling technique is widely used in the task of mining text from a large collection of documents. These topics can then be used to summarize and organize documents that include the topic terms and their relative weights. Since the release of Spark 1.3, MLlib supports the LDA, which is one of the most successfully used topic-modeling techniques in the area of text mining and **Natural Language Processing** (**NLP**). Moreover, LDA is also the first MLlib algorithm to adopt Spark GraphX.

> **Tip**
>
> To get more information about how the theory behind the LDA works, please refer to *David M. Blei, Andrew Y. Ng and Michael I. Jordan, Latent Dirichlet Allocation, Journal of Machine Learning Research 3 (2003) 993-1022*.

Figure 24 shows the output of the topic distribution from randomly generated tweet text, to be discussed further in Chapter 9, *Advanced Machine Learning with Streaming and Graph Data*. Moreover, we will provide more justification for why we used LDA other than other topic-modeling algorithms in Chapter 9, *Advanced Machine Learning with Streaming and Graph Data*:



```
      Topic: 0
Terms | Index | Weight
-----------------------
space    10665   0.046582
just     10667   0.034397
posted   10637   0.016093
love     10661   0.015652
photo    10639   0.013296
cosmic   10635   0.013212
angry    10656   0.012860
like     10666   0.012629
life     10640   0.012107
time     10662   0.011634
--------------------
Sum:= 0.188459750219041

      Topic: 1
Terms | Index | Weight
-----------------------
smile    10668   0.129227
just     10667   0.024922
good     10663   0.022404
hope     10645   0.017981
going    10655   0.015764
thanks   10648   0.014945
time     10662   0.014941
like     10666   0.014827
think    10659   0.014438
work     10649   0.012702
--------------------
Sum:= 0.28215004471848354

      Topic: 2
Terms | Index | Weight
-----------------------
grin     10664   0.078958
yang     10628   0.029173
kita     10574   0.017318
disgust          10618   0.016325
udah     10544   0.014584
science          10590   0.012792
space    10665   0.011765
nggak    10501   0.011290
kalo     10476   0.010203
angry    10656   0.009313
--------------------
Sum:= 0.21172148557919923

      Topic: 3
Terms | Index | Weight
-----------------------
like     10666   0.030890
just     10667   0.020093
know     10660   0.016473
good     10663   0.013343
that     10651   0.012687
people   10658   0.012137
right    10654   0.012097
think    10659   0.011395
love     10661   0.010943
does     10646   0.009002
--------------------
Sum:= 0.14905966677477597
```

Figure 24: The topic distribution and how it looks like

In this section, we will look at an example of topic modeling using the LDA algorithm of Spark MLlib with unstructured raw tweets datasets.

Topic modeling with Spark MLlib

In this subsection, we represent a semi-automated technique of topic modeling using Spark. The following steps show the topic modeling from data reading to printing the topics, along with their term-weights. Using other options as defaults, we train LDA on the dataset downloaded from the GitHub URL at https://github.com/minghui/Twitter-LDA/tree/master/data/Data4Model/test.

Step 1: Load required packages and APIs

Here is the code to load the required packages:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.Serializable;
```

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
import org.apache.spark.ml.clustering.LDA;
import org.apache.spark.ml.clustering.LDAModel;
import org.apache.spark.ml.feature.ChiSqSelector;
import org.apache.spark.ml.feature.HashingTF;
import org.apache.spark.ml.feature.IDF;
import org.apache.spark.ml.feature.IDFModel;
import
org.apache.spark.ml.feature.RegexTokenizer;
import
org.apache.spark.ml.feature.StopWordsRemover;
import org.apache.spark.ml.feature.StringIndexer;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;
import org.apache.spark.sql.types.DataTypes;
```

Step 2: Create a Spark session

Here is the code to create a Spark session:

```
static SparkSession spark = SparkSession
        .builder()
        .appName("JavaLDAExample")
        .master("local[*]")
        .config("spark.sql.warehouse.dir",
"E:/Exp/")
        .getOrCreate();
```

Step 3: Reading and seeing the content of the datasets

The following code illustrates how to read and see the content of the datasets:

```
Dataset<Row> df =
spark.read().text("input/test/*.txt");
```

Please note that the use of the character `*` indicates to read all the text files in the input/text directory in the project path. If we print the first 20 rows, just use the following code and you will see the following text:

```
df.show();
```

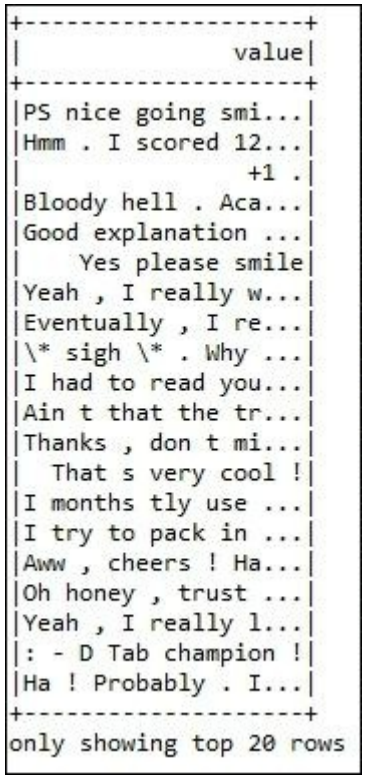


Figure 25: First 20 rows of the texts

From the preceding screenshot, it is clear that the text files we are using are nothing but very unstructured texts containing the column name label. They therefore need to be pre-processed using the feature transformation using the regular expression tokenizer before we can use them for our purposes.

Step 4: Feature Transformers using RegexTokenizer

Here is the code for `RegexTokenizer`:

```
RegexTokenizer regexTokenizer1 = new
RegexTokenizer().setInputCol("value").setOutputCol("labelText").setPatter
\t.*$");
```

If you look at the preceding code segment carefully, you will see that we have specified our input column name as `value` and our output column name as `labelText` and the pattern. Now create another data frame using the regular expression tokenizer we just tokenized using the following code segment:

```
Dataset<Row> labelTextDataFrame =
regexTokenizer1.transform(df);
```

Now, let's see what the new data frame `labelTextDataFrame` contains using the following statement:

```
labelTextDataFrame.show();
```

```
+--------------------+--------------------+
|               value|           labelText|
+--------------------+--------------------+
|PS nice going smi...|[ps nice going sm...|
|Hmm . I scored 12...|[hmm . i scored 1...|
|                +1 .|              [+1 .]|
|Bloody hell . Aca...|[bloody hell . ac...|
|Good explanation ...|[good explanation...|
|    Yes please smile|  [yes please smile]|
|Yeah , I really w...|[yeah , i really ...|
|Eventually , I re...|[eventually , i r...|
|\* sigh \* . Why ...|[\* sigh \* . why...|
|I had to read you...|[i had to read yo...|
|Ain t that the tr...|[ain t that the t...|
|Thanks , don t mi...|[thanks , don t m...|
|  That s very cool !|[that s very cool !]|
|I months tly use ...|[i months tly use...|
|I try to pack in ...|[i try to pack in...|
|Aww , cheers ! Ha...|[aww , cheers ! h...|
|Oh honey , trust ...|[oh honey , trust...|
|Yeah , I really l...|[yeah , i really ...|
|: - D Tab champion !|[: - d tab champi...|
|Ha ! Probably . I...|[ha ! probably . ...|
+--------------------+--------------------+
only showing top 20 rows
```

Figure 26: A new column with characters converted into the corresponding lowercase characters

The preceding screenshot (*Figure 26*) shows that the tokenizer created a new column and that mostly, the uppercase words or characters have been converted into the corresponding lowercase characters. Since topic modeling cares about the term-weight and frequency of each input word, we need to separate the words from the label texts, which is done by using the following code segments:

```
RegexTokenizer regexTokenizer2 = new
RegexTokenizer().setInputCol("value").setOutputCol("text").setPattern("\
\W");
```

Now let's create another data frame and see the results of the transformation using the following code:

```
Dataset<Row> labelFeatureDataFrame =
regexTokenizer2.transform(labelTextDataFrame);
```

```
labelFeaturedDataFrame.show();
```

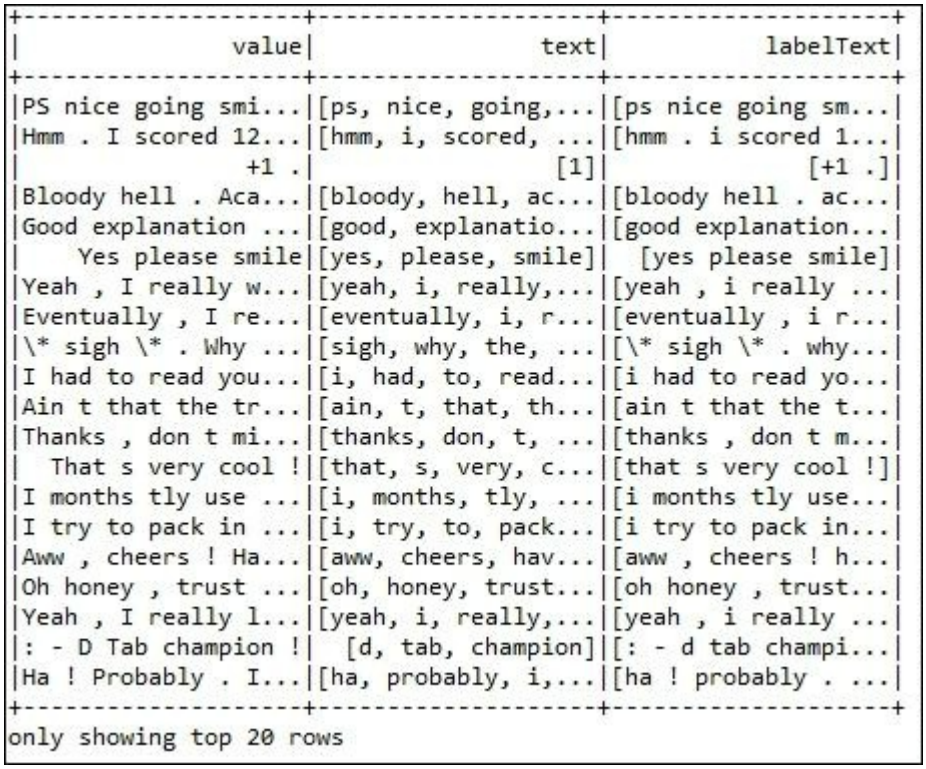
```
+--------------------+--------------------+--------------------+
|               value|           labelText|                text|
+--------------------+--------------------+--------------------+
|PS nice going smi...|[ps nice going sm...|[ps, nice, going,...|
|Hmm . I scored 12...|[hmm . i scored 1...|[hmm, i, scored, ...|
|                +1 .|              [+1 .]|                 [1]|
|Bloody hell . Aca...|[bloody hell . ac...|[bloody, hell, ac...|
|Good explanation ...|[good explanation...|[good, explanatio...|
|    Yes please smile|  [yes please smile]|[yes, please, smile]|
|Yeah , I really w...|[yeah , i really ...|[yeah, i, really,...|
|Eventually , I re...|[eventually , i r...|[eventually, i, r...|
|\* sigh \* . Why ...|[\* sigh \* . why...|[sigh, why, the, ...|
|I had to read you...|[i had to read yo...|[i, had, to, read...|
|Ain t that the tr...|[ain t that the t...|[ain, t, that, th...|
|Thanks , don t mi...|[thanks , don t m...|[thanks, don, t, ...|
|  That s very cool !|[that s very cool !]|[that, s, very, c...|
|I months tly use ...|[i months tly use...|[i, months, tly, ...|
|I try to pack in ...|[i try to pack in...|[i, try, to, pack...|
|Aww , cheers ! Ha...|[aww , cheers ! h...|[aww, cheers, hav...|
|Oh honey , trust ...|[oh honey , trust...|[oh, honey, trust...|
|Yeah , I really l...|[yeah , i really ...|[yeah, i, really,...|
|: - D Tab champion !|[: - d tab champi...|  [d, tab, champion]|
|Ha ! Probably . I...|[ha ! probably . ...|[ha, probably, i,...|
+--------------------+--------------------+--------------------+
only showing top 20 rows
```

Figure 27: Label texts as separate words separated by a comma

From the preceding screenshot (*Figure 27*), we can see that a new column, `label` has been added, which shows the label texts as separate words, separated by a comma.

Now, since we have a bunch of texts available, to make the prediction and topic modeling easier, we need to make the indexing for the words we split. But before that, we need to swap the `labelText` and `text` in a new data frame, as shown in *Figure 28*. To check if it has really happened, just print the newly created data frame:

```
Dataset<Row> newDF = labelFeatureDataFrame
        .withColumn("labelTextTemp",
labelFeatureDataFrame.col("labelText")
          .cast(DataTypes.StringType))
 .drop(labelFeatureDataFrame.col("labelText")).withColumnRenamed("labelTe
"labelText");
newDF.show();
```



value	text	labelText
PS nice going smi...	[ps, nice, going,...	[ps nice going sm...
Hmm . I scored 12...	[hmm, i, scored, ...	[hmm . i scored 1...
+1 .	[1]	[+1 .]
Bloody hell . Aca...	[bloody, hell, ac...	[bloody hell . ac...
Good explanation ...	[good, explanatio...	[good explanation...
Yes please smile	[yes, please, smile]	[yes please smile]
Yeah , I really w...	[yeah, i, really,...	[yeah , i really ...
Eventually , I re...	[eventually, i, r...	[eventually , i r...
* sigh * . Why ...	[sigh, why, the, ...	[* sigh * . why...
I had to read you...	[i, had, to, read...	[i had to read yo...
Ain t that the tr...	[ain, t, that, th...	[ain t that the t...
Thanks , don t mi...	[thanks, don, t, ...	[thanks , don t m...
That s very cool !	[that, s, very, c...	[that s very cool !]
I months tly use ...	[i, months, tly, ...	[i months tly use...
I try to pack in ...	[i, try, to, pack...	[i try to pack in...
Aww , cheers ! Ha...	[aww, cheers, hav...	[aww , cheers ! h...
Oh honey , trust ...	[oh, honey, trust...	[oh honey , trust...
Yeah , I really l...	[yeah, i, really,...	[yeah , i really ...
: - D Tab champion !	[d, tab, champion]	[: - d tab champi...
Ha ! Probably . I...	[ha, probably, i,...	[ha ! probably . ...

only showing top 20 rows

Figure 28: Swapping the labelText and text in a new data frame

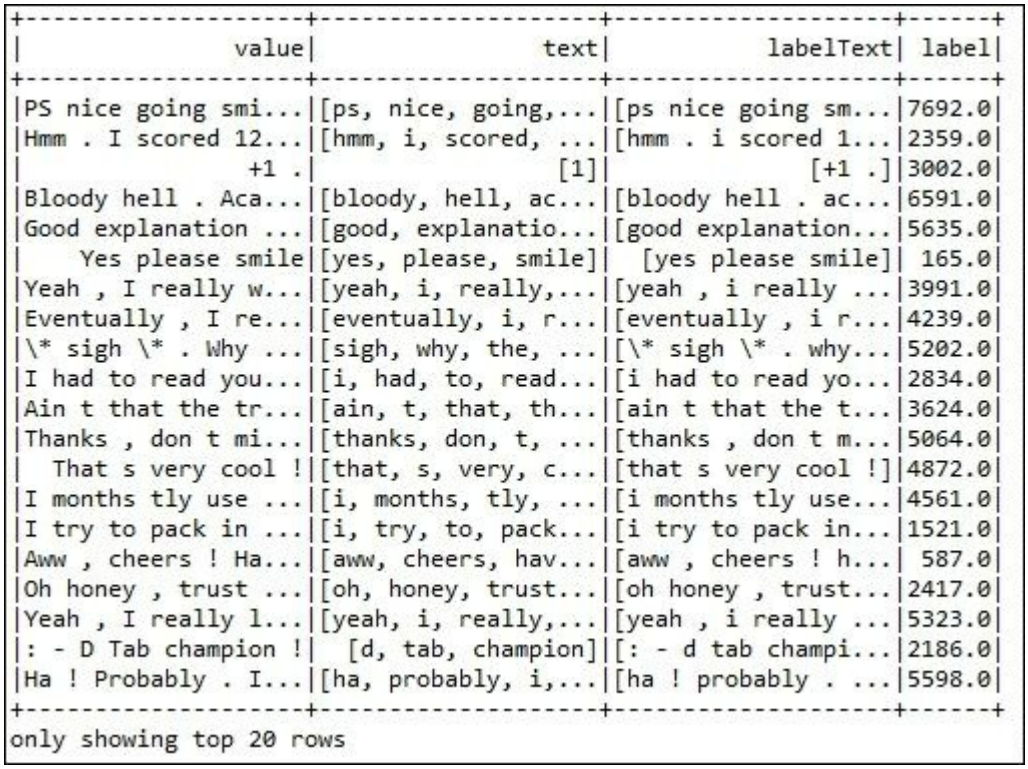
Step 5: Feature transformation by means of string indexer

Here is the code for feature transformation:

```
StringIndexer indexer = new
StringIndexer().setInputCol("labelText").setOutputCol("label");
```

Now create a new data frame for the data frame `newDF` we created in *Step 2*, and look at the contents of the data frame. Note that we have selected the old column `labelText` and set the new column simply as `label`:

```
Dataset<Row> indexed =
indexer.fit(newDF).transform(newDF);
indexed.select(indexed.col("labelText"),
indexed.col("label"),
indexed.col("text")).show();
Indexed.show();
```



value	text	labelText	label
PS nice going smi...	[ps, nice, going,...	[ps nice going sm...	7692.0
Hmm . I scored 12...	[hmm, i, scored, ...	[hmm . i scored 1...	2359.0
+1 .	[1]	[+1 .]	3002.0
Bloody hell . Aca...	[bloody, hell, ac...	[bloody hell . ac...	6591.0
Good explanation ...	[good, explanatio...	[good explanation...	5635.0
Yes please smile	[yes, please, smile]	[yes please smile]	165.0
Yeah , I really w...	[yeah, i, really,...	[yeah , i really ...	3991.0
Eventually , I re...	[eventually, i, r...	[eventually , i r...	4239.0
* sigh * . Why ...	[sigh, why, the, ...	[* sigh * . why...	5202.0
I had to read you...	[i, had, to, read...	[i had to read yo...	2834.0
Ain t that the tr...	[ain, t, that, th...	[ain t that the t...	3624.0
Thanks , don t mi...	[thanks, don, t, ...	[thanks , don t m...	5064.0
That s very cool !	[that, s, very, c...	[that s very cool !]	4872.0
I months tly use ...	[i, months, tly, ...	[i months tly use...	4561.0
I try to pack in ...	[i, try, to, pack...	[i try to pack in...	1521.0
Aww , cheers ! Ha...	[aww, cheers, hav...	[aww , cheers ! h...	587.0
Oh honey , trust ...	[oh, honey, trust...	[oh honey , trust...	2417.0
Yeah , I really l...	[yeah, i, really,...	[yeah , i really ...	5323.0
: - D Tab champion !	[d, tab, champion]	[: - d tab champi...	2186.0
Ha ! Probably . I...	[ha, probably, i,...	[ha ! probably	5598.0

only showing top 20 rows

Figure 29: Corresponding labels against the labelText column

So, as shown in *Figure 29*, we got a new column, `label`, which contains the corresponding labels against the `labelText` column. The very next step is about removing the stop words.

Step 6: Feature transformation (removing the stop words)

Here is the code for feature transformation for removing the stop words:

```
StopWordsRemover remover = new
StopWordsRemover();
String[] stopwords = remover.getStopWords();
remover.setStopWords(stopwords).setInputCol("text").setOutputCol("filtere
```

The current implementation of the `StopWordsRemover` class of Spark contains the following words as stop words. Since we don't have any precondition, we have used those words directly:



```
                "a", "about", "above", "across", "after", "afterwards", "again",
"against", "all", "almost", "alone", "along", "already", "also", "although", "always",
"am", "among", "amongst", "amoungst", "amount", "an", "and", "another",
"any", "anyhow", "anyone", "anything", "anyway", "anywhere", "are",
"around", "as", "at", "back", "be", "became", "because", "become",
"becomes", "becoming", "been", "before", "beforehand", "behind", "being",
"below", "beside", "besides", "between", "beyond", "bill", "both",
"bottom", "but", "by", "call", "can", "cannot", "cant", "co", "con",
"could", "couldnt", "cry", "de", "describe", "detail", "do", "done",
"down", "due", "during", "each", "eg", "eight", "either", "eleven", "else",
"elsewhere", "empty", "enough", "etc", "even", "ever", "every", "everyone",
"everything", "everywhere", "except", "few", "fifteen", "fify", "fill",
"find", "fire", "first", "five", "for", "former", "formerly", "forty",
"found", "four", "from", "front", "full", "further", "get", "give", "go",
"had", "has", "hasnt", "have", "he", "hence", "her", "here", "hereafter",
"hereby", "herein", "hereupon", "hers", "herself", "him", "himself", "his",
"how", "however", "hundred", "i", "ie", "if", "in", "inc", "indeed",
"interest", "into", "is", "it", "its", "itself", "keep", "last", "latter",
"latterly", "least", "less", "ltd", "made", "many", "may", "me",
"meanwhile", "might", "mill", "mine", "more", "moreover", "most", "mostly",
"move", "much", "must", "my", "myself", "name", "namely", "neither",
"never", "nevertheless", "next", "nine", "no", "nobody", "none", "noone",
"nor", "not", "nothing", "now", "nowhere", "of", "off", "often", "on",
"once", "one", "only", "onto", "or", "other", "others", "otherwise", "our",
"ours", "ourselves", "out", "over", "own", "part", "per", "perhaps",
"please", "put", "rather", "re", "same", "see", "seem", "seemed",
"seeming", "seems", "serious", "several", "she", "should", "show", "side",
"since", "sincere", "six", "sixty", "so", "some", "somehow", "someone",
"something", "sometime", "sometimes", "somewhere", "still", "such",
"system", "take", "ten", "than", "that", "the", "their", "them",
"themselves", "then", "thence", "there", "thereafter", "thereby",
"therefore", "therein", "thereupon", "these", "they", "thick", "thin",
"third", "this", "those", "though", "three", "through", "throughout",
"thru", "thus", "to", "together", "too", "top", "toward", "towards",
"twelve", "twenty", "two", "un", "under", "until", "up", "upon", "us",
"very", "via", "was", "we", "well", "were", "what", "whatever", "when",
"whence", "whenever", "where", "whereafter", "whereas", "whereby",
"wherein", "whereupon", "wherever", "whether", "which", "while", "whither",
"who", "whoever", "whole", "whom", "whose", "why", "will", "with",
"within", "without", "would", "yet", "you", "your", "yours", "yourself", "yourselves")
```

Figure 30: A bunch of stop words provided by Spark for text analytics

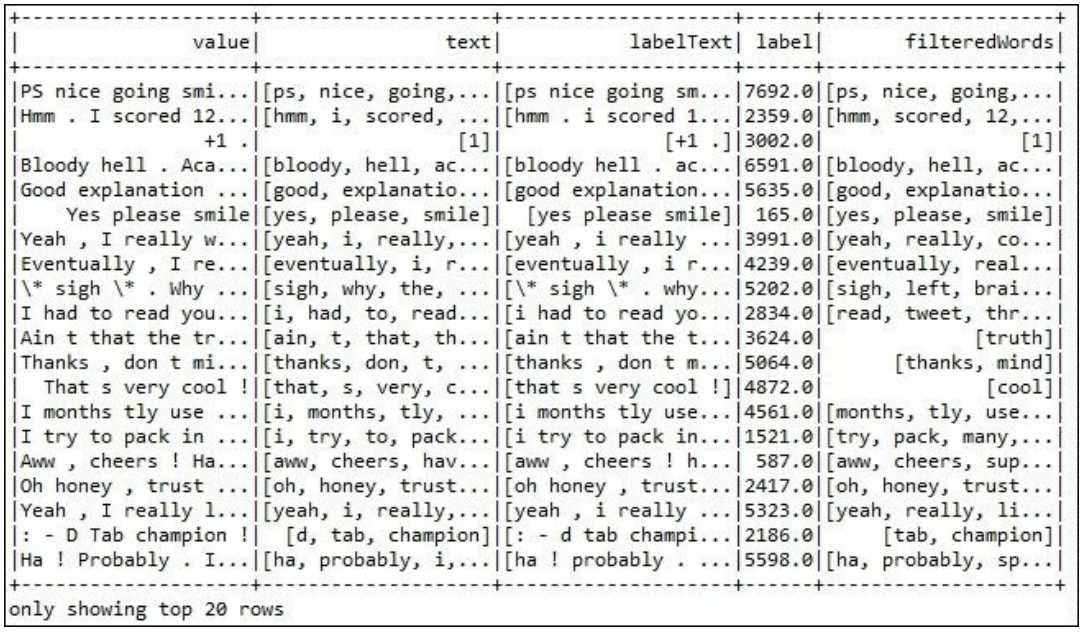
Step 7: Create a filtered dataset by removing stop words

Here is the code to create a filtered dataset by removing stop words:

```
Dataset<Row> filteredDF =
remover.transform(indexed);
filteredDF.show();
filteredDF.select(filteredDF.col("label"),
filteredDF.col("filteredWords")).show();
```

Now create a new data frame for the filtered words (that is, excluding the stop words). Let's look at the contents of the filtered dataset:

```
Dataset<Row> featurizedData =
hashingTF.transform(filteredDF);
featurizedData.show();
```



value	text	labelText	label	filteredWords
PS nice going smi...	[ps, nice, going,...	[ps nice going sm...	7692.0	[ps, nice, going,...
Hmm . I scored 12...	[hmm, i, scored, ...	[hmm . i scored 1...	2359.0	[hmm, scored, 12,...
+1 .	[1]	[+1 .]	3002.0	[1]
Bloody hell . Aca...	[bloody, hell, ac...	[bloody hell . ac...	6591.0	[bloody, hell, ac...
Good explanation ...	[good, explanatio...	[good explanation...	5635.0	[good, explanatio...
Yes please smile	[yes, please, smile]	[yes please smile]	165.0	[yes, please, smile]
Yeah , I really w...	[yeah, i, really,...	[yeah , i really ...	3991.0	[yeah, really, co...
Eventually , I re...	[eventually, i, r...	[eventually , i r...	4239.0	[eventually, real...
* sigh * . Why ...	[sigh, why, the, ...	[* sigh * . why...	5202.0	[sigh, left, brai...
I had to read you...	[i, had, to, read...	[i had to read yo...	2834.0	[read, tweet, thr...
Ain t that the tr...	[ain, t, that, th...	[ain t that the t...	3624.0	[truth]
Thanks , don t mi...	[thanks, don, t, ...	[thanks , don t m...	5064.0	[thanks, mind]
That s very cool !	[that, s, very, c...	[that s very cool !]	4872.0	[cool]
I months tly use ...	[i, months, tly, ...	[i months tly use...	4561.0	[months, tly, use...
I try to pack in ...	[i, try, to, pack...	[i try to pack in...	1521.0	[try, pack, many,...
Aww , cheers ! Ha...	[aww, cheers, hav...	[aww , cheers ! h...	587.0	[aww, cheers, sup...
Oh honey , trust ...	[oh, honey, trust...	[oh honey , trust...	2417.0	[oh, honey, trust...
Yeah , I really l...	[yeah, i, really,...	[yeah , i really ...	5323.0	[yeah, really, li...
: - D Tab champion !	[d, tab, champion]	[: - d tab champi...	2186.0	[tab, champion]
Ha ! Probably . I...	[ha, probably, i,...	[ha ! probably	5598.0	[ha, probably, sp...

only showing top 20 rows

Figure 31: Filtered words excluding the stop words

Step 8: Feature extraction using HashingTF

Here is the code for feature extraction using HashingTF:

```
int numFeatures = 5;
HashingTF hashingTF = new
HashingTF().setInputCol("filteredWords").setOutputCol("rawFeatures").setN
```

In the previous code, we have done the HashingTF for only five features, for simplicity. Now create another data frame out of the extracted features on the old data frame (that is, `filteredDF`) and show the same output:

```
Dataset<Row> featurizedData =
hashingTF.transform(filteredDF);
        featurizedData.show();
```

value	text	labelText	label	filteredWords	rawFeatures
PS nice going smi...	[ps, nice, going,...	[ps nice going sm...	7692.0	[ps, nice, going,...	(100,[5,11,70,93]...
Hmm . I scored 12...	[hmm, i, scored, ...	[hmm . i scored 1...	2359.0	[hmm, scored, 12,...	(100,[9,13,20,21,...
+1 .	[1]	[+1 .]	3002.0	[1]	(100,[44],[1.0])
Bloody hell . Aca...	[bloody, hell, ac...	[bloody hell . ac...	6591.0	[bloody, hell, ac...	(100,[17,25,29,38...
Good explanation ...	[good, explanatio...	[good explanation...	5635.0	[good, explanatio...	(100,[47,68,93,96...
Yes please smile	[yes, please, smile]	[yes please smile]	165.0	[yes, please, smile]	(100,[41,55,70],[...
Yeah , I really w...	[yeah, i, really,...	[yeah , i really ...	3991.0	[yeah, really, co...	(100,[10,18,49,70...
Eventually , I re...	[eventually, i, r...	[eventually , i r...	4239.0	[eventually, real...	(100,[4,41,58],[1...
* sigh * . Why ...	[sigh, why, the, ...	[* sigh * . why...	5202.0	[sigh, left, brai...	(100,[26,29,60,62...
I had to read you...	[i, had, to, read...	[i had to read yo...	2834.0	[read, tweet, thr...	(100,[4,14,32,33,...
Ain t that the tr...	[ain, t, that, th...	[ain t that the t...	3624.0	[truth]	(100,[0],[1.0])
Thanks , don t mi...	[thanks, don, t, ...	[thanks , don t m...	5064.0	[thanks, mind]	(100,[30,84],[1.0...
That s very cool !	[that, s, very, c...	[that s very cool !]	4872.0	[cool]	(100,[95],[1.0])
I months tly use ...	[i, months, tly, ...	[i months tly use...	4561.0	[months, tly, use...	(100,[7,15,34,35,...
I try to pack in ...	[i, try, to, pack...	[i try to pack in...	1521.0	[try, pack, many,...	(100,[45,47,59,85...
Aww , cheers ! Ha...	[aww, cheers, hav...	[aww , cheers ! h...	587.0	[aww, cheers, sup...	(100,[19,59,70,72...
Oh honey , trust ...	[oh, honey, trust...	[oh honey , trust...	2417.0	[oh, honey, trust...	(100,[11,13,29,32...
Yeah , I really l...	[yeah, i, really,...	[yeah , i really ...	5323.0	[yeah, really, li...	(100,[10,30,37,49...
: - D Tab champion !	[d, tab, champion]	[: - d tab champi...	2186.0	[tab, champion]	(100,[47,61],[1.0...
Ha ! Probably . I...	[ha, probably, i,...	[ha ! probably	5598.0	[ha, probably, sp...	(100,[35,57,58,66...

only showing top 20 rows

Figure 32: Data frame out of the extracted features on the old data frame, filteredDF

> **Tip**
>
> For more information and API documentation details for feature transformation, estimator, and hashing, please refer to the Spark website at https://spark.apache.org/docs/latest/ml-features.html.

Step 9: Feature extraction using IDF estimator

```
IDF idf = new
IDF().setInputCol("rawFeatures").setOutputCol("features");
IDFModel idfModel = idf.fit(featurizedData);
```

The preceding code creates new features from the raw features by fitting the `idfModel` that takes the featured data frame (that is, `featurizedData`) in step 5. Now let's create and show a new data frame for the rescaled data using the estimator we just created (that is, `idfModel`), which consumes the old data frame for the featurized data (that is, `featurizedData`):

```
Dataset<Row> rescaledData =
idfModel.transform(featurizedData);
rescaledData.show().
```

value	text	labelText	label	filteredWords	rawFeatures	features
PS nice going smi...	[ps, nice, going,...	[ps nice going sm...	7692.0	[ps, nice, going,...	(100,[5,11,70,93]...	(100,[5,11,70,93]...
Hmm . I scored 12...	[hmm, i, scored, ...	[hmm . i scored 1...	2359.0	[hmm, scored, 12,...	(100,[9,13,20,21,...	(100,[9,13,20,21,...
+1 .	[1]	[+1 .]	3002.0	[1]	(100,[44],[1.0])	(100,[44],[2.4623...
Bloody hell . Aca...	[bloody, hell, ac...	[bloody hell . ac...	6591.0	[bloody, hell, ac...	(100,[17,25,29,38...	(100,[17,25,29,38...
Good explanation ...	[good, explanatio...	[good explanation...	5635.0	[good, explanatio...	(100,[47,68,93,96...	(100,[47,68,93,96...
Yes please smile	[yes, please, smile]	[yes please smile]	165.0	[yes, please, smile]	(100,[41,55,70],[...	(100,[41,55,70],[...
Yeah , I really w...	[yeah, i, really,...	[yeah , i really ...	3991.0	[yeah, really, co...	(100,[10,18,49,70...	(100,[10,18,49,70...
Eventually , I re...	[eventually, i, r...	[eventually , i r...	4239.0	[eventually, real...	(100,[4,41,58],[1...	(100,[4,41,58],[2...
* sigh * . Why ...	[sigh, why, the, ...	[* sigh * . why...	5202.0	[sigh, left, brai...	(100,[26,29,60,62...	(100,[26,29,60,62...
I had to read you...	[i, had, to, read...	[i had to read yo...	2834.0	[read, tweet, thr...	(100,[4,14,32,33,...	(100,[4,14,32,33,...
Ain t that the tr...	[ain, t, that, th...	[ain t that the t...	3624.0	[truth]	(100,[0],[1.0])	(100,[0],[2.61720...
Thanks , don t mi...	[thanks, don, t, ...	[thanks , don t m...	5064.0	[thanks, mind]	(100,[30,84],[1.0...	(100,[30,84],[2.0...
That s very cool !	[that, s, very, c...	[that s very cool !]	4872.0	[cool]	(100,[95],[1.0])	(100,[95],[2.1125...
I months tly use ...	[i, months, tly, ...	[i months tly use...	4561.0	[months, tly, use...	(100,[7,15,34,35,...	(100,[7,15,34,35,...
I try to pack in ...	[i, try, to, pack...	[i try to pack in...	1521.0	[try, pack, many,...	(100,[45,47,59,85...	(100,[45,47,59,85...
Aww , cheers ! Ha...	[aww, cheers, hav...	[aww , cheers ! h...	587.0	[aww, cheers, sup...	(100,[19,59,70,72...	(100,[19,59,70,72...
Oh honey , trust ...	[oh, honey, trust...	[oh honey , trust...	2417.0	[oh, honey, trust...	(100,[11,13,29,32...	(100,[11,13,29,32...
Yeah , I really l...	[yeah, i, really,...	[yeah , i really ...	5323.0	[yeah, really, li...	(100,[10,30,37,49...	(100,[10,30,37,49...
: - D Tab champion !	[d, tab, champion]	[: - d tab champi...	2186.0	[tab, champion]	(100,[47,61],[1.0...	(100,[47,61],[2.9...
Ha ! Probably . I...	[ha, probably, i,...	[ha ! probably	5598.0	[ha, probably, sp...	(100,[35,57,58,66...	(100,[35,57,58,66...

only showing top 20 rows

Figure 33: The rescaled data using the estimator

Step 10: Chi-Squared feature selection

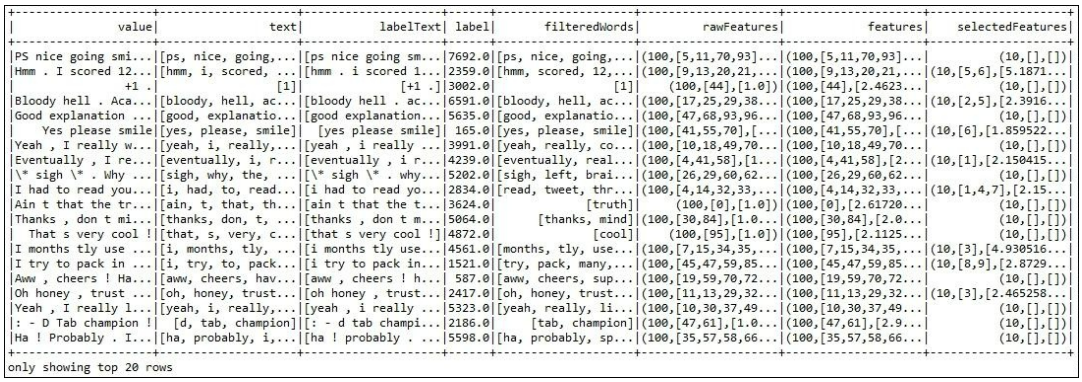
The Chi-Squared feature selection selects categorical features to use for predicting a categorical label. The following code segment does this selection:

```
ChiSqSelector selector = new
org.apache.spark.ml.feature.ChiSqSelector();
```

```
selector.setNumTopFeatures(5).setFeaturesCol("features").setLabelCol("lab
```

Now create another data frame for the selected features, as follows:

```
Dataset<Row> result =
selector.fit(rescaledData).transform(rescaledData);
result.show();
```



value	text	labelText	label	filteredWords	rawFeatures	features	selectedFeatures
PS nice going smi...	[ps, nice, going,...	[ps nice going sm...	7692.0	[ps, nice, going,...	(100,[5,11,70,93]...	(100,[5,11,70,93]...	(10,[],[])
Hmm . I scored 12...	[hmm, i, scored, ...	[hmm . i scored 1...	2359.0	[hmm, scored, 12,...	(100,[9,13,20,21,...	(100,[9,13,20,21,...	(10,[5,6],[5.1871...
+1 .	[1]	[+1 .]	3002.0	[1]	(100,[44],[1.0])	(100,[44],[2.4623...	(10,[],[])
Bloody hell . Aca...	[bloody, hell, ac...	[bloody hell . ac...	6591.0	[bloody, hell, ac...	(100,[17,25,29,38...	(100,[17,25,29,38...	(10,[2,5],[2.3916...
Good explanation ...	[good, explanatio...	[good explanation...	5635.0	[good, explanatio...	(100,[47,68,93,96...	(100,[47,68,93,96...	(10,[],[])
Yes please smile	[yes, please, smile]	[yes please smile]	165.0	[yes, please, smile]	(100,[41,55,70],[...	(100,[41,55,70],[...	(10,[6],[1.859522...
Yeah , I really w...	[yeah, i, really,...	[yeah , i really ...	3991.0	[yeah, really, co...	(100,[10,18,49,70...	(100,[10,18,49,70...	(10,[],[])
Eventually , I re...	[eventually, i, r...	[eventually , i r...	4239.0	[eventually, real...	(100,[4,41,58],[1...	(100,[4,41,58],[2...	(10,[1],[2.150415...
* sigh * . Why ...	[sigh, why, the, ...	[* sigh * . why...	5202.0	[sigh, left, brai...	(100,[26,29,60,62...	(100,[26,29,60,62...	(10,[],[])
I had to read you...	[i, had, to, read...	[i had to read yo...	2834.0	[read, tweet, thr...	(100,[4,14,32,33,...	(100,[4,14,32,33,...	(10,[1,4,7],[2.15...
Ain t that the tr...	[ain, t, that, th...	[ain t that the t...	3624.0	[truth]	(100,[0],[1.0])	(100,[0],[2.61720...	(10,[],[])
Thanks , don t mi...	[thanks, don, t, ...	[thanks , don t m...	5064.0	[thanks, mind]	(100,[30,84],[1.0...	(100,[30,84],[2.0...	(10,[],[])
That s very cool !	[that, s, very, c...	[that s very cool !]	4872.0	[cool]	(100,[95],[1.0])	(100,[95],[2.1125...	(10,[],[])
I months tly use ...	[i, months, tly, ...	[i months tly use...	4561.0	[months, tly, use...	(100,[7,15,34,35,...	(100,[7,15,34,35,...	(10,[3],[4.930516...
I try to pack in ...	[i, try, to, pack...	[i try to pack in...	1521.0	[try, pack, many,...	(100,[45,47,59,85...	(100,[45,47,59,85...	(10,[8,9],[2.8729...
Aww , cheers ! Ha...	[aww, cheers, hav...	[aww , cheers ! h...	587.0	[aww, cheers, sup...	(100,[19,59,70,72...	(100,[19,59,70,72...	(10,[],[])
Oh honey , trust ...	[oh, honey, trust...	[oh honey , trust...	2417.0	[oh, honey, trust...	(100,[11,13,29,32...	(100,[11,13,29,32...	(10,[3],[2.465258...
Yeah , I really l...	[yeah, i, really,...	[yeah , i really ...	5323.0	[yeah, really, li...	(100,[10,30,37,49...	(100,[10,30,37,49...	(10,[],[])
: - D Tab champion !	[d, tab, champion]	[: - d tab champi...	2186.0	[tab, champion]	(100,[47,61],[1.0...	(100,[47,61],[2.9...	(10,[],[])
Ha ! Probably . I...	[ha, probably, i,...	[ha ! probably	5598.0	[ha, probably, sp...	(100,[35,57,58,66...	(100,[35,57,58,66...	(10,[],[])

only showing top 20 rows

Figure 34: Chi-Squared feature selection

You can see from the preceding output/screenshot that our data is ready for the LDA model to be trained and do the topic modeling.

Step 11: Create and train an LDA model

Create and train the LDA model using the training datasets (that is, the data frame result) by specifying the *K* (which is the number of clusters the topic modeling must have to be >1, where the default value is 10) and max iteration:

```
long value = 5;
LDA lda = new
LDA().setK(10).setMaxIter(10).setSeed(value);
LDAModel model = lda.fit(result);
```

Now that we have the model trained, fitted, and ready for our purposes, let's look at our output. However, before doing that, we need to have a data frame that could capture the topic-related metrics. Use the following code:

```
System.out.println(model.vocabSize());
Dataset<Row> topics = model.describeTopics(5);
org.apache.spark.ml.linalg.Matrix metric =
model.topicsMatrix();
```

Now let's see the topics distribution. Look at the previous Dataset:

```
System.out.println(metric);
```

```
topics.show(false);
```

```
+-----+-------------------+----------------------------------------------------------------------------------------------------------------------------------------------------------------+
|topic|termIndices        |termWeights                                                                                                                                                     |
+-----+-------------------+----------------------------------------------------------------------------------------------------------------------------------------------------------------+
|0    |[94, 46, 5, 87, 1] |[0.05724679992194226, 0.05404561522861147, 0.038807677303019136, 0.038303402629897565, 0.037319310897072036]|
|1    |[64, 70, 24, 21, 28]|[0.058792671254640015, 0.0517525299663842, 0.04076539720990001, 0.03554140711651635, 0.032482319332523386]  |
|2    |[90, 42, 54, 72, 84]|[0.058705187341844614, 0.05363038917852419, 0.04998344999185573, 0.048644676451791 88, 0.04659215412613283]  |
|3    |[39, 58, 95, 18, 27]|[0.06946449925876863, 0.06784185618186234, 0.053551868720033736, 0.04780806421245308, 0.04332505013973256]  |
|4    |[83, 4, 73, 3, 37]  |[0.05473062726441959, 0.04497848764939943, 0.04480631763631575, 0.042920867455642976, 0.03738129673058811]  |
|5    |[22, 56, 96, 55, 61]|[0.059591744187659 55, 0.059290412735716475, 0.04604290672179959, 0.04340580319185875, 0.04062792155505264]  |
|6    |[40, 49, 65, 60, 19]|[0.07301343070333365, 0.05038019672074339, 0.04390529167156271, 0.04357692552008894, 0.041187348875585235]  |
|7    |[9, 25, 99, 66, 23] |[0.0944316122016 5724, 0.057819784829171 81, 0.0543292899918784, 0.05171177605698608, 0.05131017500119186]    |
|8    |[41, 89, 76, 74, 71]|[0.06027495645105484, 0.05358297019762894, 0.05250452067197807, 0.05195165321919218, 0.051141815860506486]  |
|9    |[91, 79, 88, 0, 35] |[0.07581414985273313, 0.075088527404026, 0.06429101969994398, 0.05773065894396137, 0.04755777123342685]     |
+-----+-------------------+----------------------------------------------------------------------------------------------------------------------------------------------------------------+
```

Figure 35: Corresponding term weight, topic name, and term indices

If you look at the preceding output carefully, we have found the corresponding term weight, topic name, and term indices. The preceding terms and their corresponding weights will be used in Chapter 9, *Advanced Machine Learning with Streaming and Graph Data*, for finding connected components using GraphX with Scala.

However, we also need to have the actual terms. We will show the detailed technique of retrieving the terms in Chapter 9, *Advanced Machine Learning with Streaming and Graph Data* hat extensively depends on the vocabulary of the terms that need to be developed or generated using the terms from the bag of words concept.

Scalability

The previous example shows how to perform topic modeling using the LDA algorithm as a standalone application. However, according to a Databricks blog by Joseph B. at https://databricks.com/blog/2015/03/25/topic-modeling-with-lda-mllib-meets-graphx.html, the parallelization of LDA is not straightforward, and there have been many research papers proposing different strategies. The key obstacle in this regard is that all methods involve a large amount of communication. According to the blog on the Databricks website, here are the statistics of the dataset and related training and test sets that were used during the experimentation:

- Training set size: 4.6 million documents
- Vocabulary size: 1.1 million terms
- Training set size: 1.1 billion tokens (~239 words/document)
- 100 topics
- 16-worker EC2 cluster, for example, M4.large or M3.medium depending upon budget and requirements
- Timing results: 176 secs/iteration on average over 10 iterations

Credit risk analysis pipeline with Spark

In this section, we will develop a credit risk pipeline that is commonly used in financial institutions such as banks and credit unions. First we will discuss what credit risk analysis is and why it is important before developing a Spark ML-based pipeline using a Random-Forest-based classifier. Finally, we will provide some performance improvement suggestions.

What is credit risk analysis? Why is it important?

When an applicant applies for a loan and a bank receives that application, based on the applicant's profile, the bank has to make a decision whether to approve the loan application or not.

In this regard, there are two types of risk associated with the bank's decision on the loan application:

- **Applicant is a good credit risk**: That means the client or applicant is more likely to repay the loan. Then, if the loan is not approved, the bank can potentially suffer loss of business.
- **Applicant is a bad credit risk**: That means that the client or applicant is most likely not to repay the loan. In that case, approving the loan to the client will result in financial loss to the bank.

Our common sense says that the second risk is the greater risk, as the bank has a higher chance of not being reimbursed the borrowed amount.

Therefore, most banks or credit unions evaluate the risks associated with lending money to a client, applicant, or customer. In business analytics, minimizing the risk tends to maximize the profit to the bank itself. In other words, maximizing the profit and minimizing the loss from a financial perspective is important.

Often, the bank makes a decision about approving a loan application based on different factor and parameters of an applicant. For example, the demographic and socio-economic conditions regarding their loan application.

Developing a credit risk analysis pipeline with Spark ML

In this section, we will first discuss the credit risk dataset in detail in order to gain some insight. After that, we will look at how to develop a large-scale credit risk pipeline. Finally, we will provide some performance improvement suggestions toward better prediction accuracy.

The dataset exploration

The German Credit dataset was downloaded from the UCI Machine Learning Repository at https://archive.ics.uci.edu/ml/machine-learning-databases/statlog/german/. Although a detailed description of the dataset is available in the link, we provide some brief insights here in *Table 3*. The data contains credit-related data on 21 variables and the classification of whether an applicant is considered a good or a bad credit risk for 1000 loan applicants. *Table 3* shows details about each variable that was considered before making the dataset available online:

Entry	Variable	Explanation
1	`creditability`	Capable of repaying
2	`balance`	Current balance
3	`duration`	Duration of the loan being applied for
4	`history`	Is there any bad loan history?
5	`purpose`	Purpose of the loan
6	`amount`	Amount being applied for
7	`savings`	Monthly saving
8	`employment`	Employment status

9	instPercent	Interest percent
10	sexMarried	Sex and marriage status
11	guarantors	Are there any guarantors?
12	residenceDuration	Duration of residence at the current address
13	assets	Net assets
14	age	Age of the applicant
15	concCredit	Concurrent credit
16	apartment	Residential status
17	credits	Current credits
18	occupation	Occupation
19	dependents	Number of dependents
20	hasPhone	If the applicant uses a phone
21	foreign	If the applicant is a foreigner

Table 3: German credit dataset properties

Note that, although *Table 3* describes the variables in the dataset, there is no associated header. In *Table 3*, we have shown the variable, position, and associated significance of each variable.

Credit risk pipeline with Spark ML

There will be several steps involved, from data loading, parsing, data preparation, training testing set preparation, model training, model evaluation, and result interpretation. Let's go through the steps one by one.

Step 1: Load required APIs and libraries

The following is the code for loading the required APIs and libraries:

```
import org.apache.spark.api.java.JavaRDD;
import
org.apache.spark.api.java.function.Function;
import
org.apache.spark.ml.classification.RandomForestClassificationModel;
import
org.apache.spark.ml.classification.RandomForestClassifier;
import
org.apache.spark.ml.evaluation.BinaryClassificationEvaluator;
import org.apache.spark.ml.feature.StringIndexer;
import
org.apache.spark.ml.feature.VectorAssembler;
import
org.apache.spark.mllib.evaluation.RegressionMetrics;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;
```

Step 2: Create a Spark session

The following is another code for creating a Spark session:

```
  static SparkSession spark =
SparkSession.builder()
      .appName("CreditRiskAnalysis")
      .master("local[*]")
      .config("spark.sql.warehouse.dir",
"E:/Exp/")
      .getOrCreate();
```

Step 3: Load and parse the credit risk dataset

Note that the dataset is in **Comma-Separated Value** (**CSV**) format. Now load and parse the dataset using the Databricks-provided CSV readers and prepare a Dataset of Row, as follows:

```
String csvFile = "input/german_credit.data";
Dataset<Row> df =
spark.read().format("com.databricks.spark.csv").option("header",
"false").load(csvFile);
```

Now, show the Dataset to get to know the exact structure, as follows:

```
df.show();
```

```
+---+---+---+---+---+----+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|_c0|_c1|_c2|_c3|_c4|_c5 |_c6|_c7|_c8|_c9|_c10|_c11|_c12|_c13|_c14|_c15|_c16|_c17|_c18|_c19|_c20|
+---+---+---+---+---+----+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|1  |1  |18 |4  |2  |1049|1  |2  |4  |2  |1  |4  |2  |21 |3  |1  |1  |3  |1  |1  |1  |
|1  |1  |9  |4  |0  |2799|1  |3  |2  |3  |1  |2  |1  |36 |3  |1  |2  |3  |2  |1  |1  |
|1  |2  |12 |2  |9  |841 |2  |4  |2  |2  |1  |4  |1  |23 |3  |1  |1  |2  |1  |1  |1  |
|1  |1  |12 |4  |0  |2122|1  |3  |3  |3  |1  |2  |1  |39 |3  |1  |2  |2  |2  |1  |2  |
|1  |1  |12 |4  |0  |2171|1  |3  |4  |3  |1  |4  |2  |38 |1  |2  |2  |2  |1  |1  |2  |
|1  |1  |10 |4  |0  |2241|1  |2  |1  |3  |1  |3  |1  |48 |3  |1  |2  |2  |2  |1  |2  |
|1  |1  |8  |4  |0  |3398|1  |4  |1  |3  |1  |4  |1  |39 |3  |2  |2  |2  |1  |1  |2  |
|1  |1  |6  |4  |0  |1361|1  |2  |2  |3  |1  |4  |1  |40 |3  |2  |1  |2  |2  |1  |2  |
|1  |4  |18 |4  |3  |1098|1  |1  |4  |2  |1  |4  |3  |65 |3  |2  |2  |1  |1  |1  |1  |
|1  |2  |24 |2  |3  |3758|3  |1  |1  |2  |1  |4  |4  |23 |3  |1  |1  |1  |1  |1  |1  |
|1  |1  |11 |4  |0  |3905|1  |3  |2  |3  |1  |2  |1  |36 |3  |1  |2  |3  |2  |1  |1  |
|1  |1  |30 |4  |1  |6187|2  |4  |1  |4  |1  |4  |3  |24 |3  |1  |2  |3  |1  |1  |1  |
|1  |1  |6  |4  |3  |1957|1  |4  |1  |2  |1  |4  |3  |31 |3  |2  |1  |3  |1  |1  |1  |
|1  |2  |48 |3  |10 |7582|2  |1  |2  |3  |1  |4  |4  |31 |3  |2  |1  |4  |1  |2  |1  |
|1  |1  |18 |2  |3  |1936|5  |4  |2  |4  |1  |4  |3  |23 |3  |1  |2  |2  |1  |1  |1  |
|1  |1  |6  |2  |3  |2647|3  |3  |2  |3  |1  |3  |1  |44 |3  |1  |1  |3  |2  |1  |1  |
|1  |1  |11 |4  |0  |3939|1  |3  |1  |3  |1  |2  |1  |40 |3  |2  |2  |2  |2  |1  |1  |
|1  |2  |18 |2  |3  |3213|3  |2  |1  |4  |1  |3  |1  |25 |3  |1  |1  |3  |1  |1  |1  |
|1  |2  |36 |4  |3  |2337|1  |5  |4  |3  |1  |4  |1  |36 |3  |2  |1  |3  |1  |1  |1  |
|1  |4  |11 |4  |0  |7228|1  |3  |1  |3  |1  |4  |2  |39 |3  |2  |2  |2  |1  |1  |1  |
+---+---+---+---+---+----+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
only showing top 20 rows
```

Figure 36: A snapshot of the credit risk dataset

Step 4: Create an RDD of type Credit

Create an RDD of typed class `Credit`, as follows:

```
JavaRDD<Credit> creditRDD =
df.toJavaRDD().map(new Function<Row, Credit>() {
      @Override
      public Credit call(Row r) throws Exception
{
        return new
Credit(parseDouble(r.getString(0)),
parseDouble(r.getString(1)) - 1,
            parseDouble(r.getString(2)),
parseDouble(r.getString(3)),
parseDouble(r.getString(4)),
            parseDouble(r.getString(5)),
parseDouble(r.getString(6)) - 1,
parseDouble(r.getString(7)) - 1,
```

```
            parseDouble(r.getString(8)),
        parseDouble(r.getString(9)) - 1,
        parseDouble(r.getString(10)) - 1,
            parseDouble(r.getString(11)) - 1,
        parseDouble(r.getString(12)) - 1,
            parseDouble(r.getString(13)),
        parseDouble(r.getString(14)) - 1,
            parseDouble(r.getString(15)) - 1,
        parseDouble(r.getString(16)) - 1,
            parseDouble(r.getString(17)) - 1,
        parseDouble(r.getString(18)) - 1,
            parseDouble(r.getString(19)) - 1,
        parseDouble(r.getString(20)) - 1);
          }
        });
```

The preceding code segments creates an RDD of type `Credit` after taking the variable as double values by using the `parseDouble()` method, which takes a string and returns the corresponding value in `Double` format. The `parseDouble()` method goes as follows:

```
public static double parseDouble(String str) {
  return Double.parseDouble(str);
}
```

Now we need to know the structure of the `Credit` class so that the structure itself helps to create the RDDs using the typed class.

Well, the `Credit` class is basically a singleton class that initializes all the setter and getter methods for the 21 variables from the dataset through the constructor. Here is the class:

```
public class Credit {
  private double creditability;
  private double balance;
  private double duration;
  private double history;
  private double purpose;
  private double amount;
  private double savings;
  private double employment;
  private double instPercent;
  private double sexMarried;
  private double guarantors;
  private double residenceDuration;
  private double assets;
  private double age;
  private double concCredit;
  private double apartment;
  private double credits;
  private double occupation;
  private double dependents;
```

```
  private double hasPhone;
  private double foreign;

  public Credit(double creditability, double
balance, double duration,
  double history, double purpose, double amount,
      double savings, double employment, double
instPercent,
      double sexMarried, double guarantors,
      double residenceDuration, double assets,
double age,
      double concCredit, double apartment, double
credits,
      double occupation, double dependents,
double hasPhone, double foreign) {
    super();
    this.creditability = creditability;
    this.balance = balance;
    this.duration = duration;
    this.history = history;
    this.purpose = purpose;
    this.amount = amount;
    this.savings = savings;
    this.employment = employment;
    this.instPercent = instPercent;
    this.sexMarried = sexMarried;
    this.guarantors = guarantors;
    this.residenceDuration = residenceDuration;
    this.assets = assets;
    this.age = age;
    this.concCredit = concCredit;
    this.apartment = apartment;
    this.credits = credits;
    this.occupation = occupation;
    this.dependents = dependents;
    this.hasPhone = hasPhone;
    this.foreign = foreign;
  }

  public double getCreditability() {
    return creditability;
  }

  public void setCreditability(double
creditability) {
    this.creditability = creditability;
  }

  public double getBalance() {
    return balance;
  }

  public void setBalance(double balance) {
    this.balance = balance;
  }
```

```
  public double getDuration() {
    return duration;
  }

  public void setDuration(double duration) {
    this.duration = duration;
  }

  public double getHistory() {
    return history;
  }

  public void setHistory(double history) {
    this.history = history;
  }

  public double getPurpose() {
    return purpose;
  }

  public void setPurpose(double purpose) {
    this.purpose = purpose;
  }

  public double getAmount() {
    return amount;
  }

  public void setAmount(double amount) {
    this.amount = amount;
  }

  public double getSavings() {
    return savings;
  }

  public void setSavings(double savings) {
    this.savings = savings;
  }

  public double getEmployment() {
    return employment;
  }

  public void setEmployment(double employment) {
    this.employment = employment;
  }

  public double getInstPercent() {
    return instPercent;
  }

  public void setInstPercent(double instPercent)
{
```

```
    this.instPercent = instPercent;
  }

  public double getSexMarried() {
    return sexMarried;
  }

  public void setSexMarried(double sexMarried) {
    this.sexMarried = sexMarried;
  }

  public double getGuarantors() {
    return guarantors;
  }

  public void setGuarantors(double guarantors) {
    this.guarantors = guarantors;
  }

  public double getResidenceDuration() {
    return residenceDuration;
  }

  public void setResidenceDuration(double
residenceDuration) {
    this.residenceDuration = residenceDuration;
  }

  public double getAssets() {
    return assets;
  }

  public void setAssets(double assets) {
    this.assets = assets;
  }

  public double getAge() {
    return age;
  }

  public void setAge(double age) {
    this.age = age;
  }

  public double getConcCredit() {
    return concCredit;
  }

  public void setConcCredit(double concCredit) {
    this.concCredit = concCredit;
  }

  public double getApartment() {
    return apartment;
  }
```

```
    public void setApartment(double apartment) {
      this.apartment = apartment;
    }

    public double getCredits() {
      return credits;
    }

    public void setCredits(double credits) {
      this.credits = credits;
    }

    public double getOccupation() {
      return occupation;
    }

    public void setOccupation(double occupation) {
      this.occupation = occupation;
    }

    public double getDependents() {
      return dependents;
    }

    public void setDependents(double dependents) {
      this.dependents = dependents;
    }

    public double getHasPhone() {
      return hasPhone;
    }

    public void setHasPhone(double hasPhone) {
      this.hasPhone = hasPhone;
    }

    public double getForeign() {
      return foreign;
    }

    public void setForeign(double foreign) {
      this.foreign = foreign;
    }
  }
```

If you look at the flow of the class, at first it declares 21 variables for the 21 features in the dataset. Then it initializes them using the constructor. The rest are simple setter and getter methods.

Step 5: Create a Dataset of type Row from the RDD of type Credit

The following code shows how to create a Dataset of type Row:

```
Dataset<Row> creditData =
spark.sqlContext().createDataFrame(creditRDD,
Credit.class);
```

Now save the Dataset as a temporary view, or more formally, a table in-memory for query purposes, as follows:

```
creditData.createOrReplaceTempView("credit");
```

Now let's get to know the schema of the table as follows:

```
creditData.printSchema();
```

```
root
 |-- age: double (nullable = false)
 |-- amount: double (nullable = false)
 |-- apartment: double (nullable = false)
 |-- assets: double (nullable = false)
 |-- balance: double (nullable = false)
 |-- concCredit: double (nullable = false)
 |-- creditability: double (nullable = false)
 |-- credits: double (nullable = false)
 |-- dependents: double (nullable = false)
 |-- duration: double (nullable = false)
 |-- employment: double (nullable = false)
 |-- foreign: double (nullable = false)
 |-- guarantors: double (nullable = false)
 |-- hasPhone: double (nullable = false)
 |-- history: double (nullable = false)
 |-- instPercent: double (nullable = false)
 |-- occupation: double (nullable = false)
 |-- purpose: double (nullable = false)
 |-- residenceDuration: double (nullable = false)
 |-- savings: double (nullable = false)
 |-- sexMarried: double (nullable = false)
```

Figure 37: The schema of the Dataset

Step 6: Create the feature vector using the VectorAssembler

Create a new feature vector for the 21 variables using the `VectorAssembler` class of Spark, as follows:

```
VectorAssembler assembler = new VectorAssembler()
        .setInputCols(new String[] { "balance",
"duration", "history", "purpose", "amount",
"savings",
            "employment", "instPercent",
"sexMarried", "guarantors", "residenceDuration",
"assets", "age",
            "concCredit", "apartment", "credits",
"occupation", "dependents", "hasPhone",
"foreign" })
```

```
        .setOutputCol("features");
```

Step 7: Create a Dataset by combining and transforming the assembler

Create a Dataset by transforming the assembler using the `creditData` Dataset previously created, and print the first top 20 rows of the Dataset, as follows:

```
Dataset<Row> assembledFeatures =
assembler.transform(creditData);
assembledFeatures.show();
```

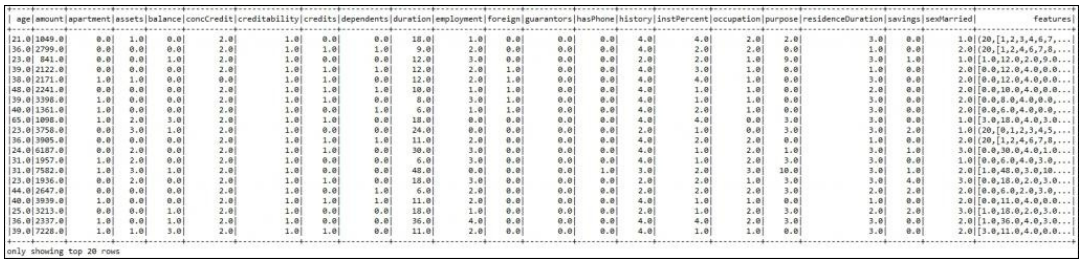


Figure 38: Newly created featured Credit Dataset

Step 8: Create label for making predictions

Create a label column out of the creditability column of the preceding Dataset (*Figure 38*), as follows:

```
StringIndexer creditabilityIndexer = new
StringIndexer().setInputCol("creditability").setOutputCol("label");
Dataset<Row> creditabilityIndexed =
creditabilityIndexer.fit(assembledFeatures).transform(assembledFeatures);
```

Now let's explore the new Dataset using the `show()` method as follows:

```
creditabilityIndexed.show();
```

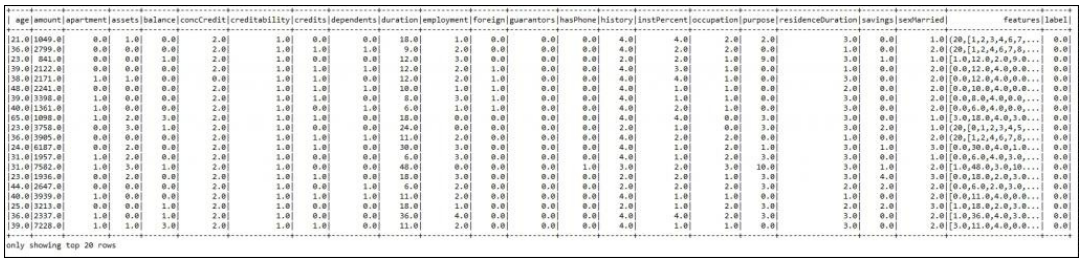


Figure 39: Dataset with a new label column

From the preceding figure, we can understand that there are only two labels associated with the Dataset, which are 1.0 and 0.0. That signifies the problem as a binary classification problem.

Step 9: Prepare the training and test set

Prepare the training and test set as follows:

```
long splitSeed = 12345L;
Dataset<Row>[] splits =
creditabilityIndexed.randomSplit(new double[]
{ 0.7, 0.3 }, splitSeed);
Dataset<Row> trainingData = splits[0];
Dataset<Row> testData = splits[1];
```

Here, the ratio is 70% and 30% for the training and testing set, respectively, with a long seed value to disallow the random result generation in each iteration.

Step 10: Train the Random Forest model

To train the Random Forest model, use the following code:

```
RandomForestClassifier classifier = new
RandomForestClassifier()
        .setImpurity("gini")
        .setMaxDepth(3)
        .setNumTrees(20)
        .setFeatureSubsetStrategy("auto")
        .setSeed(splitSeed);
```

As previously mentioned, the problem is a binary classification problem. Therefore, we will evaluate the Random Forest model using a binary evaluator for the `label` column, as follows:

```
RandomForestClassificationModel model =
classifier.fit(trainingData);
BinaryClassificationEvaluator evaluator = new
BinaryClassificationEvaluator().setLabelCol("label");
```

Now we need to collect the model performance metric on the test set that goes as follows:

```
Dataset<Row> predictions =
model.transform(testData);
model.toDebugString();
```

Step 11: Print the performance parameters

We will observe several performance parameters of the binary evaluator, for example, accuracy after fitting the model, **Mean Square Error** (**MSE**), **Mean Absolutize Error** (**MAE**), **Root Mean Squared Error** (**RMSE**), R Squared and explained variable, and so on. Let's do it as follows:

```
double accuracy =
evaluator.evaluate(predictions);
System.out.println("Accuracy after pipeline
fitting: " + accuracy);
RegressionMetrics rm = new
RegressionMetrics(predictions);
System.out.println("MSE: " +
rm.meanSquaredError());
System.out.println("MAE: " +
rm.meanAbsoluteError());
System.out.println("RMSE Squared: " +
rm.rootMeanSquaredError());
System.out.println("R Squared: " + rm.r2());
System.out.println("Explained Variance: " +
rm.explainedVariance() + "\n");
```

The preceding code segment generates the following output:

```
Accuracy after pipeline fitting:
0.7622000403307129
MSE: 1.926235109206349E7
MAE: 3338.3492063492063
RMSE Squared: 4388.8895055655585
R Squared: -1.372326447615067
Explained Variance: 1.1144695981899707E7
```

Performance tuning and suggestions

If you look at the performance metrics in Step 11, it is obvious that the credit risk predictions are not satisfactory, especially in terms of accuracy, which is only 76.22%. That means that for the given test data, our model can predict if there is a credit risk with 76.22% precision. Since we need to be more careful about such sensitive financial sectors, therefore, more accuracy is desired no doubt.

Now, if you want to increase the prediction performance, you should try training your model using a model other than the Random-Forest-based classifier. For example, a Logistic Regression or NaÃ¯ve Baseyan-based classifier.

Moreover, you can use the SVM-based classifier or neural-network-based Multilayer Perceptron classifier. In Chapter 7, *Tuning Machine Learning*

Models, we will look at how to tune the hyper parameters in order to select the best model.

Scaling the ML pipelines

Data mining and machine learning algorithms impose outstanding challenges on parallel and distributed computing platforms. Furthermore, parallelizing the machine learning algorithms is highly task-specific and often depends on the preceding questions. In Chapter 1, *Introduction to Data Analytics with Spark*, we discussed and showed how to deploy the same machine learning application on top of a cluster or cloud computing infrastructure (that is, Amazon AWS/EC2).

Following that method, we can handle datasets with enormous batch sizes or in real time. In addition to this, scaling up the machine learning applications evolves another trade-off such as cost, complexity, run-time, and technical requirements. Furthermore, making task-appropriate algorithm and platform choices for large-scale machine learning requires an understanding of the benefits, trade-offs, and constraints of the available options.

To handle these issues, in this section, we will provide some theoretical aspects of handling big Datasets for deploying large-scale machine learning applications. However, before going any further, we need to know the answer to some questions. For example:

- How do we collect the big dataset to fulfil our needs?
- How large are the big datasets and how do we handle them?
- How much training data is enough to scale-up the ML application on a big dataset?
- What is an alternative approach if we don't have enough training data?
- What sorts of machine learning algorithms should be used for fulfilling our needs?
- What platform should be chosen for parallel learning?

Here we discuss some important aspects of deploying and scaling up a machine learning application that handles the preceding big data challenges, including size, data skewness, cost, and infrastructure.

Size matters

Big data is data in volume, variety, veracity, velocity, and value that is too great to process by traditional in-memory computer systems. Scaling up machine learning applications by handling big data involves tasks such as classification, clustering, regression, feature selection, boosted decision trees, and SVMs. How do we handle 1 billion or 1 trillion data instances? Moreover, 5 billion cell phones, social networks such as Twitter produce big

datasets in an unprecedented way. On the other hand, crowdsourcing is the reality, which is labeling of 100,000+ data instances within a week.

In terms of sparsity, Big datasets cannot be too sparse but dense from a content perspective. From the machine learning perspective, to justify this claim, let's think of an example of data labeling. For instance, 1M data instances cannot belong to 1M classes, simply because it's not practical to have 1M classes but more than once data instances belong to a particular class. Therefore, based on the sparsity and size of such a large-scale dataset, making predictive analytics is another challenge, which needs to be considered and handled while scaling up.

Size versus skewness considerations

Machine learning also depends on the availability of labeled data, and its trustworthiness is based on the learning task, such as supervised, unsupervised, or semi-supervised. You might have a structured dataset, but with extreme skewness. More specifically, suppose you have 1K labeled and 1M unlabeled data points, so the labeled and unlabeled ratio is 0.1%.

Therefore, do you think that only the 1K label points are enough to train a supervised model? As another example, suppose, for instance, that you have 1M labeled and 1B unlabeled data points, where the labeled and unlabeled ratio is also 0.1%. Again, the same question arises, which is, is it enough to have only the 1M labels to train a supervised model?

Now the concern is what can be done or approach using existing labels as only guidance rather than a directive for the semi-supervised clustering, classification, or regression. Alternatively, label more data, either manually, or with a little help from the crowd. For example, suppose someone wants to cluster or classify analysis on a disease. More specifically, suppose we want to classify tweets, if particular tweets indicate an Ebola- or flu-related disease. In this case, we should use the semi-supervised approach for labeling the tweets.

However, in this case, a dataset might be very skewed, or labeling might be biased. Usually, the training data comes from different users, where an explicit user feedback might often be misleading.

Therefore, learning from the implicit feedback is a better idea; for example, collecting data by clicking on web search results. In these types of large-scale datasets, the skewness of the training data is hard to detect, as discussed in Chapter 4, *Extracting Knowledge through Feature Engineering*. Therefore, be wary of this skewness in big Datasets.

Cost and infrastructure

To scale-up your machine learning application, you will need better infrastructure and computing power to handle such big datasets. Initially, you might want to utilize a local cluster. However, sometimes, the cluster might not be enough to scale-up your ML application if the dataset increases exponentially.

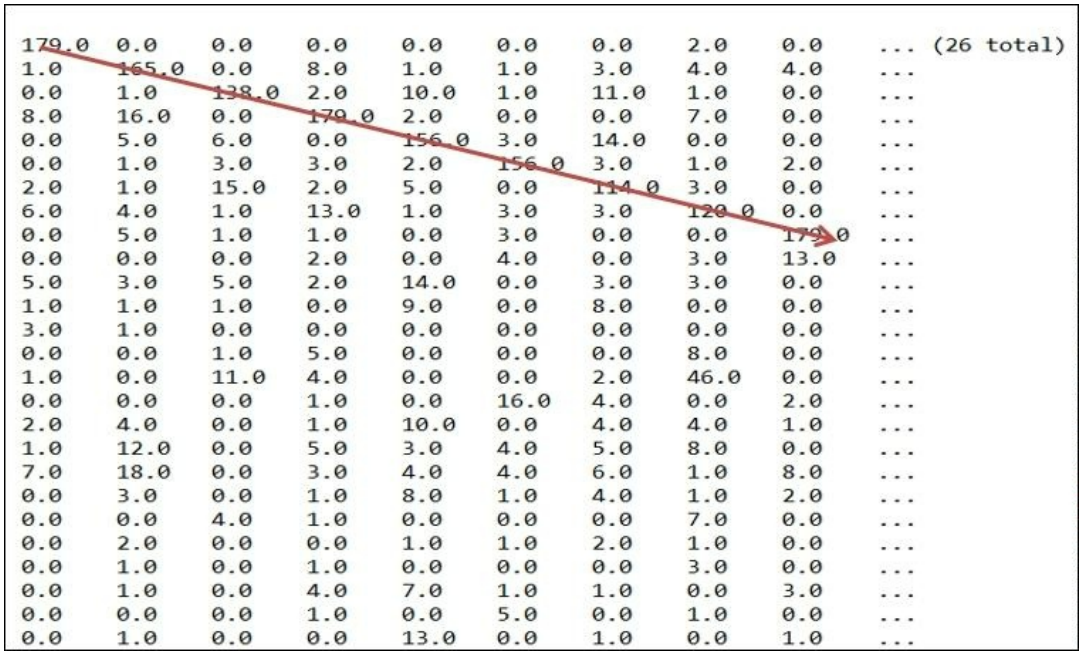
As discussed in the chapter about deploying the ML pipeline on powerful infrastructure such as Amazon AWS cloud computing like EC2, you will have to go for pay-as-you-go to enjoy the cloud as Platform as a Service and Infrastructure as a Service, even though you use your own ML application as the Software as a Service.

Tips and performance considerations

Spark also supports cross-validation for hyperparameter tuning that will be discussed broadly in the following chapter. Spark's view toward the cross-validation as a meta-algorithm, which fits the underlying estimator with user-specified combinations of parameters, cross-evaluates the fitted models and outputs the best one.

However, there is no specific requirement of the underlying estimator, which could be a pipeline, as long as it can be paired with an evaluator that outputs a scalar metric, such as precision and recall, from the predictions.

Let's recall the OCR prediction, where we found that the precision is 75%, which is obviously not satisfactory, once again. To further investigate the reason now, let's prints the confusion matrix for the label 8.0 or "I". If you look at the following matrix in *Figure 40*, you will find the number of correctly predicted instances is low:



179.0	0.0	0.0	0.0	0.0	0.0	0.0	2.0	0.0	... (26 total)
1.0	165.0	0.0	8.0	1.0	1.0	3.0	4.0	4.0	...
0.0	1.0	138.0	2.0	10.0	1.0	11.0	1.0	0.0	...
8.0	16.0	0.0	170.0	2.0	0.0	0.0	7.0	0.0	...
0.0	5.0	6.0	0.0	156.0	3.0	14.0	0.0	0.0	...
0.0	1.0	3.0	3.0	2.0	156.0	3.0	1.0	2.0	...
2.0	1.0	15.0	2.0	5.0	0.0	114.0	3.0	0.0	...
6.0	4.0	1.0	13.0	1.0	3.0	3.0	120.0	0.0	...
0.0	5.0	1.0	1.0	0.0	3.0	0.0	0.0	179.0	...
0.0	0.0	0.0	2.0	0.0	4.0	0.0	3.0	13.0	...
5.0	3.0	5.0	2.0	14.0	0.0	3.0	3.0	0.0	...
1.0	1.0	1.0	0.0	9.0	0.0	8.0	0.0	0.0	...
3.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
0.0	0.0	1.0	5.0	0.0	0.0	0.0	8.0	0.0	...
1.0	0.0	11.0	4.0	0.0	0.0	2.0	46.0	0.0	...
0.0	0.0	0.0	1.0	0.0	16.0	4.0	0.0	2.0	...
2.0	4.0	0.0	1.0	10.0	0.0	4.0	4.0	1.0	...
1.0	12.0	0.0	5.0	3.0	4.0	5.0	8.0	0.0	...
7.0	18.0	0.0	3.0	4.0	4.0	6.0	1.0	8.0	...
0.0	3.0	0.0	1.0	8.0	1.0	4.0	1.0	2.0	...
0.0	0.0	4.0	1.0	0.0	0.0	0.0	7.0	0.0	...
0.0	2.0	0.0	0.0	1.0	1.0	2.0	1.0	0.0	...
0.0	1.0	0.0	1.0	0.0	0.0	0.0	3.0	0.0	...
0.0	1.0	0.0	4.0	7.0	1.0	1.0	0.0	3.0	...
0.0	0.0	0.0	1.0	0.0	5.0	0.0	1.0	0.0	...
0.0	1.0	0.0	0.0	13.0	0.0	1.0	0.0	1.0	...

Figure 40: The confusion matrix for the label 8.0 or "I"

Now let's try to use the Random Forest model to do the prediction. But before going into the model training step, let's do some initialization of the parameters that are needed for the Random Forest classifier, which also supports multiclass classification such as the logistic regression model with LBFGS:

```
Integer numClasses = 26;
```

```
HashMap<Integer, Integer>categoricalFeaturesInfo
= new HashMap<Integer, Integer>();
Integer numTrees = 5; // Use more in practice.
String featureSubsetStrategy = "auto"; // Let the
algorithm choose.
String impurity = "gini";
Integer maxDepth = 20;
Integer maxBins = 40;
Integer seed = 12345;
```

Now train the model by specifying the previous parameters, as follows:

```
final RandomForestModelmodel =
RandomForest.trainClassifier(training,
numClasses, categoricalFeaturesInfo, numTrees,
featureSubsetStrategy, impurity, maxDepth,
maxBins, seed);
```

Now let's see how it performs. We'll reuse the same code segments that we used in step 9. Refer to the following screenshot:

```
16/06/30 23:39:07 INFO DAGScheduler: ResultStage 53 (collectAsMap at MulticlassMetrics.scala:54) finished in 0.003 s
16/06/30 23:39:07 INFO DAGScheduler: Job 28 finished: collectAsMap at MulticlassMetrics.scala:54, took 0.229544 s
Precision = 0.8920171673819742
Recall = 0.8920171673819742
True Positive Rate = 0.9292452830188679
False Positive Rate = 0.003206841261357563
Weighted True Positive Rate = 0.8920171673819741
Weighted False Positive Rate = 0.0042955330254867636
16/06/30 23:39:07 INFO SparkContext: Invoking stop() from shutdown hook
16/06/30 23:39:07 INFO SparkUI: Stopped Spark web UI at http://192.168.42.251:4040
```

Figure 41: Performance metrics for the precision and recall

```
210.0  0.0    1.0    1.0    0.0    1.0    0.0    1.0    0.0    ... (26 total)
0.0    189.0  0.0    3.0    1.0    1.0    4.0    3.0    0.0    ...
0.0    0.0    189.0  0.0    2.0    0.0    0.0    1.0    0.0    ...
0.0    8.0    0.0    206.0  1.0    1.0    1.0    2.0    0.0    ...
0.0    1.0    1.0    0.0    199.0  0.0    3.0    0.0    1.0    ...
0.0    1.0    0.0    1.0    0.0    179.0  0.0    0.0    3.0    ...
1.0    1.0    1.0    1.0    2.0    2.0    180.0  1.0    1.0    ...
0.0    4.0    1.0    3.0    2.0    1.0    2.0    171.0  1.0    ...
0.0    3.0    0.0    1.0    0.0    1.0    0.0    1.0    197.0  ...
0.0    1.0    0.0    3.0    0.0    2.0    0.0    1.0    7.0    ...
0.0    1.0    1.0    1.0    9.0    0.0    1.0    5.0    1.0    ...
0.0    0.0    4.0    0.0    0.0    0.0    4.0    0.0    1.0    ...
0.0    1.0    1.0    2.0    1.0    0.0    1.0    0.0    0.0    ...
0.0    1.0    0.0    3.0    0.0    1.0    0.0    3.0    0.0    ...
0.0    1.0    4.0    2.0    0.0    0.0    0.0    0.0    0.0    ...
0.0    2.0    1.0    0.0    3.0    15.0   2.0    2.0    0.0    ...
1.0    2.0    0.0    0.0    1.0    0.0    2.0    0.0    0.0    ...
0.0    8.0    1.0    1.0    0.0    1.0    1.0    0.0    1.0    ...
0.0    7.0    2.0    0.0    6.0    0.0    3.0    2.0    1.0    ...
0.0    0.0    2.0    0.0    3.0    0.0    0.0    0.0    0.0    ...
0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    ...
1.0    7.0    1.0    0.0    0.0    4.0    0.0    1.0    1.0    ...
0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    ...
0.0    1.0    1.0    1.0    1.0    0.0    0.0    1.0    0.0    ...
0.0    0.0    1.0    0.0    0.0    1.0    0.0    0.0    0.0    ...
0.0    1.0    0.0    0.0    2.0    0.0    0.0    0.0    0.0    ...
```

Figure 42: The improved confusion matrix for the label 8.0 or "I"

If you look at *Figure 42*, we have a significant improvement in terms of all the parameters printed, and the precision has been increased from 75.30% to 89.20%. The reason behind this is the improved interpretation of the Random Forest model toward the global maxima calculation for prediction accuracy and the confusion matrix, as shown in *Figure 38*; you will find significant improvements in the number of predicted instances marked by a diagonal arrow.

Through a process of trial and error, you can settle on a short list of algorithms that show promise, but how do you know which is the best? Moreover, as previously mentioned, it is difficult to find a well-performing machine learning algorithm for your dataset. Therefore, if you are still unsatisfied with the accuracy of 89.20%, I suggest you tune the parametric values and look at the precision and the recall.

Summary

In this chapter, we have shown several machine learning applications and tried to differentiate between Spark MLlib and Spark ML. We also showed that it's really difficult to develop a complete machine learning application using only Spark ML or Spark MLlib.

However, we would like to argue that a combined approach, or interoperability, between these two APIs would be best for these purposes. In addition, we learned how to build an ML pipeline by using both Spark ML libraries and how to scale-up the basic model by considering some performance considerations.

Tuning an algorithm or machine learning application can simply be thought of as a process by which one goes through to optimize the parameters that impact the model in order to enable the algorithm to perform at its best (in terms of runtime and memory usage).

In Chapter 7, *Tuning Machine Learning Models*, we will discuss more about tuning machine learning models. We will try to reuse some of the applications from this chapter and Chapter 5, *Supervised and Unsupervised Learning by Examples*, to tune up the performance by tuning several parameters.

Chapter 7. Tuning Machine Learning Models

Tuning an algorithm or a machine learning application is simply a process that one goes through in order to enable the algorithm to perform optimally (in terms of runtime and memory usage) when optimizing the parameters that impact the model. This chapter aims to guide the reader through model tuning. It will cover the main techniques used to optimize an ML algorithm's performance. Techniques will be explained both from the MLlib and Spark ML perspective. In Chapter 5, *Supervised and Unsupervised Learning by Examples* and Chapter 6, *Building Scalable Machine Learning Pipelines*, we described how to develop some complete machine learning applications and pipelines from data collection to model evaluation. In this chapter, we will try to reuse some of those applications to improve the performance by tuning several parameters such as Hyperparameter tuning, Grid search parameter tuning with MLlib and Spark ML, random search parameter tuning and cross-validation. The hypothesis ,which is also an important statistical test, will be discussed. In summary, the following topics will be covered in this chapter:

- Details about machine learning model tuning
- Typical challenges in model tuning
- Evaluating machine learning models
- Validation and evaluation techniques
- Parameter tuning for machine learning
- Hypothesis testing
- Machine learning model selection

Details about machine learning model tuning

Fundamentally, one can argue that the ultimate goal of **machine learning** (**ML**) is to make a machine that can automatically build models from data without requiring tedious and time-consuming human involvement. You will see that one of the difficulties in ML is that learning algorithms are like decision trees, random forests, and clustering techniques that require you to set parameters before you use the models for practical purposes. Alternatively, you need to set some constraints on those parameters.

How you set those parameters depends on a set of factors and the specification. Your goal in this regard is usually to set those parameters to the optimal values that enable you to complete a learning task in the best possible way. Thus, tuning an algorithm or ML technique can be simply thought of as a process where one goes through a series of steps in which

they optimize the parameters that impact the model's performance in order to enable the algorithm to perform in the best way.

In Chapter 3, *Understanding the Problem by Understanding the Data*, and Chapter 5, *Supervised and Unsupervised Learning by Examples*, we discussed some techniques to choose the best algorithm based on your data and discuss the most widely used algorithms. To get the best result out of your model, you have to first define what the *best* actually is. We will discuss tuning in both abstract and concrete ways.

In the abstract sense of machine learning, tuning involves working with variables or based on parameters that have been identified to affect system performance as evaluated by some appropriate metric. Hence, the improved performance reveals which parameter settings are more favorable (that is, tuned) or less favorable (that is, un-tuned). In common sense terms, tuning is essentially selecting the best parameters for an algorithm to optimize its performance, given the working environment of hardware, specific workloads, and so on And tuning in machine learning is an automated process for doing this.

Well, let's get to the point and make the discussion more concrete through some examples. If you take an ML algorithm for clustering like KNN or K-Means, as a developer/data scientist/data engineer you need to specify the number of K in your model or centroids.

So the question is 'how can you do this?' Technically, there is no shortcut around the need to tune the model. Computationally a naïve approach would be to try with different values of K as a model and of course observing how it goes to inter and intra-group error as you vary the number of K in your model.

The second example could be by using the **Support Vector Machine** (**SVM**) for classification tasks. As you know, an SVM classification requires an initial learning phase in which the training data are used to adjust the classification parameters. This really denotes an initial parameter-tuning phase where you might try to tune the models in order to achieve high-quality results.

The third practical example suggest that there is no such thing as a perfect set of optimizations for all deployments of an Apache web server. A sysadmin learns from the data on the job, so to speak, and optimizes it's own Apache web server configuration as appropriate for its specific environment. Now imagine an automated process for doing those three things, that is, a system that can learn from data on its own; the definition of machine learning. A system that tunes its own parameters in such a data-based fashion would be an instance of tuning in machine learning. Now, let us summarize the main points of why we evaluate the predictive performance of a model:

- We want to estimate the generalization error, the predictive performance of our model on future (unseen) data.
- We want to increase the predictive performance by tweaking the learning algorithm and selecting the best-performing model from a given hypothesis space.
- We want to identify the machine learning algorithm that is best-suited for the problem at hand; thus, we want to compare different algorithms, selecting the best-performing one as well as the best-performing model from the algorithm's hypothesis space.

In a nutshell, there are four steps in the process of finding the best parameter set:

- **Define the parametric space**: First, we need to decide the exact parameter values we would like to consider for the algorithm.
- **Define the cross-validation settings**: Secondly we need to decide how to choose the optimal cross-validation folds for the data (to be discussed later in this chapter).
- **Define the metric**: Thirdly, we need to decide which metric to use for determining the best combination of parameters. For example, accuracy, root means squared error, precision, recall, or f-score, and so on.
- **Train, evaluate and compare**: Fourthly, for each unique combination of the parameter values, cross-validation is carried out and based on the error metric defined by the user in the third step, the best-performing model can be chosen.

There are many techniques and algorithms available for model tunings like hyperparameter optimization or model selection, hyperparameter tuning, grid search parameter tuning, random search parameter tuning and **Cross Validation** (**CV**). Unfortunately, the current implementation of Spark has developed only a few of them including Cross Validator and TrainValidationSplit.

Therefore, we will try to use these two hyperparameters to tune different models including Random Forest, Liner Regression, and Logistic Regression. Some applications from Chapter 5, *Supervised and Unsupervised Learning by Examples* and Chapter 6, *Building Scalable Machine Learning Pipelines* will be re-used without providing many details again to make the model tuning easier.

Typical challenges in model tuning

After the following discussion, you might be thinking that this process is difficult, and you'd be right. In fact, because of the difficulty in determining what optimal model parameters are, often some more complex learning algorithms are used before experimenting effectively with simpler options with better-tuned parameters. As we've already discussed, machine learning involves a lot of experimentation. And the tuning of the internal knobs of a learning algorithm, commonly referred to as hyperparameters, are equally important from model building to prediction and before deployment.

Technically, running a learning algorithm over a training dataset with different hyperparameter settings will result in different models, and of course different performance parameters. According to Oracle developers, it is not recommended to begin tuning without having first established clear objectives, since you cannot succeed if there is any definition of success.

Subsequently, we are typically interested in selecting the best-performing model from the training data set; we need to find a way to estimate their respective performances in order to rank them against each other. Going one step beyond mere algorithm fine-tuning, we are usually not only experimenting with the one single algorithm that we think would be the best solution under the given circumstances. More often than not, we want to compare different algorithms to each other, often in terms of predictive and computational performance.

Often there is a very basic question regarding parameter tuning using grid search and random search. Typically, some machine learning methods have parameters that need to be tuned using either of them. For example, according to Wei Ch. et al., (*A General Formulation for Support Vector Machine, proceedings of the 9th International Conference on Neural Information Processing (ICONIP, 02), V-5, 18-22 Nov. 2002*) the standard formulation of SVMs is as follows:

$$\mathcal{R}(\mathbf{w}, b) = C \cdot \sum_{i=1}^{n} \ell\big(y_i, f(x_i)\big) + \frac{1}{2}\|\mathbf{w}\|^2 \qquad (1)$$

Figure 1: The standard formula for SVM

Now, suppose we need to tune model parameter C, and we need to do it with ease. It's clearly seen from the equation that, tuning C also involves other parameters like *xi*, *i* and *w*; where, the regularization parameter C > 0,

is the norm of w. In the RHS it is the stabilizer and $\sum_{i=1}^{n} l(y_i, f(x_i))$ is the empirical loss term depending upon the target function f(xi). In the standard SVMs (either linear SVM or other variants), the regularized functionalities can be minimized further by solving the convex quadratic optimization problem. Once the problem is solved, it guarantees the unique global minimum solution to gain the best predictive performance. Therefore, the whole process is more or less an optimization problem.

In summary, *Figure 2, The model tuning process, consideration, and workflow* shows the tuning process and its consideration as a workflow:

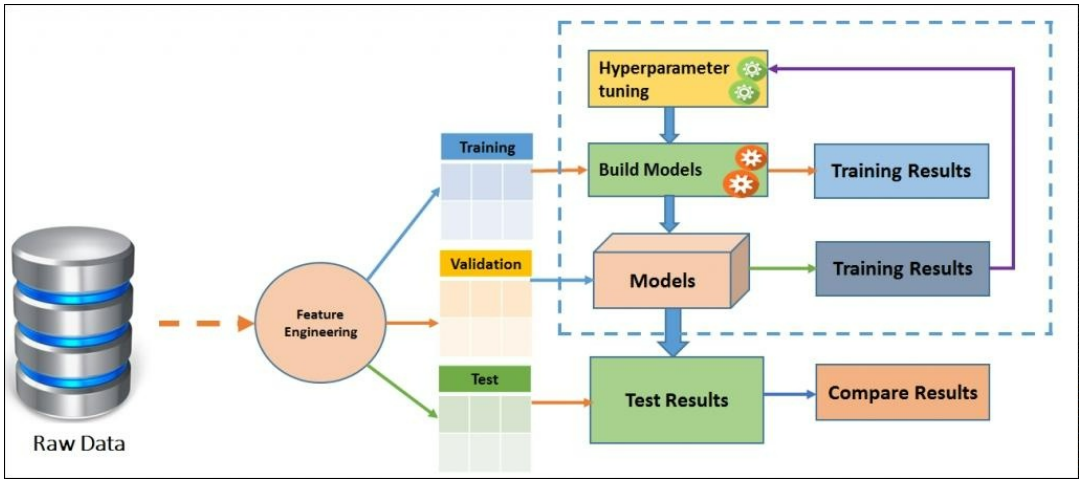


Figure 2: The model tuning process, consideration, and workflow

Now, if you are given the raw dataset, you will most probably do the pre-processing and split the dataset into training and test sets. Therefore, to tune hyperparameter C, you need to first split the training set into a validation training set and a validation test set.

After that, you might try tuning the parameters using the validation training set and validation test set. Then use the best parameters you've got and retrain the model on the complete training set. Now you can perform the testing on the test set as the final step.

Up to this point your approach seems to be okay, but which of the following two options do you think is better, on average?

- Would it be better to use the final model from validation, which was trained on a validation training set for the final testing?
- Or would it better to use the entire training set and retrain the model with the best parameters from the grid or random search? Although the parameters were not optimised for this set, we have final training data in this case.

Are you intending to go for option 1, because the parameters were already optimized on this training (that is, the validation training set) set? Or are you intending to go for option 2 because, although the parameters were not optimized for the training set, you have the final training data in this case? We suggest you go for option 2, but only if you trust your validation setup in option 2. The reason is that you have performed the **Cross Validation** (**CV**) to identify the most general parameters setup or else model selection or whatever you're trying to optimize. These findings should be applied to the entire training set and tested once on the test set.

Well, suppose you went for option 2; now the second challenge is evolving. How do we estimate the performance of a machine learning model? Technically, you might argue that we should nourish the training data to our learning algorithm to learn the optimal model. Then we could predict the labels based on the test labels. Thirdly, we count the number of wrong predictions on the test dataset to compute the model's error rate.

That's it? Not so fast my friend! Be contingent on our goal, unfortunately guesstimating the performance of that model is not that insignificant. Maybe we should address the previous question from another angle: why do we care about performance estimation at all? Well, ideally, the estimated performance of a model tells how well it performs on unobserved data - making predictions on future data is often the main problem we want to solve in applications of machine learning or the development of novel algorithms.

Finally, there are several other challenges depending upon the data structure, problem type, problem domain and appropriate use cases that need to be addressed that you will come across when you start practicing a lot.

Evaluating machine learning models

In this section, we will discuss how to evaluate a machine learning model because you should always evaluate a model to determine if it is ready to perform well consistently, predicting the target for new and future data. Obviously future data might have many unknown target values. Therefore, you need to check performance-related metrics such as the accuracy metric of the ML model on the data. In this regard, you need to provide a dataset containing scores generated from a trained model and then evaluate the model to compute a set of industry-standard evaluation metrics.

To evaluate a model appropriately, you need to present a sample of data that has been labeled with the target and this data will be used as the ground truth or facts dataset from the training data source. As we have already discussed, evaluating the predictive accuracy of an ML model with the same training dataset might not be useful.

The reason is that the model itself can remember the training data based on the rewards it receives instead of generalizing from it. Thus, when the ML model training has finished, you can get to know the target values to be predicted from the presented observations in the model. After that you can compare the predicted values that are returned by the ML model you have trained against the known target value.

Finally, you might be interested in computing the summary metric that shows the performance metrics to indicate how well the predicted and true values match accuracy parameters such as precision, recall, weighted true positive, weighted true negative, lift, and so on. In this section, however, we will particularly discuss how we can evaluate regression, classification (that is, binary classification, multiclass classification) and clustering model in the first place.

Evaluating a regression model

Assume you that you are an online currency trader and you work on Forex or Fortrade. Right now you have two currency pairs in mind to buy or sell, for example, GBP/USD and USD/JPY pairs. If you look at these two pairs carefully, you'll see that USD is common in both pairs. Now, if you observe the historical prices of USD, GBP or JPY you can predict the future outcome of whether you should open the trade in buy or sell.

This kind of problem can be treated as the typical regression problem. Here, the target variable (price in this case) is a continuous numeric value changing over time, based on the market opening time. Therefore, for

making predictions on the prices, based on the given feature values of a certain currency (that is, USD, GBP or JPY in this example), we can fit a simple linear regression or logistic regression model.

In this case, the feature values could be the historical prices and some external factors that drift the value of a certain currency or currency pair. In this way, the trained build model can predict the price of a certain currency.

The regression models (that is, linear, logistic or generalized liner regression models) can be used for finding or calculating the score of the same dataset we trained, now that the predicted prices of all of the currencies or the historical prices of these three currencies are available. We can further evaluate the performance of the model by analyzing, on average, how much the predicted prices deviate compared to the actual prices. In this way, people can guess whether the predicted price will go up or down and can earn money from online currency websites such as Forex or Fortrade and so on.

Evaluating a binary classification model

As we have already discussed in the binary classification scenario, there are only two possible outcomes for the target variable. For example: {0, 1}, {spam, hap}, {B, N}, {false, true} and {negative, positive} and so on. Now assume that you are given a dataset comprising researchers around the world with demographic, socio-economic and employment variables and you would like to predict the Ph.D. scholarship amount (that is, salary level) as a binary variable with the values of, say, {<=1.5K$, >=4.5K$}.

In this particular example, the negative class would represent the researcher whose salary or scholarship is less than or equal to $1,500 monthly. Consequently, the positive class, on the other hand, represents all other researchers whose salary is more than or equal to $4,500.

Now, from the problem scenario, it is clear that it is also a regression problem. As a result, you would train a model, score the data, and evaluate the results and see how much it deviates with actual labels. Therefore, in this type of problem, you would perform an experiment to evaluate the performance of a two-class (that is, binary class) logistic regression model, which is one of the most commonly used binary classifier in the area of ML.

Evaluating a multiclass classification model

In Chapter 6, *Building Scalable Machine Learning Pipelines*, we developed several applications and pipelines, and you might remember that we also developed a multiclass classification problem for the OCR dataset using logistic regression and showed the result using Multiclass Metrics. In that example, there were 26 classes for 26 characters (that is, from A to Z). We had to predict the class or label of a certain character and whether it really

fell under the correct class or labels. This kind of regression problem can be resolved using the multiclass classification method.

Therefore, in this type of problem, you would perform an experiment to evaluate the performance of a multiclass (that is, more than two class) logistic regression model too.

Evaluating a clustering model

Since the clustering models differ expressively from classification and regression models in many different aspects, if you evaluate a clustering model you will find a different set of statistics and performance related metrics for clustering models. The performance metrics that were returned in the clustering model evaluation technique, describe how many data points were assigned to each cluster, the amount of separation between clusters, and how tightly the data points are bunched within each cluster.

For example, if you recall, in Chapter 5, *Supervised and Unsupervised Learning by Examples*, you found a clustering problem that we discussed and resolved using Spark ML and MLlib in the *Unsupervised Learning with Spark: An example section*. In that particular example, we showed K-Means clustering of the neighborhood using the Saratoga NY Homes dataset, and showed an exploratory analysis based on price and lot size features for possible neighborhoods of houses located in the same area. This kind of problem can be resolved and evaluated using a clustering model. However, the current implementation of Spark does not yet provide any developed algorithm for model evaluation.

Validation and evaluation techniques

There are several widely used terms in machine learning application development that can be a bit tricky and confusing, so let's talk through them and sort them out. These terms include model, target function, hypothesis, confusion matrix, model deployment, induction algorithm, classifier, learning algorithms, cross-validation, and parameters:

- **Target function**: In reinforcement learning or predictive modeling, let's say we focus on modeling an object. The ultimate target is to learn or approximate a specific and unknown but targeted function. The target function is denoted as $f(x) = y$; where x and y both are variable and $f(x)$ is the true function that we want to model and it also signifies that we are trying to maximize or achieve the target value y.
- **Hypothesis**: A statistical hypothesis (don't confuse this with the research hypothesis proposed by the researcher) is a generalized function that is testable on the basis of observing a process. The process is similar to the true function that is modeled through a set of random variables from the training dataset. The objective behind the hypothesis testing is that a hypothesis is proposed for measuring the statistical relationship between the two data sets for the example training set and test set, where, both the datasets have to be statistically significant from an idealized (remember not a randomized or normalized model) model.
- **Learning algorithm**: As already stated, the ultimate goal of a machine learning application is to find or approximate the target function. In this continuous process, the learning algorithm is a set of instructions that models the target function using the training dataset. Technically, a learning algorithm often comes with a hypothesis space and formulates the final hypothesis.
- **Model**: A statistical model is a mathematical model that exemplifies a set of assumptions while generating sample and similar data from a larger population. Finally, the model often represents a significantly idealized form for the data-generating process. Also, in the machine learning area, the terms hypothesis and model are often used interchangeably.
- **Induction algorithm**: An induction algorithm takes input specific instances to produce a generalized model beyond these input instances.
- **Model deployment**: Model deployment usually denotes applying an already built and developed model to the real data in order to make a prediction for an example.
- **Cross-validation**: this is a method for estimating accuracy in terms of an error out of a machine learning model by dividing the data into K mutually exclusive subsets or folds of approximately equal size. The

model then is trained and tested K times in iteration, each time on the available dataset excluding a fold and then tested on that fold.
- **Classifier**: A classifier is a special case of hypothesis or discrete-valued function that is used to assign the most categorical class labels to particular data points such as the label point.
- **Regressor**: A regressor is also a special case of hypothesis that does the mapping from unlabeled features to a value within a predefined metric space.
- **Hyperparameters**: Hyperparameters are the tuning parameters of a machine learning algorithm to which a learning algorithm fits the training data.

According to Pedro D. et al., *A Few Useful Things to Know about Machine Learning* at http://homes.cs.washington.edu/~pedrod/papers/cacm12.pdf, we also need a few more things outlined, as follows:

- **Representation**: a classifier or regressor must be represented in some formal language that a computer can process by creating proper hypothesis spaces.
- **Evaluation**: An evaluation function (that is, objective function or scoring function) is needed to distinguish a good vs bad classifier r regressor, which is used internally by the algorithm by which the model has been build or trained.
- **Optimization**: We also need to have a method for searching among the classifiers or regressor, aiming for the highest scoring one. Thus the optimization is the key to the efficiency of the learner that helps to determine the optimal parameters.

In a nutshell, the key formula in learning in an ML algorithm is:

$$Learning = Representation + Evaluation + Optimization$$

Consequently, to validate and evaluate the trained model you need to understand the above terms very clearly so that you can conceptualize the ML problem and the proper uses of the Spark ML and MLlib APIs.

Parameter tuning for machine learning models

In this section, we will discuss tuning parameters and technique for machine learning models such as hyperparameter tuning, random search parameter tuning, and grid search parameter tuning and cross-validation.

Hyperparameter tuning

Hyperparameter tuning is a technique for choosing the right combination of parameters based on the performance of presented data. It is one of the fundamental requirements to obtain meaningful and accurate results from machine learning algorithms in practice. For example, suppose we have two hyperparameters to tune for a pipeline presented in *Figure 3*, a Spark ML pipeline model using a logistic regression estimator (dash lines only happen during pipeline fitting).

We can see that we have put three candidate values for each. Therefore, there would be nine combinations in total. However, only four are shown in the diagram, namely **Tokenizer**, **HashingTF**, **Transformer** and **Logistic Regression** (**LR**). Now we want to find the one that will lead to the model with the best evaluation result eventually. As we have already discussed in Chapter 6, *Building Scalable Machine Learning Pipelines,* the fitted model consists of the tokenizer, the hashing TF feature extractor, and the fitted logistic regression model:

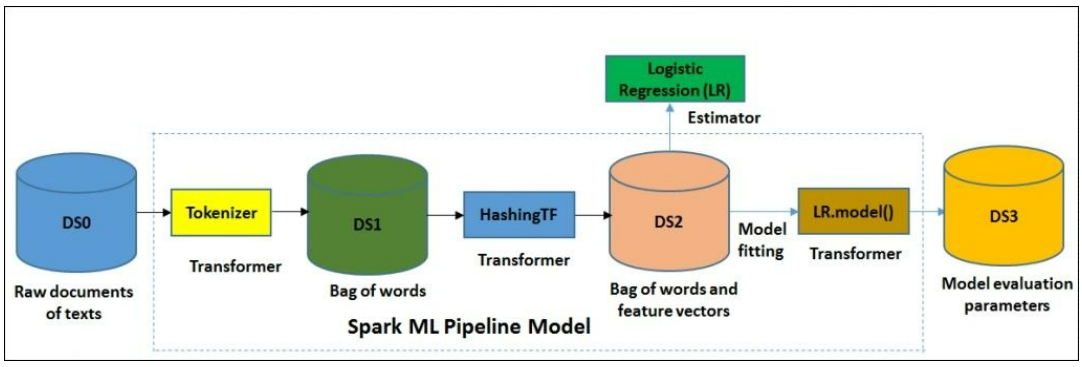


Figure 3: Spark ML pipeline model using logistic regression estimator (dash lines only happen during pipeline fitting)

Figure 3 shows the typical workflow of the previously mentioned pipeline. The dash line however happens only during the pipeline fitting.

As mentioned earlier, the fitted pipeline model is a transformer. The transformer can be used for prediction, model validation, and model inspection. In addition, we also argued that one ill-fated distinguishing

characteristic of the ML algorithms is that typically they have many hyperparameters that need to be tuned for better performance.

For example, the degree of regularizations in these hyperparameters is distinctive from the model parameters optimized by the Spark MLlib. As a consequence, it is really hard to guess or measure the best combination of hyperparameters without expert knowledge of the data and the algorithm to use. Since the complex dataset is based on the ML problem type, the size of the pipeline and the number of hyperparameters may grow exponentially (or linearly), the hyperparameter tuning becomes cumbersome even for an ML expert, not to mention that the result of the tuning parameters may become unreliable

According to Spark API documentation provided at http://spark.apache.org/docs/latest/ml-guide.html, a unique and uniform API is used for specifying Spark ML Estimators and Transformers. A `ParamMap` is a set of (parameter, value) pairs with a `Param` as a named parameter with self-contained documentation provided by Spark. Technically, there are two ways for passing the parameters to an algorithm as specified in the following options:

- Setting parameters. For example, if an LR is an instance of `LogisticRegression` (that is, Estimator), you can call the `setMaxIter()` method as follows: `LR.setMaxIter(5)`. It essentially fits the model pointing the regression instance as follows: `LR.fit()`. In this particular example, there would be at most five iterations.
- The second option involves passing a `ParamMaps` to `fit()` or `transform()` (refer *Figure 1* for details). In this circumstance, any parameters will be overridden by the `ParamMaps` previously specified via setter methods in the ML application-specific codes or algorithms.

Tip

An excellent way to create a shortlist of well-performing algorithms for your dataset is to use the Caret package in R since tuning in Spark is not that robust. Caret is a package in R created and maintained by Max Kuhn from Pfizer. Development started in 2005 and was later made open source and uploaded to CRAN which is actually an acronym which stands for **Classification And Regression Training** (**CARET**). It was initially developed out of the need to run multiple different algorithms for a given problem. Interested readers can have a look at that package for theoretical and practical consideration by visiting: http://topepo.github.io/caret/index.html.

Grid search parameter tuning

Suppose you have selected your hyperparameters, and by applying tuning, you now also need to find the features. In this regard, a full grid search of the space of hyperparameters and features is computationally too intensive. Therefore, you need to perform a fold of the K-fold cross-validation instead of a full grid search:

- Tune the required hyperparameters using cross-validation on the training set of the fold, using all the available features
- Select the required features using those hyperparameters
- Repeat the computation for each fold in K
- The final model is constructed on all the data using the N most prevalent features that were selected from each fold of CV

The interesting thing is that the hyperparameters would also be tuned again using all the data in a cross-validation loop. Would there be a large downside from this method as compared to a full grid search? In essence, I am doing a line search in each dimension of free parameters (finding the best value in one dimension, holding that constant then finding the best in the next dimension), rather than every single combination of parameter settings.

The most important downside for searching along single parameters instead of optimizing them altogether is that you ignore interactions. It is quite common that, for instance, more than one parameter influences model complexity. In that case, you need to look at their interaction in order to successfully optimize the hyperparameters. Depending on how large your data set is and how many models you compare, optimization strategies that return the maximum observed performance may run into trouble (this is true for both grid search and your strategy).

The reason is that searching through a large number of performance estimates for the maximum skims the variance of the performance estimate: you may just end up with a model and training/test split combination that accidentally happens to look good. Even worse, you may get several perfect-looking combinations, and the optimization then cannot know which model to choose and thus becomes unstable.

Random search parameter tuning

The default method for optimizing tuning parameters in the train is to use a grid search. This approach is usually effective but, in cases when there are many tuning parameters, it can be inefficient. There are a number of models where this can be beneficial in finding reasonable values of the tuning parameters in a relatively short time. However, there are some models where the efficiency in a small search field can cancel out other optimizations. Unfortunately, the current implementation in Spark for

hyperparameter tuning does not provide any technique for random search tuning.

> **Tip**
>
> In contrast, for example, a number of models in CARET utilize the *sub-model trick* where M tuning parameter combinations are evaluated; potentially far fewer than M model fits are required. This approach is best leveraged when a simple grid search is used. For this reason, it may be inefficient to use a random search. Finally, many of the models wrapped by train have a small number of parameters. The average number of parameters is 1.7. To use a random search, another option is available in **trainControl** called search. Possible values of this argument are `grid` and `random`. The built-in models contained in CARET contain code to generate random tuning parameter combinations. The total number of unique combinations is specified by the **tuneLength** option to train.

Cross-validation

Cross-validation (also called the **Rotation Estimation** (**RE**)) is a model validation technique for assessing the quality of the statistical analysis and results. The target is to make the model generalize towards an independent test set.

One of the perfect uses of the cross-validation technique is making a prediction from a machine learning model. Technically, it will help if you want to estimate how a predictive model will perform accurately in practice when you deploy it as an ML application.

During the cross-validation process, a model is usually trained with a dataset of a known type. Conversely, it is tested using a dataset of unknown type. In this regard, cross-validation help describes a dataset to test the model in the training phase using the validation set.

However, in order to minimize the flaws in the machine learning model such as overfitting and underfitting, the cross-validation technique provides insights into how the model will generalize to an independent set.

There are two types of cross-validation that can be typed as follows:

- **Exhaustive cross-validation**: This includes leave-p-out cross-validation and leave-one-out cross-validation

- **Non-exhaustive cross-validation**: This includes the K-fold cross-validation and repeated random sub-sampling validation cross-validation

Detailed discussion of these will not be conducted in this book due to page limitation. Moreover, using Spark ML and Spark MLlib, readers will be able to perform the cross-validation following our examples in the next section.

Except for the time series data, in most of the cases, the researcher/data scientist/data engineer uses 10-fold cross-validation instead of testing on a validation set (where K = 10). This is the most widely used cross-validation technique across the use cases and problem type. Moreover, to reduce the variability, multiple iterations of cross-validation are performed using different partitions; finally, the validation results are averaged over the rounds across.

Using cross-validation instead of conventional validation has two main advantages outlined as follows:

- Firstly, if there is not enough data available to partition across the separate training and test sets, there's the chance of losing significant modelling or testing capability.
- Secondly, the K-fold cross-validation Estimator has a lower variance than a single hold-out set Estimator. This low variance limits the variability and is again very important if the amount of available data is limited.

In these circumstances, a fair way to properly estimate the model prediction and related performance is to use cross-validation as a powerful general technique for model selection and validation.

A more technical example will be shown in the *Machine learning model selection* section. Let's draw a concrete example to illustrate this. Suppose, we need to perform manual features and a parameter selection for the model tuning and, after that, perform a model evaluation with a 10-fold cross-validation on the entire dataset. What would be the best strategy? We would suggest you go for the strategy that provides an optimistic score as follows:

- Divide the dataset into training, say 80%, and testing 20% or whatever you chose
- Use the K-fold cross-validation on the training set to tune your model
- Repeat the CV until you find your model optimized and therefore tuned
- Now use your model to predict on the testing set to get an estimate of out-of-model errors

Hypothesis testing

Hypothesis testing is a statistical tool used to determine whether a result is statistically significant. Additionally, it can also be used to justify whether the result you received occurred by chance or whether it is an actual result.

In this regard, moreover, according to Oracle developers at https://docs.oracle.com/cd/A57673_01/DOC/server/doc/A48506/method.htm, a certain workflow would provide better performance tuning. The typical steps they have suggested are as follows:

- Set clear goals for tuning
- Create minimum repeatable tests
- Test the hypothesis
- Keep all records
- Avoid common errors
- Stop tuning when the objectives are achieved

Usually, the observed value tobs of the test statistic T is first computed. After that, the probability, also called p-value, is calculated under the null hypothesis. Finally, if and only if the p-value is less than the significance level (the selected probability) threshold, the null hypothesis is rejected in favor of the alternative hypothesis.

To find out more, refer to the following publication: *R.A. Fisher et al., Statistical Tables for Biological Agricultural and Medical Research, 6th ed., Table IV, Oliver & Boyd, Ltd., Edinburgh*.

Here are two rules of thumb (although these may vary for your case depending on data quality and types):

- If the p-value is $p > 0.05$, accept your hypothesis. Note that if a deviation is small enough that chance could be the acceptance level. A p-value of 0.6, for example, means that there is a 60% probability of any deviation from the expected result. However, this is within the range of an acceptable deviation.
- If the p-value is $p < 0.05$, reject your hypothesis by concluding that some factors other than chance are operating for the deviation to be perfect. Similarly, a p-value of 0.01 means that there is only a 1% chance that this deviation is due to chance alone, which means that other factors must be involved, and these need to be addressed.

However, these two rules might not be applicable in every hypothesis testing. In the next subsection, we will show an example of hypothesis testing using Spark MLlib.

Hypothesis testing using ChiSqTestResult of Spark MLlib

According to the API documentation provided by Apache at http://spark.apache.org/docs/latest/mllib-statistics.html#hypothesis-testing, the current implementation of the Spark MLlib supports Pearson's chi-squared ($\chi 2$) tests for goodness of fit and independence:

- The goodness of the fit, or if the independence test is being conducted, is determined by the input data types
- The goodness of the fit test requires an input of type vector (mostly dense vectors although it works for sparse vectors). On the other hand, the independence test requires a Matrix as an input format

In addition to these, Spark MLlib also supports input type RDD [LabeledPoint] to enable feature selection via chi-square independence tests, especially for the SVM or regression based test where the statistical class provides the necessary methods to run Pearson's chi-squared tests.

Additionally, Spark MLlib provides a 1-sample, 2-sided implementation of the **Kolmogorov-Smirnov** (**KS**) test for equality of probability distributions. Spark MLlib provides online implementations of some tests to support use cases like A/B testing. These tests may be performed on a Spark Streaming DStream [(Boolean, Double)] where the first element of each tuple indicates the control group (false) or treatment group (true) and the second element is the value of an observation.

However, due to brevity and page limitation, this two testing technique will not be discussed. The following example demonstrates how to run and interpret hypothesis tests through the `ChiSqTestResult`. In the example, we will show three tests: goodness of fit of the result on the dense vector created from the breast cancer diagnosis dataset, an independence test for a randomly created matrix and finally, an independence test on a contingency table from the cancer dataset itself.

Step 1: Load required packages

Here is the code to load the required packages:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import org.apache.spark.api.java.JavaRDD;
import
org.apache.spark.api.java.function.Function;
import org.apache.spark.mllib.linalg.DenseVector;
import org.apache.spark.mllib.linalg.Matrices;
import org.apache.spark.mllib.linalg.Matrix;
```

```
import org.apache.spark.mllib.linalg.Vector;
import
org.apache.spark.mllib.regression.LabeledPoint;
import org.apache.spark.mllib.stat.Statistics;
import
org.apache.spark.mllib.stat.test.ChiSqTestResult;
import org.apache.spark.rdd.RDD;
import org.apache.spark.sql.SparkSession;
import
com.example.SparkSession.UtilityForSparkSession;
```

Step 2: Create a Spark session

The following code helps us to create a Spark Session:

```
static SparkSession spark =
UtilityForSparkSession.mySession();
```

The implementation of the `UtilityForSparkSession` class is as follows:

```
public class UtilityForSparkSession {
  public static SparkSession mySession() {
  SparkSession spark = SparkSession
               .builder()

 .appName("JavaHypothesisTestingOnBreastCancerData
")
               .master("local[*]")
              .config("spark.sql.warehouse.dir",
"E:/Exp/")
              .getOrCreate();
    return spark;
  }
}
```

Step 3: Perform the goodness of fit test

First we need to prepare a dense vector from a categorical dataset such as the Wisconsin Breast Cancer Diagnosis dataset. As we have already provided many examples on this dataset, we will not discuss the data exploration anymore in this section. The following line of code collects the vectors that we created using the `myVector()` method:

```
Vector v = myVector();
```

Where, the implementation of the `myVector()` method goes as follows:

```
public static Vector myVector() throws
NumberFormatException, IOException {
```

```
BufferedReader br = new BufferedReader(new
FileReader(path));
    String line = nulNow let's compute the
goodness of the fit. Note, if a second vector
tol;
    Vector v = null;
    while ((line = br.readLine()) != null) {
      String[] tokens = line.split(",");
      double[] features = new double[30];
      for (int i = 2; i < features.length; i++) {
        features[i-2] =

 Double.parseDouble(tokens[i]);
      }
      v = new DenseVector(features);
    }
    return v;
  }
```

Now let's compute the goodness of the fit. Note, if a second vector to test is not supplied as a parameter, the test runs occur against a uniform distribution automatically:

```
ChiSqTestResult goodnessOfFitTestResult =
Statistics.chiSqTest(v);
```

Now let's print the result of the goodness using the following:

```
System.out.println(goodnessOfFitTestResult +
"\n");
```

Note the summary of the test includes the p-value, degrees of freedom, test statistic, the method used, and the null hypothesis. We got the following output:

```
Chi squared test summary:
method: pearson
degrees of freedom = 29
statistic = 4528.611649568829
pValue = 0.0
```

There is a very strong presumption against a null hypothesis: the observed follows the same distribution as expected.

Since the p-value is low enough to be insignificant, consequently we cannot accept the hypothesis based on the data.

Step 4: An independence test on the contingency matrix

First let's create a contingency 4x3 matrix randomly. Here, the matrix appears as follows:

```
 ((1.0, 3.0, 5.0, 2.0), (4.0, 6.0, 1.0, 3.5),
(6.9, 8.9, 10.5, 12.6))
Matrix mat = Matrices.dense(4, 3, new double[]
{ 1.0, 3.0, 5.0, 2.0, 4.0, 6.0, 1.0, 3.5, 6.9,
8.9, 10.5, 12.6});
```

Now let's conduct Pearson's independence test on the input contingency matrix:

```
ChiSqTestResult independenceTestResult =
Statistics.chiSqTest(mat);
```

Now let's evaluate the test result and give the summary of the test including the p-value and the degrees of freedom:

```
System.out.println(independenceTestResult +
"\n");
```

We got the following statistic summarized as follows:

```
Chi squared test summary:
method: pearson
degrees of freedom = 6
statistic = 6.911459343085576
pValue = 0.3291131185252161
No presumption against null hypothesis: the
occurrence of the outcomes is statistically
independent.
```

Step 5: An independence test on the contingency table

First, let's create a contingency table by means of RDDs from the cancer dataset as follows:

```
static String path =
"breastcancer/input/wdbc.data";
RDD<String> lines =
spark.sparkContext().textFile(path, 2);
JavaRDD<LabeledPoint> linesRDD =
lines.toJavaRDD().map(new Function<String,
LabeledPoint>() {
    public LabeledPoint call(String lines) {
    String[] tokens = lines.split(",");
    double[] features = new double[30];
    for (int i = 2; i < features.length; i++) {
```

```
    features[i - 2] =
Double.parseDouble(tokens[i]);
            }
    Vector v = new DenseVector(features);
    if (tokens[1].equals("B")) {
    return new LabeledPoint(1.0, v); // benign
      } else {
    return new LabeledPoint(0.0, v); // malignant
        }
      }
    });
```

We have constructed a contingency table from the raw (feature, label) pairs and used it to conduct the independence test. Now let's conduct the test as `ChiSquaredTestResult` for every feature against the label:

```
ChiSqTestResult[] featureTestResults =
Statistics.chiSqTest(linesRDD.rdd());
```

Now let's observe the test result against each column (that is, for each 30 feature point) using the following code segment:

```
int i = 1;
for (ChiSqTestResult result : featureTestResults)
{
System.out.println("Column " + i + ":");
System.out.println(result + "\n");
i++;
}
```

```
Column 1:
Chi-squared test summary:
method: Pearson
degrees of freedom = 455
statistic = 513.7450859274513
pValue = 0.02929608473276224
Strong presumption against null hypothesis: the
occurrence of the outcomes is statistically
independent.

Column 2:
Chi-squared test summary:
method: Pearson
degrees of freedom = 478
statistic = 498.41630331377735
pValue = 0.2505929829141742
No presumption against null hypothesis: the
occurrence of the outcomes is statistically
independent.

Column 3:
Chi-squared test summary:
method: Pearson
```

```
degrees of freedom = 521
statistic = 553.3147340697276
pValue = 0.1582572931194156
No presumption against null hypothesis: the
occurrence of the outcomes is statistically
independent.
.
.
Column 30:
Chi-squared test summary:
method: Pearson
degrees of freedom = 0
statistic = 0.0
pValue = 1.0
No presumption against null hypothesis: the
occurrence of the outcomes is statistically
independent.
```

From this result, we can see that for some feature points (that is, column) we have a large p-value compared to the others. Readers are, therefore, advised to select the proper dataset and do the hypothesis test prior to applying the hyperparameter tuning. There is no concrete example in this regard since the result may vary against the datasets you have.

Hypothesis testing using the Kolmogorov–Smirnov test from Spark MLlib

Since Spark release 1.1.0, Spark also provides the facility of doing the hypothesis testing using for the real-time streaming data through the Kolmogorov-Smirnov test. Where, the probability of obtaining a test statistic result (at least as extreme as the one) that was actually observed. It actually assumes that the null hypothesis is always true.

> **Tip**
>
> For more details, interested readers should refer to the Java class (`JavaHypothesisTestingKolmogorovSmirnovTestExample.java`) in the Spark distribution under the following directory: `spark-2.0.0-bin-hadoop2.7\examples\src\main\java\org\apache\spark\examples\mllib`.

Streaming significance testing of Spark MLlib

Other than the Kolmogorov-Smirnov test, Spark also supports Streaming significance testing, which is an online implementation of the hypothesis

testing like the A/B testing? These tests can be performed on a Spark streaming using DStream (more technical discussion on this topic will be carried out in Chapter 9, *Advanced Machine Learning with Streaming and Graph Data*).

The MLlib based streaming significance testing supports the following two parameters:

- **peacePeriod**: This is the number of initial data points from the stream to ignore. This is actually used to mitigate novelty effects and the quality of the streaming you will be receiving.
- **windowSize**: This is the number of past batches over which to perform hypothesis testing. If you set its value to 0, it will perform cumulative processing using all the prior batches received and processed.

Interested readers should refer to the Spark API documentation at http://spark.apache.org/docs/latest/mllib-statistics.html#hypothesis-testing.

Machine learning model selection

Most machine learning algorithms are dependent on various parameters. When we train a model, we need to provide values for those parameters. The efficacy of the trained model is dependent on the model parameters that we choose. The process of finding out the optimal set of parameters is known as model selection.

Model selection via the cross-validation technique

When performing machine learning using Python's scikit-learn library or R, you can often get a reasonably predictive performance by using out-of-the-box settings for your models. However, the payoff can be huge if you invest some time in tuning models to your specific problem and data set.

However, we also need to consider other issues like overfitting, cross-validation, and bias-variance trade-off. These ideas are central to doing a good job at optimizing the hyperparameters of algorithms.

In this section, we will explore the concepts behind Hyperparameter optimization and demonstrate the process of tuning and training a logistic regression classifier for the famous Spam Filtering dataset. The goal is to tune and apply a logistic regression to these features in order to predict whether a given email/SMS is spam or not-spam:

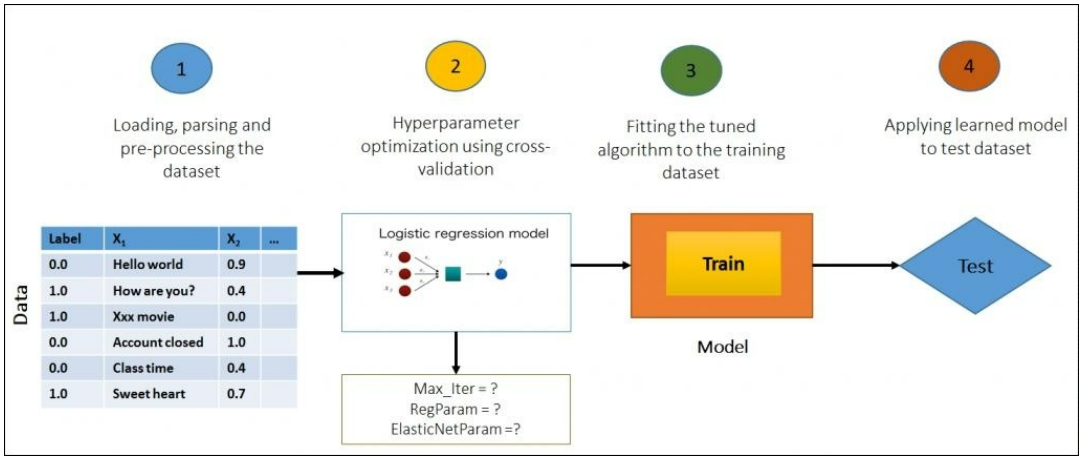


Figure 4: Model selection via cross-validation

Cross-validation and Spark

Pipelines enable model selection by tuning an entire Pipeline at once, rather than tuning each element in the Pipeline unconnectedly. See the API documentation at http://spark.apache.org/docs/latest/ml-guide.html.

The current implementation of Spark ML supports model selection using the `CrossValidator` class. It takes an Estimator, a set of `ParamMaps`, and an Evaluator. The model selection task begins with splitting the dataset (that is, splitting it into a set of folds) the folds of which are then used as separate training and test datasets.

For example, with K=10 folds, the `CrossValidator` will generate 10 (training, test) dataset pairs. Each of these uses two thirds (2/3) of the data for the training, and the other third (1/3) for the testing. After that, the `CrossValidator` iterates through the set of `ParamMaps`. For each `ParamMap`, it trains the given Estimator and evaluates it using the available Evaluator. The Evaluator can be a related ML task for example:

- `RegressionEvaluator` for regression related problems
- `BinaryClassificationEvaluator` for binary data and its related problems
- `MultiClassClassificationEvaluator` for a multiclass problem

When it comes to the best `ParamMap` selection, a default metric is used. Note that the `ParamMap` can be also be overridden by the `setMetric()` method in each of these evaluators. In contrast, when it comes to the best model selection:

- The `ParamMap` produces the best evaluation metric
- The evaluations metrics are then averaged over the K folds
- Finally, the best model is selected

Once the best `ParamMap` and model are selected, the `CrossValidator` fits the Estimator using them for the entire dataset.

To get a clearer insight into the `CrossValidator` and to select from a grid of parameters, Spark uses the `ParamGridBuilder` utility to construct the parameter grid. For example, suppose the parameter grid has a value of 4 as the `hashingTF.numFeatures` and a value of 3 for the LR.`regParam`. Also, lets say that the `CrossValidator` uses 10 folds.

Once, these values are multiplying results to 120 (that is, 4*3 *10 = 120), this signifies that a significant amount of different models (that is, 120) are being trained. Therefore, using the `CrossValidator` sometimes can be very expensive. Nevertheless, it is also a well-established method for choosing associated performance and hyperparameters that are sounder statistically compared to the heuristic-based hand tuning.

Tip

Interested readers may refer to the following three books for more insight:

Evan R. Sparks et al., *Automating Model Search for Large-Scale Machine Learning*, ACM, 978-1-4503-3651-2/15/08, http://dx.doi.org/10.1145/2806777.2806945.

Cawley, G. C. & Talbot, N. L on over-fitting in model selection and subsequent selection bias in performance evaluation, *The Journal of Machine Learning Research*, JMLR. org, 2010, 11, 2079-2107.

N. Japkowicz and M. Shah, *Evaluating learning algorithms: a classification perspective*, Cambridge University Press, 2011.

In the next sub-section, we will show how to perform cross-validation on a dataset for model selection using Spark ML API.

Cross-validation using Spark ML for SPAM filtering a dataset

In this sub-section, we will show you how to perform cross-validation on the e-mail spam dataset for model selection. We will use logistic regression in the first place then we will move forward for other models. Finally, we will recommend the most suitable model for e-mail spam classification.

Step 1: Import necessary packages/libraries/APIs

Here is the code to import necessary packages/libraries/APIs:

```
import java.io.Serializable;
import java.util.Arrays;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import
org.apache.spark.api.java.function.Function;
import org.apache.spark.ml.Pipeline;
import org.apache.spark.ml.PipelineStage;
import
org.apache.spark.ml.classification.LogisticRegression;
import
org.apache.spark.ml.evaluation.BinaryClassificationEvaluator;
import org.apache.spark.ml.feature.HashingTF;
import org.apache.spark.ml.feature.Tokenizer;
import org.apache.spark.ml.param.ParamMap;
```

```
import org.apache.spark.ml.tuning.CrossValidator;
import
org.apache.spark.ml.tuning.CrossValidatorModel;
import
org.apache.spark.ml.tuning.ParamGridBuilder;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;
import scala.Tuple2;
```

Step 2: Initialize the necessary Spark environment

The following code helps us to initialize the necessary Spark environment:

```
 static SparkSession spark = SparkSession
       .builder()

 .appName("CrossValidationforSpamFiltering")
       .master("local[*]")
       .config("spark.sql.warehouse.dir",
"E:/Exp/")
       .getOrCreate();
```

Here we set the application name as cross-validation, the master URL as `local[*]`, and the Spark session as the entry point of the program. Please set these parameters accordingly. Most importantly, set the warehouse directory as `E:/Exp/` and replace it with the appropriate path.

Step 3: Prepare a Dataset from the SMS spam dataset

Take the e-mail spam data as your input, prepare a dataset out of the data, using it as raw text, and check if the data was read properly by calling the `show()` method:

```
Dataset<Row> df =
spark.read().text("input/SMSSpamCollection.txt");
df.show();
```



Figure 5: The top 20 rows

To learn more about the data, please refer to the section *Pipeline - an example with Spark ML*, in Chapter 4, *Extracting Knowledge through Feature Engineering*, and the description for the dataset exploration. As you can see in *Figure 5*, The top 20 rows, there are only two labels - spam or ham (that is, it is a binary classification problem) associated along with the text values (that is, each row).

However, there is no numeric label or ID. Therefore, we need to prepare the dataset (training set) such that the data frame also contains the ID, and labels along with text (that is, value) so that we can prepare a test set and predict the corresponding labels using any classification algorithm (that is, logistic regression) and can decide if our model selection is appropriate.

However, for doing so we need to prepare the training dataset first. As you can see, the data frame shown above has only one column and, as mentioned previously, we do need to have three columns. If we prepare an RDD from the previous Dataset (that is, `df`), it would be easier for us to make that transformation.

Step 4: Create Java pairs of RDDs to store the rows and indices

Create a Java pair of RDDs by converting (transforming) the DataFrame (that is, `df`) to Java RDD and by zipping the index:

```
JavaPairRDD<Row, Long> rowRDD =
df.toJavaRDD().zipWithIndex();
```

Step 5: Create LabeledDocument RDDs

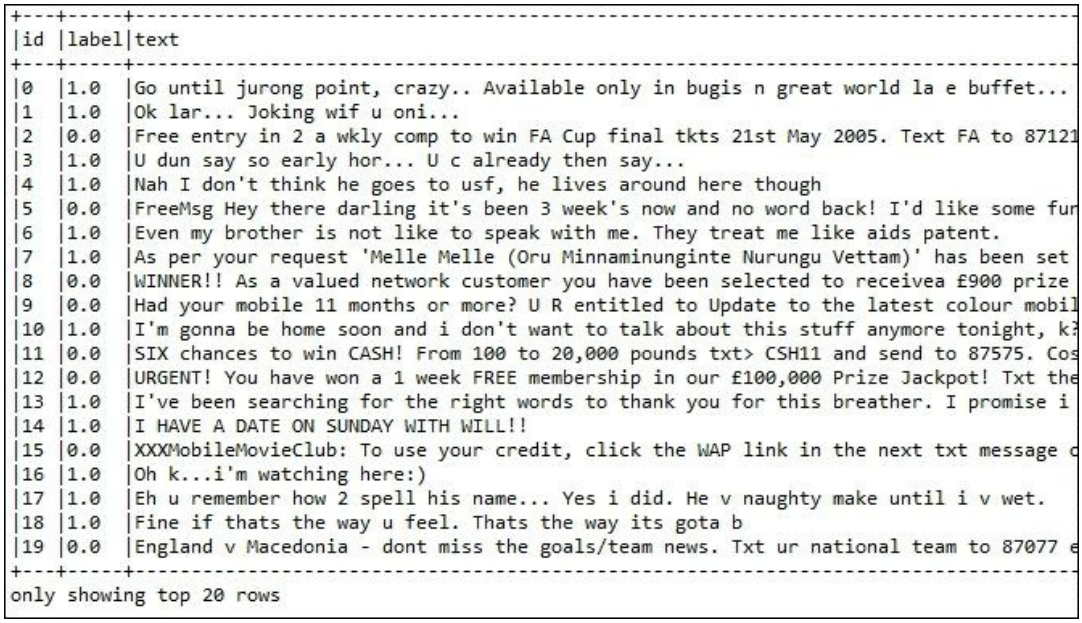
Create `LabeledDocument` Java RDDs by splitting the dataset based on two labels and converting the text label to a numeric label (that is, `1.0` if ham otherwise 0.0). Note that the `LabeledDocument` is a user-defined class that is discussed in *step 6*:

```
JavaRDD<LabeledDocument> splitedRDD =
rowRDD.map(new Function<Tuple2<Row, Long>,
LabeledDocument>() {
@Override
public LabeledDocument call(Tuple2<Row, Long> v1)
throws Exception {
  Row r = v1._1;
  long index = v1._2;
  String[] split = r.getString(0).split("\t");
  if(split[0].equals("ham"))
    return new LabeledDocument(index,split[1],
1.0);
  else
    return new LabeledDocument(index,split[1],
0.0);
      }
    });
```

Step 6: Prepare the training dataset

Prepare the training dataset from the `LabeledDocument` RDDs using the `createDataFrame()` method and by specifying the class. Finally, see the data frame structure using the `show()` method as follows:

```
Dataset<Row> training =
spark.createDataFrame(splitedRDD,
LabeledDocument.class);
training.show(false);
```



id	label	text
0	1.0	Go until jurong point, crazy.. Available only in bugis n great world la e buffet...
1	1.0	Ok lar... Joking wif u oni...
2	0.0	Free entry in 2 a wkly comp to win FA Cup final tkts 21st May 2005. Text FA to 87121
3	1.0	U dun say so early hor... U c already then say...
4	1.0	Nah I don't think he goes to usf, he lives around here though
5	0.0	FreeMsg Hey there darling it's been 3 week's now and no word back! I'd like some fur
6	1.0	Even my brother is not like to speak with me. They treat me like aids patent.
7	1.0	As per your request 'Melle Melle (Oru Minnaminunginte Nurungu Vettam)' has been set
8	0.0	WINNER!! As a valued network customer you have been selected to receivea £900 prize
9	0.0	Had your mobile 11 months or more? U R entitled to Update to the latest colour mobil
10	1.0	I'm gonna be home soon and i don't want to talk about this stuff anymore tonight, k?
11	0.0	SIX chances to win CASH! From 100 to 20,000 pounds txt> CSH11 and send to 87575. Cos
12	0.0	URGENT! You have won a 1 week FREE membership in our £100,000 Prize Jackpot! Txt the
13	1.0	I've been searching for the right words to thank you for this breather. I promise i
14	1.0	I HAVE A DATE ON SUNDAY WITH WILL!!
15	0.0	XXXMobileMovieClub: To use your credit, click the WAP link in the next txt message c
16	1.0	Oh k...i'm watching here:)
17	1.0	Eh u remember how 2 spell his name... Yes i did. He v naughty make until i v wet.
18	1.0	Fine if thats the way u feel. Thats the way its gota b
19	0.0	England v Macedonia - dont miss the goals/team news. Txt ur national team to 87077 e

only showing top 20 rows

Figure 6: The newly created label and ID from the dataset; where ID is the number of row.

From *Figure 6*, *The newly created label and ID from the dataset; where ID is the number of rows*, we can see that the new training dataset has three columns: ID, test, and label. It can actually be done by adding a new ID against each row (that is, each line) of the original document. Let's create a class, called `Document` for our purpose, that should set a unique ID against each line of text. The structure of the class can be something like the following:

```
public class Document implements Serializable {
  private long id;
  private String text;
  //Initialise the constructor that should take
two parameters: id and text//
  Set the id
  Get the id
  Set the text
  Get the text
  }
```

Now let's look at the structure of the class constructor:

```
  public LabeledDocument (long id, String text) {

      this.id = id;
           this.text = text;
  }
```

The setter and getter for the ID could be something like this:

```
  public void setId(long id) {
```

```
    this.id = id;
  }
  public long getId() {
    return this.id;
  }
```

Similarly the setter and getter methods for the text could be something like this:

```
  public String getText() {
    return this.text;
  }

  public void setText(String text) {
    this.text = text;
  }
```

Therefore, if we summarize, the `Document` class could be something like this:

```
import java.io.Serializable;
public class Document implements Serializable {
  private long id;
  private String text;
  public Document(long id, String text) {
    this.id = id;
    this.text = text;
  }
  public long getId() {
    return this.id;
  }
  public void setId(long id) {
    this.id = id;
  }
  public String getText() {
    return this.text;
  }
  public void setText(String text) {
    this.text = text;
  }
}
```

The structure of the `LabeledDocument` class, on the other hand, can be as follows and that can be extended from the Document class (to be discussed later):

Now let's look at the structure of the class constructor:

```
  public LabeledDocument(long id, String text,
double label) {
    this.label = label;
```

```
  }
```

However, we're not done yet since we will be extending the `Document` class we need to inherit the constructor from the `Document` class using the `super()` method as follows:

```
  public LabeledDocument(long id, String text,
double label) {
    super(id, text);
    this.label = label;
  }
```

Now the setter method can be something like this:

```
  public void setLabel(double label) {
    this.label = label;
  }
```

And of course the getter method for the label could be something like this:

```
  public double getLabel() {
    return this.label;
  }
```

Therefore, in a nutshell, the `LabelDocument` class is as follows:

```
import java.io.Serializable;
public class LabeledDocument extends Document
implements Serializable {
  private double label;
  public LabeledDocument(long id, String text,
double label) {
    super(id, text);
    this.label = label;
  }
  public double getLabel() {
    return this.label;
  }
  public void setLabel(double label) {
    this.label = label;
  }
}
```

Step 7: Configure an ML pipeline

Configure an ML pipeline, which consists of three stages: `tokenizer`, `hashingTF`, and `lr`:

```
Tokenizer tokenizer = new
Tokenizer().setInputCol("text").setOutputCol("words");
HashingTF hashingTF = new
HashingTF().setNumFeatures(1000).setInputCol(tokenizer.getOutputCol()).se
LogisticRegression lr = new
LogisticRegression().setMaxIter(10).setRegParam(0.01);
Pipeline pipeline = new Pipeline().setStages(new
PipelineStage[] { tokenizer, hashingTF, lr });
```

Step 8: Construct a grid of parameters to search over

Currently, Spark uses a `ParamGridBuilder` to construct a grid of parameters to search over. In this regard, suppose we have three values for `hashingTF.numFeatures` and two values for `lr.regParam`, this grid will have 3 x 2 = 6 parameter settings for the `CrossValidator` to choose from:

```
ParamMap[] paramGrid = new ParamGridBuilder()
.addGrid(hashingTF.numFeatures(), new int[] { 10,
100, 1000 }).addGrid(lr.regParam(), new double[]
{ 0.1, 0.01 })
 .build();
```

We now treat the pipeline as an estimator, wrapping it in a `CrossValidator` instance. This will allow us to jointly choose parameters for all Pipeline stages. A `CrossValidator` requires an Estimator, a set of Estimator `ParamMaps`, and an Evaluator. Note that the evaluator here is a `BinaryClassificationEvaluator` and its default metric is `areaUnderROC`.

Step 9: Create a CrossValidator instance

Here is the code to create a `CrossValidator` instance:

```
    CrossValidator cv = new CrossValidator()
        .setEstimator(pipeline)
        .setEvaluator(new
BinaryClassificationEvaluator())
        .setEstimatorParamMaps(paramGrid)
        .setNumFolds(5); // 5-fold cross
validation
```

Step 10: Run cross-validation

Run the cross-validation and choose the best set of parameters. Just use the following code segments:

```
CrossValidatorModel cvModel = cv.fit(training);
```

Now your `CrossValidator` model is ready to perform the prediction. However, before that, we need a test set or validation set. Now let's prepare a sample test set. Just create a dataset using the following code segment:

```
    Dataset<Row> test =
spark.createDataFrame(Arrays.asList(
      new Document(4L, "FreeMsg CALL j k"),
      new Document(5L, "Siva  hostel"),
      new Document(6L, "darren now"),
     new Document(7L, "Sunshine Quiz! Win a super
Sony")),Document.class);
```

Now let's see the structure of the test set by calling the `show()` method in *Figure 7*:

```
+---+-------------------------------+
|id |text                           |
+---+-------------------------------+
|4  |FreeMsg CALL j k               |
|5  |Siva  hostel                   |
|6  |darren now                     |
|7  |Sunshine Quiz! Win a super Sony|
+---+-------------------------------+

16/10/07 17:07:27 INFO SparkContext: Invoking stop() from shutdown hook
16/10/07 17:07:27 INFO SparkUI: Stopped Spark web UI at http://10.2.17.24:4040
16/10/07 17:07:27 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
16/10/07 17:07:27 INFO MemoryStore: MemoryStore cleared
16/10/07 17:07:27 INFO BlockManager: BlockManager stopped
16/10/07 17:07:27 INFO BlockManagerMaster: BlockManagerMaster stopped
```

Figure 7: The test set

Step 11: Create a dataset to collect the prediction parameters

The following code illustrates how to create a dataset:

```
Dataset<Row> predictions =
cvModel.transform(test);
```

Step 12: Display the prediction parameters for each text in the test set

With the help of the following code, we can display the prediction parameters:

```
for (Row r : predictions.select("id", "text",
"probability", "prediction").collect())
    {
System.out.println("(" + r.get(0) + ", " +
r.get(1) + ") --> prob=" + r.get(2) + ",
prediction=" + r.get(3));
    }
```

```
16/10/07 17:07:27 INFO Instrumentation: LogisticRegression-logreg_fecf20c19158-1728512138-61: training
16/10/07 17:07:27 INFO CodeGenerator: Code generated in 7.595764 ms
16/10/07 17:07:27 INFO CodeGenerator: Code generated in 4.648481 ms
(4, FreeMsg CALL j k) --> prob=[0.039177968591713346,0.9608220314082866], prediction=1.0
(5, Siva  hostel) --> prob=[0.022516972660610823,0.9774830273393893], prediction=1.0
(6, darren now) --> prob=[0.026564789363067196,0.9734352106369328], prediction=1.0
(7, Sunshine Quiz! Win a super Sony) --> prob=[0.08666820730759227,0.9133317926924077], prediction=1.0
16/10/07 17:07:27 INFO CodeGenerator: Code generated in 4.048974 ms
16/10/07 17:07:27 INFO CodeGenerator: Code generated in 5.99755 ms
```

Figure 8: Prediction against each text and ID

Therefore, if you compare the results shown in *Figure 6*, *The newly created label and ID from the dataset; where ID is the number of rows,* you'll find that the prediction accuracy increases from more sophisticated methods for measuring predictive accuracy that can be used to identify places where the error rate can be optimized depending on the costs of each type of error.

Model selection via training validation split

According to the API documentation provided by Spark at http://spark.apache.org/docs/latest/ml-guide.html, Spark also offers a `TrainValidationSplit` for hyperparameter tuning along with the `CrossValidator`. The idea of the `TrainValidationSplit` is it only evaluates each combination of the parameters compared to cross-validation that iterates to k times. It is, therefore, computationally less expensive and produces the result more quickly. The results, however, will not be as reliable as the `CrossValidator`. There is an exception: if the training dataset is sufficiently large then it can also produce reliable results.

The theory behind the `TrainValidationSplit` is that it takes the following three as inputs:

- An Estimator
- A set of `ParamMaps` provided in the `estimatorParamMaps` parameter
- An Evaluator

Consequently, it begins the model selection by splitting the dataset into two parts using the `trainRatio` parameter. The `trainRatio` parameter, on the other hand, is used for separate training and test datasets.

For example, with *trainRatio = 0.75* (the default value is also 0.75), the `TrainValidationSplit` algorithm generates a training and testing pair. In that case, 75% of the total data is used for training the model. Consequently, the rest of the 25% is used as the validation set.

Similar to the `CrossValidator`, the `TrainValidationSplit` also iterates through a set of ParamMaps as mentioned earlier. For each combination of the parameters, it trains the given Estimator in each iteration.

Consequently, the model is evaluated using the given Evaluator. After that, the best model is selected as the best option, since the `ParamMap` produces

the best evaluation metric and thereby eases the model selection. The `TrainValidationSplit` finally fits the Estimator using the best available `ParamMap` and for the entire dataset.

Linear regression–based model selection for an OCR dataset

In this sub-section, we will show how to perform train validation split tuning for OCR data. The logistic regression will be used in the first place; then we will move forward for other models. Finally, we will recommend the most suitable parameters for the OCR data classification.

Step 1: Import necessary packages/libraries/APIs:

```
import
org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator;
import org.apache.spark.ml.param.ParamMap;
import
org.apache.spark.ml.regression.LinearRegression;
import
org.apache.spark.ml.tuning.ParamGridBuilder;
import
org.apache.spark.ml.tuning.TrainValidationSplit;
import
org.apache.spark.ml.tuning.TrainValidationSplitModel;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;
```

Step 2: Initialize necessary Spark environment

```
SparkSession spark = SparkSession
        .builder()
        .appName("TrainSplitOCR")
            .master("local[*]")
             .config("spark.sql.warehouse.dir",

               "E:/Exp/")
            .getOrCreate();
```

Here we set the application name as `TrainValidationSplit`, the master URL as `local[*]`, and the Spark Context is the entry point of the program. Please set these parameters accordingly.

Step 3: Prepare the OCR data as a libsvm format

If you recall *Figure 19* in Chapter 6, *Building Scalable Machine Learning Pipelines,* you will remember the data as follows in *Figure 9*, *A snapshot of the original OCR dataset as a Data Frame*:

letter	xbox	ybox	width	height	onpix	xbar	ybar	x2bar	y2bar	xybar	x2ybar	xy2bar	xedge	xedgey	yedge	yedgex
T	2	8	3	5	1	8	13	0	6	6	10	8	0	8	0	8
I	5	12	3	7	2	10	5	5	4	13	3	9	2	8	4	10
D	4	11	6	8	6	10	6	2	6	10	3	7	3	7	3	9
N	7	11	6	6	3	5	9	4	6	4	4	10	6	10	2	8
G	2	1	3	1	1	8	6	6	6	6	5	9	1	7	5	10
S	3	2	3	3	2	8	8	7	5	7	5	7	2	8	9	8
O	6	11	7	8	5	7	6	9	6	7	5	9	4	8	5	5
J	3	6	4	4	2	6	6	4	4	14	8	12	1	6	1	6
C	6	11	7	8	3	7	8	7	11	4	7	14	1	7	4	8
M	7	11	11	8	9	3	8	4	5	10	11	10	10	9	5	7
W	12	14	12	8	5	9	10	4	3	5	10	7	10	12	2	6
H	6	9	8	7	6	8	6	6	7	7	7	9	6	8	4	8
G	3	6	4	4	2	6	6	5	5	6	6	9	2	8	4	8
L	2	3	3	4	1	0	1	5	6	0	0	6	0	8	0	8
L	1	3	3	1	1	6	4	1	7	8	3	10	0	7	2	9
X	8	12	8	6	4	3	10	4	7	12	11	9	3	7	3	4
B	5	9	7	7	10	9	8	4	4	6	8	6	6	11	8	7
M	6	9	9	7	6	5	6	3	5	10	9	9	8	5	2	7
G	4	7	6	5	3	6	6	6	8	6	5	9	3	10	4	8
O	4	7	5	5	3	7	7	8	6	7	6	8	3	8	3	8

only showing top 20 rows

Figure 9: A snapshot of the original OCR dataset as a Data Frame

However, the current implementation of the `TrainValidationSplitModel` API only works on datasets that are already in `libsvm` format.

Tip

Interested readers should refer to the following research article for more in depth knowledge: Chih-Chung Chang and Chih-Jen Lin, *LIBSVM - A Library for Support Vector Machines*. ACM Transactions on Intelligent Systems and Technology, 2:27:1--27:27, 2011. The software is available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

Therefore, we do need to convert the dataset from the current tab separated OCR data to a `libsvm` format.

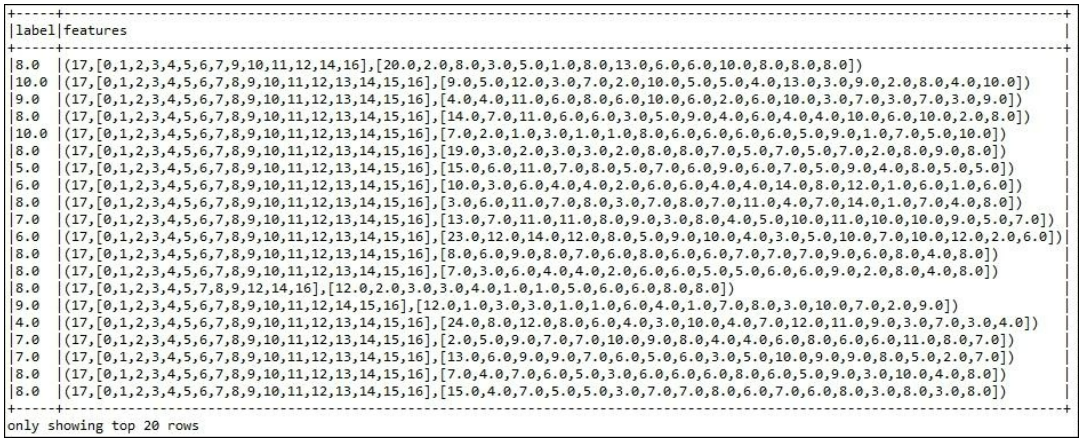
Tip

Readers should use the dataset provided with Packt packages or can convert the CSV/CSV file to the corresponding `libsvm` format.
Interested readers can refer to our public script provided on GitHub at https://github.com/rezacsedu/CSVtoLibSVMConverterinR that directly converts a CSV file to `libsvm` format. Just properly show the input and output file path and run the script on your RStudio.

Step 4: Prepare the OCR data set and also prepare the training and test set

We are assuming that readers already have downloaded the data or have converted the OCR data using our GitHub script or using their own script. Now, take the OCR `libsvm` format data as input and prepare the Dataset out of the data as raw texts and check if the data was read properly by calling the `show()` method as follows:

```
Dataset<Row> data =
spark.read().format("libsvm").load("input/Letterdata_libsvm.data");
data.show(false);
```



```
+-----+--------------------------------------------------------------------------------------------------------------------------------------------------------+
|label|features                                                                                                                                                |
+-----+--------------------------------------------------------------------------------------------------------------------------------------------------------+
|8.0  |(17,[0,1,2,3,4,5,6,7,9,10,11,12,14,16],[20.0,2.0,8.0,3.0,5.0,1.0,8.0,13.0,6.0,6.0,10.0,8.0,8.0,8.0])                                                   |
|10.0 |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[9.0,5.0,12.0,3.0,7.0,2.0,10.0,5.0,5.0,4.0,13.0,3.0,9.0,2.0,8.0,4.0,10.0])                             |
|9.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[4.0,4.0,11.0,6.0,8.0,6.0,10.0,6.0,2.0,6.0,10.0,3.0,7.0,3.0,7.0,3.0,9.0])                               |
|8.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[14.0,7.0,11.0,6.0,6.0,3.0,5.0,9.0,4.0,6.0,4.0,4.0,10.0,6.0,10.0,2.0,8.0])                             |
|10.0 |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[7.0,2.0,1.0,3.0,1.0,1.0,8.0,6.0,6.0,6.0,6.0,5.0,9.0,1.0,7.0,5.0,10.0])                               |
|8.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[19.0,3.0,2.0,3.0,3.0,2.0,8.0,8.0,7.0,5.0,7.0,5.0,7.0,2.0,8.0,9.0,8.0])                               |
|5.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[15.0,6.0,11.0,7.0,8.0,5.0,7.0,6.0,9.0,6.0,7.0,5.0,9.0,4.0,8.0,5.0,5.0])                               |
|6.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[10.0,3.0,6.0,4.0,4.0,2.0,6.0,6.0,4.0,4.0,14.0,8.0,12.0,1.0,6.0,1.0,6.0])                              |
|8.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[3.0,6.0,11.0,7.0,8.0,3.0,7.0,8.0,7.0,11.0,4.0,7.0,14.0,1.0,7.0,4.0,8.0])                              |
|7.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[13.0,7.0,11.0,11.0,8.0,9.0,3.0,8.0,4.0,5.0,10.0,11.0,10.0,10.0,9.0,5.0,7.0])                          |
|6.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[23.0,12.0,14.0,12.0,8.0,5.0,9.0,10.0,4.0,3.0,5.0,10.0,7.0,10.0,12.0,2.0,6.0])                         |
|8.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[8.0,6.0,9.0,8.0,7.0,6.0,8.0,6.0,6.0,7.0,7.0,7.0,9.0,6.0,8.0,4.0,8.0])                                |
|8.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[7.0,3.0,6.0,4.0,4.0,2.0,6.0,6.0,5.0,5.0,6.0,6.0,9.0,2.0,8.0,4.0,8.0])                                |
|8.0  |(17,[0,1,2,3,4,5,7,8,9,12,14,16],[12.0,2.0,3.0,3.0,4.0,1.0,1.0,5.0,6.0,6.0,8.0,8.0])                                                                  |
|9.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,14,15,16],[12.0,1.0,3.0,3.0,1.0,1.0,6.0,4.0,1.0,7.0,8.0,3.0,10.0,7.0,2.0,9.0])                                      |
|4.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[24.0,8.0,12.0,8.0,6.0,4.0,3.0,10.0,4.0,7.0,12.0,11.0,9.0,3.0,7.0,3.0,4.0])                            |
|7.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[2.0,5.0,9.0,7.0,7.0,10.0,9.0,8.0,4.0,4.0,6.0,8.0,6.0,6.0,11.0,8.0,7.0])                               |
|7.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[13.0,6.0,9.0,9.0,7.0,6.0,5.0,6.0,3.0,5.0,10.0,9.0,9.0,8.0,5.0,2.0,7.0])                               |
|8.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[7.0,4.0,7.0,6.0,5.0,3.0,6.0,6.0,6.0,8.0,6.0,5.0,9.0,3.0,10.0,4.0,8.0])                                |
|8.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[15.0,4.0,7.0,5.0,5.0,3.0,7.0,7.0,8.0,6.0,7.0,6.0,8.0,3.0,8.0,3.0,8.0])                                |
+-----+--------------------------------------------------------------------------------------------------------------------------------------------------------+
only showing top 20 rows
```

Figure 10: The top 20 rows

```
// Prepare training and test data.
Dataset<Row>[] splits = data.randomSplit(new
double[] {0.9, 0.1}, 12345);
Dataset<Row> training = splits[0];
Dataset<Row> test = splits[1];
```

To learn more about the data, please refer to the section *Pipeline - An Example with Spark ML*, in Chapter 4, *Extracting Knowledge through Feature Engineering*, and the description of dataset exploration. As you can see in *Figure 2*, *Spark ML pipeline model using logistic regression estimator (dash lines only happen during pipeline fitting)*, there are only two labels (spam or ham) associated along with the text values (that is, each row). However, there is no numeric label or ID.

Therefore, we need to prepare the dataset (training set) so that the Dataset also contains the ID, and labels along with text (that is, value) so that we can prepare a test set and predict their corresponding labels for any classification algorithm (that is, logistic regression) and can decide if our model selection is appropriate.

However, in order to do so we first need to prepare the training dataset. As you can see, the data frame shown above has only one column and as previously mentioned we do need to have three columns. If we prepare an

RDD from the above Dataset (that is, `df`) it would be easier for us to make that transformation.

Step 5: Configure an ML pipeline using linear regression

```
LinearRegression lr = new LinearRegression();
```

Step 6: Construct a grid of parameters to search over

Currently, Spark uses a `ParamGridBuilder` to construct a grid of parameters to search over. In this regard, then, with three values for `hashingTF.numFeatures` and two values for `lr.regParam`, this grid will have 3 x 2 = 6 parameter settings for the `CrossValidator` to choose from:

```
ParamMap[] paramGrid = new ParamGridBuilder()
.addGrid(hashingTF.numFeatures(), new int[] { 10,
100, 1000 }).addGrid(lr.regParam(), new double[]
{ 0.1, 0.01 })
.build();
```

We now treat the pipeline as an Estimator, wrapping it in a `CrossValidator` instance. This will allow us to jointly choose parameters for all pipeline stages. As already discussed, a `CrossValidator` requires an Estimator, a set of Estimator `ParamMaps`, and an Evaluator.

> **Note**
>
> Note that the evaluator here is a `BinaryClassificationEvaluator` and its default metric is `areaUnderROC`.

Step 7: Create a TrainValidationSplit instance:

```
TrainValidationSplit trainValidationSplit = new
TrainValidationSplit()
    .setEstimator(lr)
    .setEvaluator(new
MulticlassClassificationEvaluator())
    .setEstimatorParamMaps(paramGrid)
    .setTrainRatio(0.7);
```

In this case the estimator is simply the linear regression that we created in *Step 4*. A `TrainValidationSplit` requires an Estimator, a set of Estimator `ParamMaps`, and an Evaluator. In this case, 70% of the data will be used as training and the remaining 30% for the validation.

Step 8: Run TrainValidationSplit, and chooses parameters

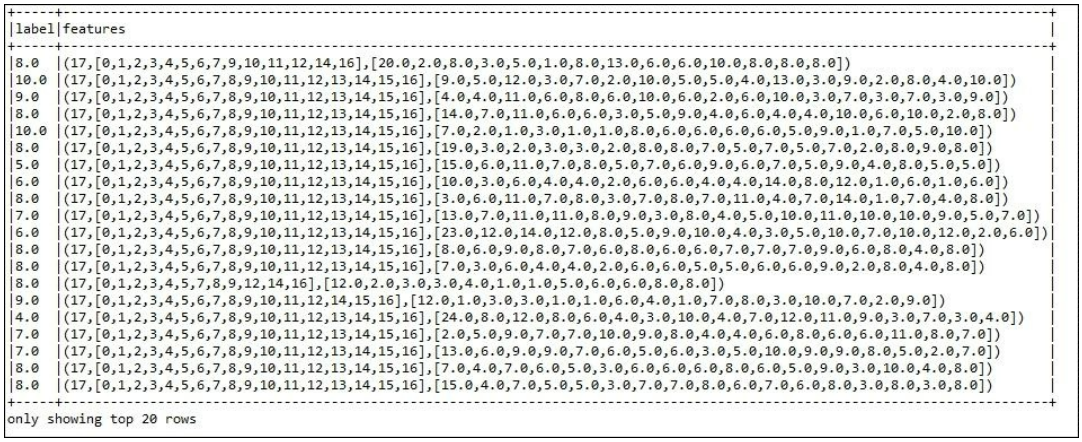
Run the `TrainValidationSplit` and choose the best set of parameters for your problem using the training set. Just use the following code segment:

```
TrainValidationSplitModel model =
trainValidationSplit.fit(training);
```

Step 9: Making a prediction on the test set

Make predictions on the test data where model is the model with the combination of parameters that performed best. Finally, to show the predictions, use the following code segment:

```
Dataset<Row> per_param = model.transform(test);
per_param.show(false);
```



```
+-----+--------------------------------------------------------------------------------------------------------------------------------------------------------+
|label|features                                                                                                                                                |
+-----+--------------------------------------------------------------------------------------------------------------------------------------------------------+
|8.0  |(17,[0,1,2,3,4,5,6,7,9,10,11,12,14,16],[20.0,2.0,8.0,3.0,5.0,1.0,8.0,13.0,6.0,6.0,10.0,8.0,8.0,8.0])                                                   |
|10.0 |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[9.0,5.0,12.0,3.0,7.0,2.0,10.0,5.0,5.0,4.0,13.0,3.0,9.0,2.0,8.0,4.0,10.0])                             |
|9.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[4.0,4.0,11.0,6.0,8.0,6.0,10.0,6.0,2.0,6.0,10.0,3.0,7.0,3.0,7.0,3.0,9.0])                             |
|8.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[14.0,7.0,11.0,6.0,6.0,3.0,5.0,9.0,4.0,6.0,4.0,4.0,10.0,6.0,10.0,2.0,8.0])                            |
|10.0 |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[7.0,2.0,1.0,3.0,1.0,1.0,8.0,6.0,6.0,6.0,6.0,5.0,9.0,1.0,7.0,5.0,10.0])                               |
|8.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[19.0,3.0,2.0,3.0,3.0,2.0,8.0,8.0,7.0,5.0,7.0,5.0,7.0,2.0,8.0,9.0,8.0])                               |
|5.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[15.0,6.0,11.0,7.0,8.0,5.0,7.0,6.0,9.0,6.0,7.0,5.0,9.0,4.0,8.0,5.0,5.0])                               |
|6.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[10.0,3.0,6.0,4.0,4.0,2.0,6.0,6.0,4.0,4.0,14.0,8.0,12.0,1.0,6.0,1.0,6.0])                             |
|8.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[3.0,6.0,11.0,7.0,8.0,3.0,7.0,8.0,7.0,11.0,4.0,7.0,14.0,1.0,7.0,4.0,8.0])                              |
|7.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[13.0,7.0,11.0,11.0,8.0,9.0,3.0,8.0,4.0,5.0,10.0,11.0,10.0,10.0,9.0,5.0,7.0])                          |
|6.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[23.0,12.0,14.0,12.0,8.0,5.0,9.0,10.0,4.0,3.0,5.0,10.0,7.0,10.0,12.0,2.0,6.0])                          |
|8.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[8.0,6.0,9.0,8.0,7.0,6.0,8.0,6.0,6.0,7.0,7.0,7.0,9.0,6.0,8.0,4.0,8.0])                                |
|8.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[7.0,3.0,6.0,4.0,4.0,2.0,6.0,6.0,5.0,5.0,6.0,6.0,9.0,2.0,8.0,4.0,8.0])                                |
|8.0  |(17,[0,1,2,3,4,5,7,8,9,12,14,16],[12.0,2.0,3.0,3.0,4.0,1.0,1.0,5.0,6.0,6.0,8.0,8.0])                                                                  |
|9.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,14,15,16],[12.0,1.0,3.0,3.0,1.0,1.0,6.0,4.0,1.0,7.0,8.0,3.0,10.0,7.0,2.0,9.0])                                      |
|4.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[24.0,8.0,12.0,8.0,6.0,4.0,3.0,10.0,4.0,7.0,12.0,11.0,9.0,3.0,7.0,3.0,4.0])                            |
|7.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[2.0,5.0,9.0,7.0,7.0,10.0,9.0,8.0,4.0,4.0,6.0,8.0,6.0,6.0,11.0,8.0,7.0])                               |
|7.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[13.0,6.0,9.0,9.0,7.0,6.0,5.0,6.0,3.0,5.0,10.0,9.0,9.0,8.0,5.0,2.0,7.0])                               |
|8.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[7.0,4.0,7.0,6.0,5.0,3.0,6.0,6.0,6.0,8.0,6.0,5.0,9.0,3.0,10.0,4.0,8.0])                                |
|8.0  |(17,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[15.0,4.0,7.0,5.0,5.0,3.0,7.0,7.0,8.0,6.0,7.0,6.0,8.0,3.0,8.0,3.0,8.0])                                |
+-----+--------------------------------------------------------------------------------------------------------------------------------------------------------+
only showing top 20 rows
```

Figure 11: Prediction against each feature and label

In *Figure 11*, we showed the row prediction against the actual label. The first column is the actual label, the second column signifies the feature vector, and the third column shows the raw prediction based on the feature vectors the `TrainValidationSplitModel` created.

Logistic regression-based model selection for the cancer dataset

In this sub-section, we will show how to perform train validation split tuning for OCR data. We will use the logistic regression in the first place; then we will move forward for other models. Finally, we will recommend the most suitable parameters for the OCR data classification.

Step 1: Import the necessary packages/libraries/APIs

```
import org.apache.spark.api.java.JavaRDD;
import
org.apache.spark.api.java.function.Function;
```

```
import
org.apache.spark.ml.classification.LogisticRegression;
import
org.apache.spark.ml.evaluation.RegressionEvaluator;
import org.apache.spark.ml.feature.LabeledPoint;
import org.apache.spark.ml.linalg.DenseVector;
import org.apache.spark.ml.linalg.Vector;
import org.apache.spark.ml.param.ParamMap;
import
org.apache.spark.ml.tuning.ParamGridBuilder;
import
org.apache.spark.ml.tuning.TrainValidationSplit;
import
org.apache.spark.ml.tuning.TrainValidationSplitModel;
import org.apache.spark.rdd.RDD;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;
```

Step 2: Initialize the necessary Spark environment

```
static SparkSession spark = SparkSession
  .builder()
  .appName("CrossValidationforSpamFiltering")
  .master("local[*]")
  .config("spark.sql.warehouse.dir", "C:/Exp/").
  getOrCreate();
```

Here we set the application name as `CancerDiagnosis`, the master URL as `local[*]` and the Spark Context as the entry point of the program. Please set these parameters accordingly.

Step 3: Create the Java RDD

Parse the cancer diagnosis data and prepare the Java RDDs for strings:

```
String path = "breastcancer/input/wdbc.data";
RDD<String> lines =
spark.sparkContext().textFile(path, 3);
```

Step 4: Prepare the cancer diagnosis LabeledPoint RDDs

As already discussed in Chapter 6, *Building Scalable Machine Learning Pipelines*, the cancer diagnosis dataset contains two labels *B* and *M* for Benign and Malignant. However, we need to convert them into a numeric label. Just use the following code to convert all of them from label transforming to `LabeledPoint` RDDs preparation:

```
JavaRDD<LabeledPoint> linesRDD =
lines.toJavaRDD().map(new Function<String,
LabeledPoint>() {
```

```
  public LabeledPoint call(String lines) {
  String[] tokens = lines.split(",");
  double[] features = new double[30];
  for (int i = 2; i < features.length; i++) {
      features[i - 2] =
             Double.parseDouble(tokens[i]);
    }
       Vector v = new DenseVector(features);
       if (tokens[1].equals("B")) {
  return new LabeledPoint(1.0, v); // benign
} else {
return new LabeledPoint(0.0, v); // malignant
}
  }
});
```

```
1  (0.0,[17.99,10.38,122.8,1001.0,0.1184,0.2776,0.3001,0.1471,0.2419,0.07871,1
2  (0.0,[20.57,17.77,132.9,1326.0,0.08474,0.07864,0.0869,0.07017,0.1812,0.0566
3  (0.0,[19.69,21.25,130.0,1203.0,0.1096,0.1599,0.1974,0.1279,0.2069,0.05999,0
4  (0.0,[11.42,20.38,77.58,386.1,0.1425,0.2839,0.2414,0.1052,0.2597,0.09744,0.
5  (0.0,[20.29,14.34,135.1,1297.0,0.1003,0.1328,0.198,0.1043,0.1809,0.05883,0.
6  (0.0,[12.45,15.7,82.57,477.1,0.1278,0.17,0.1578,0.08089,0.2087,0.07613,0.33
7  (0.0,[18.25,19.98,119.6,1040.0,0.09463,0.109,0.1127,0.074,0.1794,0.05742,0.
8  (0.0,[13.71,20.83,90.2,577.9,0.1189,0.1645,0.09366,0.05985,0.2196,0.07451,0
9  (0.0,[13.0,21.82,87.5,519.8,0.1273,0.1932,0.1859,0.09353,0.235,0.07389,0.30
10 (0.0,[12.46,24.04,83.97,475.9,0.1186,0.2396,0.2273,0.08543,0.203,0.08243,0.
11 (0.0,[16.02,23.24,102.7,797.8,0.08206,0.06669,0.03299,0.03323,0.1528,0.0569
12 (0.0,[15.78,17.89,103.6,781.0,0.0971,0.1292,0.09954,0.06606,0.1842,0.06082,
13 (0.0,[19.17,24.8,132.4,1123.0,0.0974,0.2458,0.2065,0.1118,0.2397,0.078,0.95
```

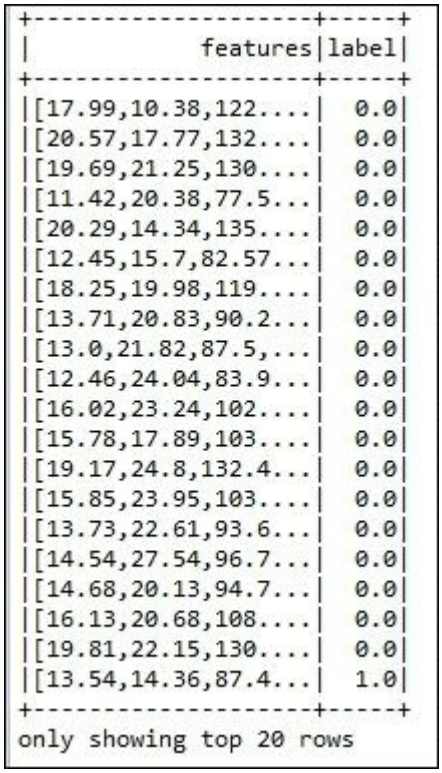
Figure 12: The Label Point RDDs snapshot

As you can see in *Figure 9*, *The top 20 rows*, the labels B and M have been converted into 1.0 and 0.0. Now we need to create a data frame out of the label point RDDs.

Step 5: Create a Dataset and also prepare the training and test set

Create a Dataset from the previous RDDs (that is, `linesRDD`) by specifying the Label Point class:

```
Dataset<Row> data =
spark.sqlContext().createDataFrame(linesRDD,
LabeledPoint.class);
data.show();
```



```
+--------------------+-----+
|            features|label|
+--------------------+-----+
|[17.99,10.38,122....|  0.0|
|[20.57,17.77,132....|  0.0|
|[19.69,21.25,130....|  0.0|
|[11.42,20.38,77.5...|  0.0|
|[20.29,14.34,135....|  0.0|
|[12.45,15.7,82.57...|  0.0|
|[18.25,19.98,119....|  0.0|
|[13.71,20.83,90.2...|  0.0|
|[13.0,21.82,87.5,...|  0.0|
|[12.46,24.04,83.9...|  0.0|
|[16.02,23.24,102....|  0.0|
|[15.78,17.89,103....|  0.0|
|[19.17,24.8,132.4...|  0.0|
|[15.85,23.95,103....|  0.0|
|[13.73,22.61,93.6...|  0.0|
|[14.54,27.54,96.7...|  0.0|
|[14.68,20.13,94.7...|  0.0|
|[16.13,20.68,108....|  0.0|
|[19.81,22.15,130....|  0.0|
|[13.54,14.36,87.4...|  1.0|
+--------------------+-----+
only showing top 20 rows
```

Figure 13: The created Dataset showing the top 20 rows.

```
Dataset<Row>[] splits=data.randomSplit(new
double[] {0.8, 0.2});
Dataset<Row> training = splits[0];
Dataset<Row> test = splits[1];
```

Note, that you will have to set the ratio of the random split based on your data and problem type accordingly.

Step 6: Configure an ML pipeline using logistic regression:

```
LogisticRegression lr = new LogisticRegression();
```

Step 7: Construct a grid of parameters to search over

Currently, Spark uses a `ParamGridBuilder` to construct a grid of parameters to search over. In this regard, suppose we have three values for `hashingTF.numFeatures` and two values for `lr.regParam`, this grid will have 3 x 2 = 6 parameter settings for `CrossValidator` to choose from:

```
ParamMap[] paramGrid = new ParamGridBuilder()
.addGrid(lr.regParam(), new double[] {0.1, 0.01})
.addGrid(lr.fitIntercept())
.addGrid(lr.elasticNetParam(), new double[] {0.0,
0.5, 1.0})
.build();
```

Note, that you will have to set the values of above parameters based on your data and problem type accordingly.

Step 8: Create a TrainValidationSplit instance:

```
TrainValidationSplit trainValidationSplit = new
TrainValidationSplit()
.setEstimator(lr)
.setEvaluator(new RegressionEvaluator())
.setEstimatorParamMaps(paramGrid)
.setTrainRatio(0.8);
```

In this case, the estimator is simply the linear regression that we created in *Step 4*. Prepare the cancer diagnosis `LabeledPoint` RDDs. A `TrainValidationSplit` requires an Estimator, a set of Estimator `ParamMaps`, and an Evaluator that supports binary classification since our dataset has only two classes where 80% is used for the purpose of training and the remaining 20% for the validation.

Step 9: Run the TrainValidationSplit and choose the parameters

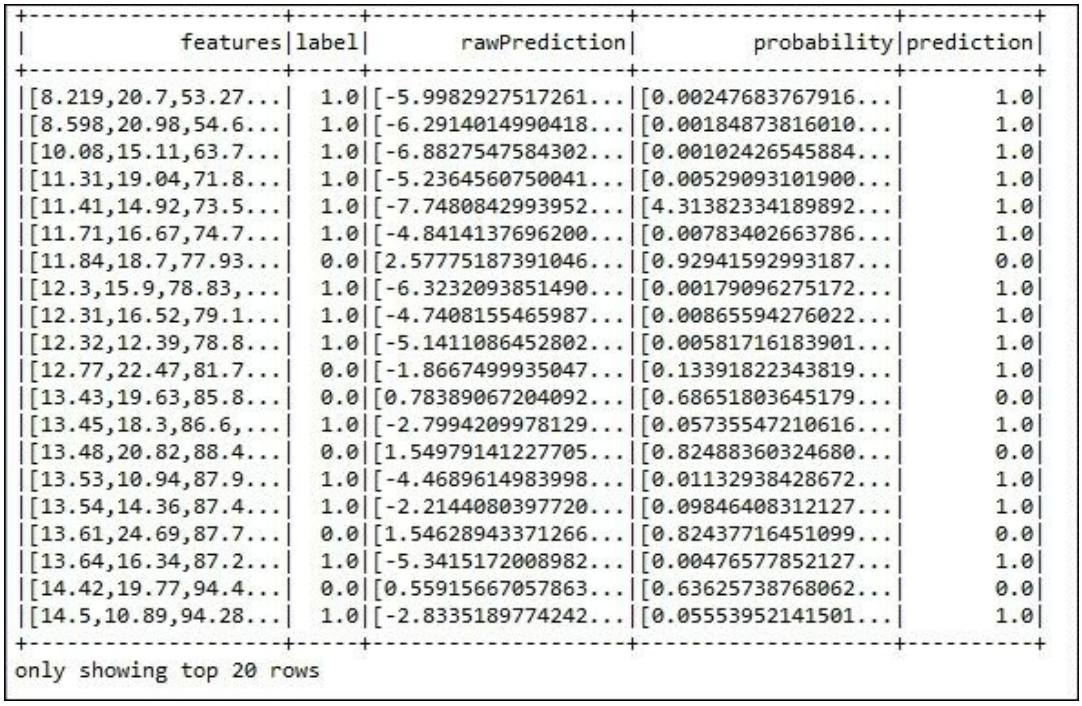
Run the `TrainValidationSplit`, and choose the best set of parameters for your problem using the training set. Just use the following code segments:

```
TrainValidationSplitModel model =
trainValidationSplit.fit(training);
```

Step 10: Make predictions on the test set

Make predictions on the test data where the model is the model with a combination of parameters that performed the best. Finally, show the predictions. Just use the following code segment for doing so:

```
Dataset<Row> per_param = model.transform(test);
per_param.show();
```



features	label	rawPrediction	probability	prediction
[8.219,20.7,53.27...	1.0	[-5.9982927517261...	[0.00247683767916...	1.0
[8.598,20.98,54.6...	1.0	[-6.2914014990418...	[0.00184873816010...	1.0
[10.08,15.11,63.7...	1.0	[-6.8827547584302...	[0.00102426545884...	1.0
[11.31,19.04,71.8...	1.0	[-5.2364560750041...	[0.00529093101900...	1.0
[11.41,14.92,73.5...	1.0	[-7.7480842993952...	[4.31382334189892...	1.0
[11.71,16.67,74.7...	1.0	[-4.8414137696200...	[0.00783402663786...	1.0
[11.84,18.7,77.93...	0.0	[2.57775187391046...	[0.92941592993187...	0.0
[12.3,15.9,78.83,...	1.0	[-6.3232093851490...	[0.00179096275172...	1.0
[12.31,16.52,79.1...	1.0	[-4.7408155465987...	[0.00865594276022...	1.0
[12.32,12.39,78.8...	1.0	[-5.1411086452802...	[0.00581716183901...	1.0
[12.77,22.47,81.7...	0.0	[-1.8667499935047...	[0.13391822343819...	1.0
[13.43,19.63,85.8...	0.0	[0.78389067204092...	[0.68651803645179...	0.0
[13.45,18.3,86.6,...	1.0	[-2.7994209978129...	[0.05735547210616...	1.0
[13.48,20.82,88.4...	0.0	[1.54979141227705...	[0.82488360324680...	0.0
[13.53,10.94,87.9...	1.0	[-4.4689614983998...	[0.01132938428672...	1.0
[13.54,14.36,87.4...	1.0	[-2.2144080397720...	[0.09846408312127...	1.0
[13.61,24.69,87.7...	0.0	[1.54628943371266...	[0.82437716451099...	0.0
[13.64,16.34,87.2...	1.0	[-5.3415172008982...	[0.00476577852127...	1.0
[14.42,19.77,94.4...	0.0	[0.55915667057863...	[0.63625738768062...	0.0
[14.5,10.89,94.28...	1.0	[-2.8335189774242...	[0.05553952141501...	1.0

only showing top 20 rows

Figure 14: Prediction against each feature and label

Therefore, if you compare these results with those shown in *Figure 6, The newly created label and ID from the dataset; where ID is the number of rows*, you find that the prediction accuracy increases for more sophisticated methods of measuring predictive accuracy that can be used to identify places where the error rate can be optimized depending on the costs of each type of error.

Summary

Tuning an algorithm or machine learning application can be thought of as simply a process through which one goes when they optimise the parameters that impact the model in order to enable the algorithm to perform the best (in terms of run-time and memory usages). In this chapter, we have shown how to perform ML model tuning using train-validation split and cross-validation techniques of Spark ML.

We also want to mention that the tuning related support and algorithms are still not well enriched until the date of (14th October 2016) the current release of Spark. Interested readers are encouraged to visit the Spark tuning page at http://spark.apache.org/docs/latest/ml-tuning.html for more updates since we believe that more features will be added to the Spark website and they will certainly provide enough documentation.

In the next chapter, we will discuss how to make your machine learning algorithm or models adaptable for new datasets. This chapter covers advanced machine learning techniques to be able to make algorithms adaptable to new data. It will mainly focus on batch/streaming architectures and on online learning algorithms by using Spark streaming.

The ultimate target is to bring dynamism to the static machine learning models. Readers will also see how machine learning algorithms learn incrementally over the data; that is to say the models are updated each time they see a new training instance.

Chapter 8. Adapting Your Machine Learning Models

This chapter covers advanced **machine learning** (**ML**) techniques in order to be able to make algorithms adaptable to new data. Readers will also see how machine learning algorithms learn incrementally over the data, that is to say that the models are updated each time they see a new training instance. Learning in dynamic environments by conceding different constraints will also be discussed. In summary, the following topics will be covered in this chapter:

- Adapting machine learning models
- Generalization of ML models
- Adapting through incremental algorithms
- Adapting through reusing ML models
- Machine learning in dynamic environments

Adapting machine learning models

As we discussed earlier, as a part of the ML training process, a model is trained using a set of data (that is, training, test, and validation set). Machine learning models that can adapt to their environments and learn from their experience have attracted consumers and researchers from diverse areas, including computer science, engineering, mathematics, physics, neuroscience, and cognitive science. In this section, we will provide a technical overview of how to adopt machine learning models for the new data and requirements.

Technical overview

Technically, the same models might need to be retrained at a later stage if required for the betterment. This is really dependent on several factors, for example, when the new data becomes available, or when the consumer of the API has their own data to train the model or when the data needs to be filtered and the model trained with the subset of data. In these scenarios, ML algorithms should provide enough APIs to provide a convenient way to allow its consumers to produce a client that can be used on a one-time or regular basis, so that they can retrain the model using their own data.

As a result, the client will be able to evaluate the results of retraining and updating the web service API accordingly. Alternatively, they will be able to use the newly trained model. In this regard, there are several contexts of

domain adaptation. However, they differ in the information considered for application type and requirements:

- **Unsupervised domain adaptation**: The learning sample contains a set of labeled source examples, a set of unlabeled source examples, and an unlabeled set of target examples
- **Semi-supervised domain adaptation**: In this situation, we also consider a small set of labeled target examples
- **Supervised domain adaptation**: All the examples considered are supposed to be labeled:

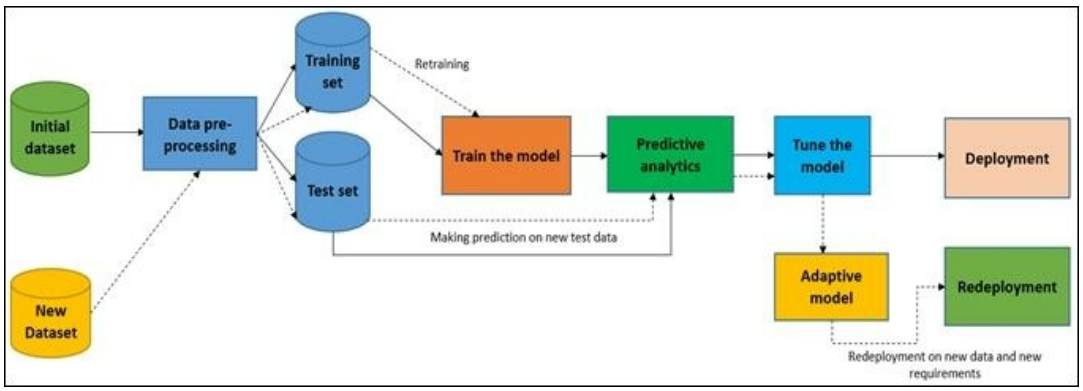


Figure 1: The retraining process overview (dashed line presents the retrain steps)

Technically, there should be three alternatives for making the ML models adaptable:

- The most widely used machine learning techniques and algorithms include decision trees, decision rules, neural networks, statistical classifiers, and probabilistic graphical models, and they all need to be developed so that they can be adaptable for the new requirements
- Secondly, previous mentioned algorithms or techniques should be generalized so that they can be used with minimum effort
- Moreover, more robust theoretical frameworks and algorithms such as Bayesian learning theory, classical statistical theory, minimum description length theory, and statistical mechanics approaches need to be developed in order to understand computational learning theory

The benefits from these three adaptation properties and techniques will provide insight into experimental results that will also guide machine learning communities to contribute towards the different learning algorithms.

The generalization of ML models

In Chapter 5, *Supervised and Unsupervised Learning by Example*, we discussed how and why to generalize the learning algorithm to fit semi-supervised learning, active learning, structured prediction, and reinforcement learning. In this section, we will discuss how to generalize the linear regression algorithm on the **Optical Character Recognition** (**OCR**) dataset to show an example of generalization of the linear regression model.

Generalized linear regression

As discussed in Chapter 5, *Supervised and Unsupervised Learning by Example*, the linear regression and logistic regression techniques assume that the output follows a Gaussian distribution. The **generalized linear models** (**GLMs**) on the other hand are specifications of the linear models where the response variable, that is, Yi, follows the linear distribution from an exponential family of distribution.

Spark's `GeneralizedLinearRegression` API allows us a flexible specification of the GLMs. The current implementation of the generalized linear regression in Spark can be used for numerous types of prediction problems, for example, linear regression, Poisson regression, logistic regression, and others.

However, the limitation is that only a subset of the exponential family distributions is supported in the current implementation of the Spark-based GLM algorithm. In addition, there is another scalability issue that only 4096 features are supported through its `GeneralizedLinearRegression` API. Consequently, if the number of features are beyond 4096, the algorithm will throw an exception.

Fortunately, your models with an increased number of features can be trained using the LinearRegression and LogisticRegression estimators, as shown in several examples in Chapter 6, *Building Scalable Machine Learning Pipelines*.

Generalized linear regression with Spark

In this sub-section, we discuss a step-by-step example that shows how to apply the generalized linear regression on the `libsvm` version of the **Optical Character Recognition** (**OCR**) data that we discussed in Chapter 7, *Tuning Machine Learning Models*. Since the same dataset will be re-used here, we've decided not to describe them further.

Step 1: Load the necessary API and packages

Here is the code to load the necessary API and packages:

```
import java.util.Arrays;
import
org.apache.spark.ml.regression.GeneralizedLinearRegression;
import
org.apache.spark.ml.regression.GeneralizedLinearRegressionModel;
import
org.apache.spark.ml.regression.GeneralizedLinearRegressionTrainingSummary
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;
```

Step 2: Create the Spark session

The following code shows how to create the Spark session:

```
SparkSession spark = SparkSession
    .builder()

 .appName("JavaGeneralizedLinearRegressionExample")
    .master("local[*]")
    .config("spark.sql.warehouse.dir", "E:/Exp/")
    .getOrCreate();
```

Step 3: Load and create the Dataset

Load and create the dataset from the OCR dataset. Here we have specified the dataset format as `libsvm`:

```
Dataset<Row>dataset =
spark.read().format("libsvm").load("input/Letterdata_libsvm.data");
```

Step 4: Prepare the training and test sets

The following code illustrates how to prepare the training and test sets:

```
double[] weights = {0.8, 0.2};
long seed = 12345L;
Dataset<Row>[] split =
dataset.randomSplit(weights, seed);
Dataset<Row> training = split[0];
Dataset<Row> test = split[1];
```

Step 5: Create a generalized linear regression estimator

Create the generalized linear regression estimator by specifying the family, link, and max iteration and regression parameters. Here we have selected the family as `"gaussian"` and link as `"identity"`:

```
GeneralizedLinearRegression glr = new
GeneralizedLinearRegression()
.setFamily("gaussian")
.setLink("identity")
.setMaxIter(10)
.setRegParam(0.3);
```

Note that according to the API documentation at http://spark.apache.org/docs/latest/ml-classification-regression.html#generalized-linear-regression, the following options are supported with this algorithm implementation with Spark:

Family	Response Type	Supported Links
Gaussian	Continuous	Identity*, Log, Inverse
Binomial	Binary	Logit*, Probit, CLogLog
Poisson	Count	Log*, Identity, Sqrt
Gamma	Continuous	Inverse*, Idenity, Log
* Canonical Link		

Figure 2: Available supported families with the current implementation of Generalized Linear Regression

Step 6: Fit the model

Here is the code to fit the model:

```
GeneralizedLinearRegressionModel model =
glr.fit(training);
```

Step 7: Check the coefficients and intercept

Print the coefficients and intercept for the linear regression model that we created in Step 6:

```
System.out.println("Coefficients: " +
model.coefficients());
System.out.println("Intercept: " +
model.intercept());
```

Output for these two parameters will be similar to the following:

```
Coefficients:
[-0.0022864381796305487,-
0.002728958263362158,0.001582003618682323,-
0.0027708788253722914,0.0021962329827476565,-
0.014769839282003813,0.027752802299957722,0.005757124632688538,0.01386944
0.010555326094498824,-0.006062727351948948,-
0.01618167221020619,0.02894330366681715,-
0.006180003317929849,-
0.0025768386348180294,0.015161831324693125,0.8125261496082304]
Intercept: 1.2140016821111255
```

It is to be noted that the `System.out.println` method will not work in cluster mode. This will work only in standalone or Pseudo mode. It is only for the verification of the result.

Step 8: Summarize the model

Summarize the model over the training set and print out some metrics:

```
GeneralizedLinearRegressionTrainingSummary
summary = model.summary();
```

Step 9: Verify some generalized metrics

Let's print some generalized metrics such as **Coefficient Standard Errors** (**CSE**), T values, P values, Dispersions, Null deviance, the Residual degree of freedom null, AIC and Deviance residuals. Owing to the page limitations, we have not shown how these values are significant or the calculations procedure:

```
System.out.println("Coefficient Standard Errors:
"
      +
Arrays.toString(summary.coefficientStandardErrors()));
System.out.println("T Values: " +
Arrays.toString(summary.tValues()));
System.out.println("P Values: " +
Arrays.toString(summary.pValues()));
System.out.println("Dispersion: " +
summary.dispersion());
System.out.println("Null Deviance: " +
summary.nullDeviance());
System.out.println("Residual Degree Of Freedom
Null: " + summary.residualDegreeOfFreedomNull());
System.out.println("Deviance: " +
summary.deviance());
System.out.println("Residual Degree Of Freedom: "
+ summary.residualDegreeOfFreedom());
    System.out.println("AIC: " + summary.aic());
```

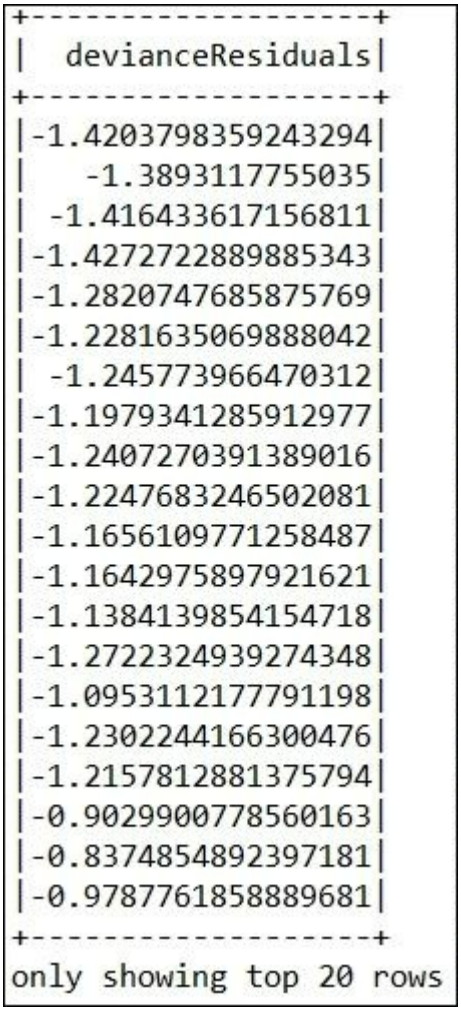
Let's see the values for the training set we created previously:

```
Coefficient Standard
Errors:[2.877963555951775E-
4, 0.0016618949921257992, 9.147115254397696E-4,
0.001633197607413805, 0.0013194682048354774,
0.001427648472211677, 0.0010797461071614422,
0.001092731825368789, 7.922778963434026E-4,
9.413717346009722E-
4, 8.746375698587989E-4, 9.768068714323967E-4,
0.0010276211138097238, 0.0011457739746946476,
0.0015025626835648176, 9.048329671989396E-4,
0.0013145697411570455, 0.02274018067510297]
T Values:[-7.944639100457261,
-
1.6420762300218703, 1.729510971146599,
-
1.6965974067032972, 1.6644834446931607,
-
10.345571455081481, 25.703081600282317,
5.2685613240426585, 17.50578259898057,
-
11.212707697212734, -6.931702411237277,
-
16.56588695621814, 28.165345454527458,
-
5.3937368577226055, -1.714962485760994,
16.756497468951743, 618.0928437414578,
53.385753589911985]
P Values:[1.9984014443252818E-15,
0.10059394323065063, 0.08373705354670546,
0.0897923347927514, 0.09603552109755675, 0.0,
0.0, 1.3928712139232857E-7, 0.0, 0.0,
4.317657342767234E-
12, 0.0, 0.0, 6.999167956323049E-8,
0.08637155105770145, 0.0, 0.0, 0.0]
Dispersion: 0.07102433332236015
Null Deviance: 41357.85510971454
Residual Degree Of Freedom Null: 15949
Deviance: 1131.5596784918419
Residual Degree Of Freedom: 15932
AIC: 3100.6418768238423
```

Step 10: Show the deviance residuals

The following code is used to show the deviance residuals:

```
summary.residuals().show();
```



```
+-------------------+
|  devianceResiduals|
+-------------------+
|-1.4203798359243294|
|   -1.3893117755035|
| -1.416433617156811|
|-1.4272722889885343|
|-1.2820747685875769|
|-1.2281635069888042|
| -1.245773966470312|
|-1.1979341285912977|
|-1.2407270391389016|
|-1.2247683246502081|
|-1.1656109771258487|
|-1.1642975897921621|
|-1.1384139854154718|
|-1.2722324939274348|
|-1.0953112177791198|
|-1.2302244166300476|
|-1.2157812881375794|
|-0.9029900778560163|
|-0.8374854892397181|
|-0.9787761858889681|
+-------------------+
only showing top 20 rows
```

Figure 3: Summary of the deviance residuals for the OCR dataset

Tip

Interested readers should refer to the following web page to get more information on and insight into this algorithm and its implementation details: http://spark.apache.org/docs/latest/ml-classification-regression.html

Adapting through incremental algorithms

According to Robi Polikar et al., (*Learn++: An Incremental Learning Algorithm for Supervised Neural Networks, IEEE Transactions on Systems, Man, And Cybernetics, V-21, No-4, November 2001*), various algorithms have been suggested for incremental learning. The incremental learning is therefore implied for solving different problems. In some literature, the term incremental learning has been used to refer to either the growing of or pruning of a classifier. Alternatively, it may refer to the selection of most informative training samples for solving a problem in an incremental way.

In other cases, making a regular ML algorithm incremental means performing some form of controlled modification of weights in the classifier, by retraining with misclassified signals. Some algorithms are capable of learning new information; however, they do not synchronously satisfy all of the previously mentioned criteria. Moreover, they either require access to the old data or need to forget the prior knowledge along the way, and as they are unable to accommodate new classes they are not adaptable for new datasets.

Considering the previously mentioned issues, in this section, we will discuss how to adopt ML models using an incremental version of the original algorithms. Incremental SVM, Bayesian Network, and Neural networks will be discussed in brief. Moreover, when applicable, we will provide regular Spark implementation of these algorithms.

Incremental support vector machine

It's pretty difficult to make a regular ML algorithm incremental. In short, it's possible but not altogether easy. If you want to do it you have to change the underlying source codes in the Spark library you are using or implement the training algorithm yourself.

Unfortunately, Spark does not have an incremental version of SVM implemented. However, before making the linear SVM incremental, you need to first understand the linear SVM itself. Therefore, we provide some concepts of linear SVMs in the next sub-section using Spark for the new dataset.

Tip

According to our knowledge, we have found only two possible solutions called SVMHeavy (http://people.eng.unimelb.edu.au/shiltona/svm/) and

LaSVM (http://leon.bottou.org/projects/lasvm), which support incremental training. But we haven't used either. Interested readers should follow these two papers on incremental SVMs to get some insight. These two papers are straightforward and show good research if you're just getting started:

http://cbcl.mit.edu/cbcl/publications/ps/cauwenberghs-nips00.pdf.

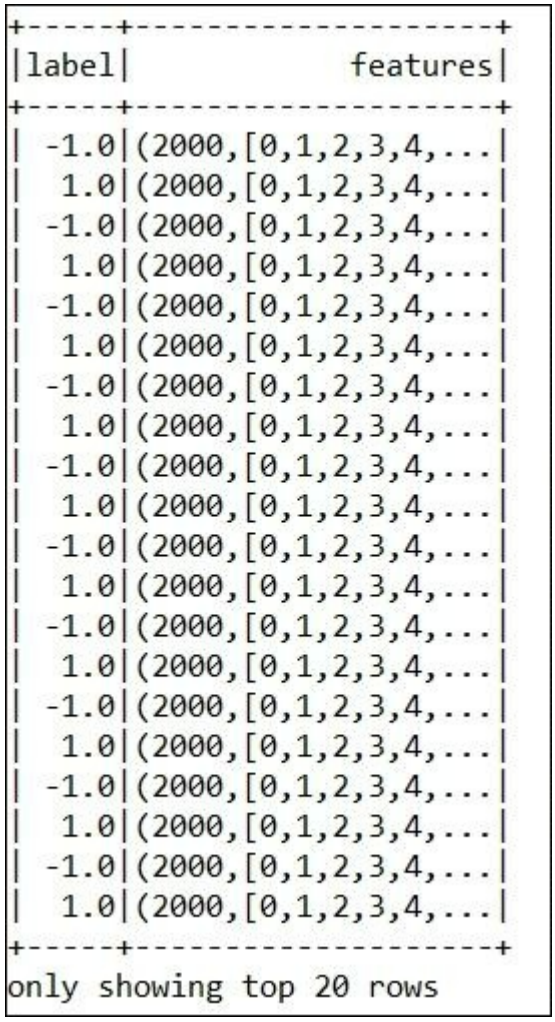
http://www.jmlr.org/papers/volume7/laskov06a/laskov06a.pdf.

Adapting SVMs for new data with Spark

In this section, we will first discuss how to perform binary classification using linear SVMs of Spark implementation. Then we will show how to adopt the same algorithm for the new data type.

Step 1: Data collection and exploration

We have collected a colon cancer dataset from https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html. Originally, the dataset was labeled as -1.0 and 1.0 as follows:



```
+-----+--------------------+
|label|            features|
+-----+--------------------+
| -1.0|(2000,[0,1,2,3,4,...|
|  1.0|(2000,[0,1,2,3,4,...|
| -1.0|(2000,[0,1,2,3,4,...|
|  1.0|(2000,[0,1,2,3,4,...|
| -1.0|(2000,[0,1,2,3,4,...|
|  1.0|(2000,[0,1,2,3,4,...|
| -1.0|(2000,[0,1,2,3,4,...|
|  1.0|(2000,[0,1,2,3,4,...|
| -1.0|(2000,[0,1,2,3,4,...|
|  1.0|(2000,[0,1,2,3,4,...|
| -1.0|(2000,[0,1,2,3,4,...|
|  1.0|(2000,[0,1,2,3,4,...|
| -1.0|(2000,[0,1,2,3,4,...|
|  1.0|(2000,[0,1,2,3,4,...|
| -1.0|(2000,[0,1,2,3,4,...|
|  1.0|(2000,[0,1,2,3,4,...|
| -1.0|(2000,[0,1,2,3,4,...|
|  1.0|(2000,[0,1,2,3,4,...|
| -1.0|(2000,[0,1,2,3,4,...|
|  1.0|(2000,[0,1,2,3,4,...|
+-----+--------------------+
only showing top 20 rows
```

Figure 4: Original colon cancer data snapshot

Tip

The dataset was used in the following publication: *U. Alon, N. Barkai, D. A. Notterman, K. Gish, S.Ybarra, D.Mack, and A. J. Levine. Broad patterns of gene expression revealed by clustering analysis of tumour and normal colon tissues probed by oligonucleotide arrays*. *Cell Biology, 96:6745-6750, 1999*. Interested readers should refer to the publication to get more insights into the dataset.

After that, instance-wise normalization is carried out to mean zero and variance one. Then feature wise normalization is carried out to get zero and variance one as a pre-processing step. However, for simplicity, we have considered -1.0 as 0.1, since SVM does not recognize symbols (that is, + or -). Therefore, the dataset now contains two labels 1 and 0 (that is, to say it's a binary classification problem). After pre-processing and scaling, there are two classes and 2000 features. Here is a sample of the dataset in *Figure 5*:

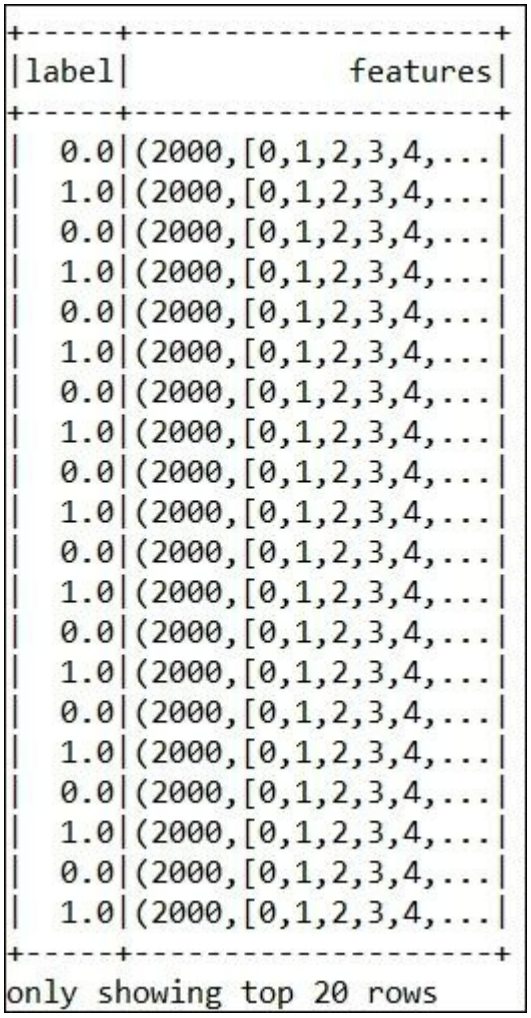


Figure 5: Pre-processed colon cancer data

Step 2: Load the necessary packages and APIs

Here is the code to load the necessary packages:

```
import org.apache.spark.api.java.JavaRDD;
import
org.apache.spark.api.java.function.Function;
import
org.apache.spark.mllib.classification.SVMModel;
import
org.apache.spark.mllib.classification.SVMWithSGD;
import
org.apache.spark.mllib.evaluation.BinaryClassificationMetrics;
import
org.apache.spark.mllib.evaluation.MulticlassMetrics;
import
org.apache.spark.mllib.optimization.L1Updater;
import
org.apache.spark.mllib.regression.LabeledPoint;
import org.apache.spark.mllib.util.MLUtils;
```

```
import org.apache.spark.sql.SparkSession;
```

Step 3: Configure the Spark session

The following code helps us to create the Spark session:

```
SparkSession spark = SparkSession
    .builder()
    .appName("JavaLDAExample")
    .master("local[*]")
    .config("spark.sql.warehouse.dir", "E:/Exp/")
    .getOrCreate();
```

Step 4: Create a Dataset out of the data

Here is the code to create a Dataset:

```
String path = "input/colon-cancer.data";
JavaRDD<LabeledPoint>data =
MLUtils.loadLibSVMFile(spark.sparkContext(),
path).toJavaRDD();
```

Step 5: Prepare the training and test sets

Here is the code to prepare the training and test sets:

```
    JavaRDD<LabeledPoint>training =
data.sample(false, 0.8, 11L);
training.cache();
    JavaRDD<LabeledPoint>test =
data.subtract(training);
```

Step 6: Build and train the SVM model

The following code illustrates how to build and train the SVM model:

```
intnumIterations = 500;
final SVMModel model =
SVMWithSGD.train(training.rdd(), numIterations);
```

Step 7: Compute the raw prediction score on the test set

Here is the code to compute the raw prediction:

```
JavaRDD<Tuple2<Object, Object>>scoreAndLabels =
test.map(
newFunction<LabeledPoint, Tuple2<Object,
Object>>() {
```

```
public Tuple2<Object, Object> call(LabeledPoint
p) {
          Double score =
model.predict(p.features());
returnnew Tuple2<Object, Object>(score,
p.label());
        }});
```

Step 8: Evaluate the model

Here is the code to evaluate the model:

```
BinaryClassificationMetrics metrics = new
BinaryClassificationMetrics(JavaRDD.toRDD(scoreAndLabels));
System.out.println("Area Under PR = " +
metrics.areaUnderPR());
System.out.println("Area Under ROC = " +
metrics.areaUnderROC());
Area Under PR = 0.6266666666666666
Area Under ROC = 0.875
```

However, the value of ROC is between 0.5 and 1.0. Where the value is more than 0.8 this indicates a good classifier and if the value of ROC is less than 0.8, this signals a bad classifier. The `SVMWithSGD.train()` method by default performs Level Two (L2) regularization with the regularization parameter set to 1.0.

If you want to configure this algorithm, you should customize the `SVMWithSGD` further by creating a new object directly. After that, you can further the setter methods to set the value of the object.

Interestingly, all the other Spark MLlib algorithms can be customized this way. However, after the customization has been completed, you need to build the source code to make changes up to API level. Interested readers can add themselves to the Apache Spark mailing list if they want to contribute to the open source.

Note the source code of Spark is available on GitHub at the URL https://github.com/apache/spark as an open source and it sends pull requests to enrich Spark. More technical discussion can be found at the Spark website at http://spark.apache.org/.

For example, the following code produces a level one (`L1`) regularized variant of SVMs with the regularization parameter set to 0.1, and runs the training algorithm for 500 iterations as follows:

```
SVMWithSGD svmAlg = new SVMWithSGD();
svmAlg.optimizer()
      .setNumIterations(500)
      .setRegParam(0.1)
```

```
        .setUpdater(new L1Updater());
final SVMModel model =
svmAlg.run(training.rdd());
```

Your model is now trained. Now if you perform *step 7* and *step 8*, the following metrics will be generated:

```
Area Under PR = 0.9380952380952381
Area Under ROC = 0.95
```

If you compare this result with the result produced in *step 8*, it's much better now, isn't it? However, depending on the data preparation, you might experience different results.

It indicates a better classification (please see also at https://www.researchgate.net/post/What_is_the_value_of_the_area_under_the_roc_curve_AU
In this way the SVM can be optimized or adaptive for the new data type.

However, the parameters (that is, number of iterations, regression params, and updater) should be set accordingly.

Incremental neural networks

The incremental version of the neural network in R or Mat lab provides adaptability using the adapt function. Does this update instead of overwriting iteratively? To verify this statement, readers can try using the R or Mat lab version of the incremental neural network-based classifier that may need to select a subset of your first data chunk as the second chunk in training. If it is overwriting, when you use the trained net with the subset to test your first data chunk, it will likely poorly predict the data that does not belong to the subset.

Multilayer perceptron classification with Spark

To date, there is no implementation of the incremental version of the neural network in Spark yet. According to the API documentation provided at https://spark.apache.org/docs/latest/ml-classification-regression.html#multilayer-perceptron-classifier, Spark's **Multilayer Perceptron Classifier** (**MLPC**) is a classifier based on the **Feedforward Artificial Neural Network** (**FANN**). The MLPC consists of multiple layers of nodes including hidden layers. Each layer is fully connected to the next layer and so on in a network. A node in the input layer represents the input data. All other nodes map inputs to outputs by a linear combination of the inputs with the node's weights *w* and bias *b* and by applying the activation function. The number of nodes *N* in the output layer corresponds to the number of classes.

MLPC also performs backpropagation for learning the model. Spark uses the logistic loss function for optimization and **Limited-memory Broyden-Fletcher-Goldfarb-Shanno** (**L-BFGS**) as an optimization routine. Note that the L-BFGS is an optimization algorithm in the family of **Quasi-Newton Method** (**QNM**) that approximates the Broyden-Fletcher-Goldfarb-Shanno algorithm using a limited main memory. To train the multilayer perceptron classifier, the following parameters need to be set:

- Layer
- Tolerance of iteration
- The block size of the learning
- Seed size
- Max iteration number

Note the layers consist of the input, hidden, and output layers. Moreover, a smaller value of convergence tolerance will lead to higher accuracy with the cost of more iterations. The default block size parameter is 128 and the maximum number of iteration is set to be 100 as a default value. We suggest you set these values accordingly and carefully.

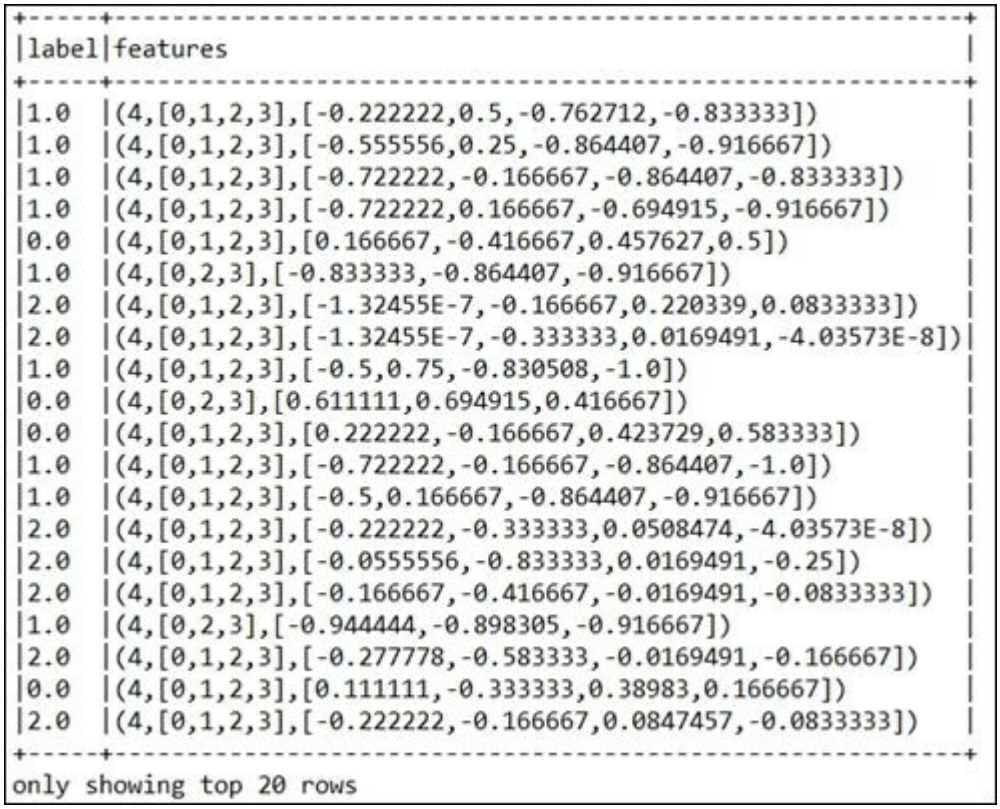
In this sub-section, we will show how Spark has implemented the neural network learning algorithms through the multilayer perception classifier on the Iris dataset.

Step 1: Dataset collection, processing, and exploration

The original Iris plant dataset was collected from the UCI machine learning repositories (http://www.ics.uci.edu/~mlearn/MLRepository.html) and then pre-processed, scaled to libsvm format by Chang et al., and placed as the libsvm a comprehensive library for support vector machine at (https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass.html) for the binary, multi-class, and multi-label classification task. The Iris dataset contains three classes and four features, where the sepal and petal lengths are scaled according to the libsvm format. More specifically, here is the attribute information:

- Class: Iris Setosa, Iris Versicolour, Iris Virginica (column 1)
- Sepal length in cm (column 2)
- Sepal width in cm (column 3)
- Petal length in cm (column 4)
- Petal width in cm (column 5)

A snapshot of the dataset is shown in *Figure 6*:



```
+-----+--------------------------------------------------------------+
|label|features                                                      |
+-----+--------------------------------------------------------------+
|1.0  |(4,[0,1,2,3],[-0.222222,0.5,-0.762712,-0.833333])            |
|1.0  |(4,[0,1,2,3],[-0.555556,0.25,-0.864407,-0.916667])           |
|1.0  |(4,[0,1,2,3],[-0.722222,-0.166667,-0.864407,-0.833333])      |
|1.0  |(4,[0,1,2,3],[-0.722222,0.166667,-0.694915,-0.916667])       |
|0.0  |(4,[0,1,2,3],[0.166667,-0.416667,0.457627,0.5])               |
|1.0  |(4,[0,2,3],[-0.833333,-0.864407,-0.916667])                   |
|2.0  |(4,[0,1,2,3],[-1.32455E-7,-0.166667,0.220339,0.0833333])      |
|2.0  |(4,[0,1,2,3],[-1.32455E-7,-0.333333,0.0169491,-4.03573E-8])   |
|1.0  |(4,[0,1,2,3],[-0.5,0.75,-0.830508,-1.0])                      |
|0.0  |(4,[0,2,3],[0.611111,0.694915,0.416667])                      |
|0.0  |(4,[0,1,2,3],[0.222222,-0.166667,0.423729,0.583333])          |
|1.0  |(4,[0,1,2,3],[-0.722222,-0.166667,-0.864407,-1.0])            |
|1.0  |(4,[0,1,2,3],[-0.5,0.166667,-0.864407,-0.916667])             |
|2.0  |(4,[0,1,2,3],[-0.222222,-0.333333,0.0508474,-4.03573E-8])     |
|2.0  |(4,[0,1,2,3],[-0.0555556,-0.833333,0.0169491,-0.25])          |
|2.0  |(4,[0,1,2,3],[-0.166667,-0.416667,-0.0169491,-0.0833333])     |
|1.0  |(4,[0,2,3],[-0.944444,-0.898305,-0.916667])                   |
|2.0  |(4,[0,1,2,3],[-0.277778,-0.583333,-0.0169491,-0.166667])      |
|0.0  |(4,[0,1,2,3],[0.111111,-0.333333,0.38983,0.166667])           |
|2.0  |(4,[0,1,2,3],[-0.222222,-0.166667,0.0847457,-0.0833333])      |
+-----+--------------------------------------------------------------+
only showing top 20 rows
```

Figure 6: Irish dataset snapshot

Step 2: Load the required packages and APIs

Here is the code to load the required packages and APIs:

```
import
org.apache.spark.ml.classification.MultilayerPerceptronClassificationMode
import
org.apache.spark.ml.classification.MultilayerPerceptronClassifier;
import
org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;
import
com.example.SparkSession.UtilityForSparkSession;
```

Step 3: Create a Spark session

The following code helps us to create the Spark session:

```
SparkSession spark =
UtilityForSparkSession.mySession();
```

Note, the `mySession()` method that creates and returns a Spark session object is as follows:

```
public static SparkSession mySession() {
SparkSession spark = SparkSession.builder()
.appName("MultilayerPerceptronClassificationModel")
.master("local[*]")
.config("spark.sql.warehouse.dir", "E:/Exp/")
.getOrCreate();
    return spark;
  }
```

Step 4: Parse and prepare the dataset

Load the input data as `libsvm` format:

```
String path = "input/iris.data";
Dataset<Row> dataFrame =
spark.read().format("libsvm").load(path);
```

Step 5: Prepare the training and test set

Prepare the training and test set: training = 70%, test = 30%, and seed = 12345L:

```
Dataset<Row>[] splits = dataFrame.randomSplit(new
double[] { 0.7, 0.3 }, 12345L);
Dataset<Row> train = splits[0];
Dataset<Row> test = splits[1];
```

Step 6: Specify the layers for the neural network

Specify the layers for the neural network. Here, input layer size 4 (features), two intermediate layers (that is, hidden layers) of size 4 and 3, and output size 3 (classes):

```
int[] layers = newint[] { 4, 4, 3, 3 };
```

Step 7: Create the multilayer perceptron estimator

Create the `MultilayerPerceptronClassifier` trainer and set its parameters. Here, set the value of `param [[layers]]` using the `setLayers()` method from *Step 6*. Set the convergence tolerance of iterations using the `setTol()` method, since, a smaller value will lead to higher accuracy with the cost of more iterations.

Note the default is `1E-4`. Set the value of Param `[[blockSize]]` using the `setBlockSize()` method, where the default is 128KB. Set the seed for weight initialization if the weights using the `setInitialWeights()` are not set. Finally, set the maximum number of iterations using the `setMaxIter()` method, where the default is 100:

```
MultilayerPerceptronClassifier trainer = new
MultilayerPerceptronClassifier()
        .setLayers(layers)
        .setTol(1E-4)
        .setBlockSize(128)
        .setSeed(12345L)
        .setMaxIter(100);
```

Step 8: Train the model

Train the `MultilayerPerceptronClassificationModel` using the preceding estimator from *step 7*:

```
MultilayerPerceptronClassificationModel model =
trainer.fit(train);
```

Step 9: Compute the accuracy on the test set

Here is the code to compute the accuracy on the test set:

```
Dataset<Row> result = model.transform(test);
Dataset<Row> predictionAndLabels =
result.select("prediction", "label");
```

Step 10: Evaluate the model

Evaluate the model, calculate the metrics`, and print the accuracy, weighted precision and weighted recall:

```
MulticlassClassificationEvaluator evaluator = new
MulticlassClassificationEvaluator().setMetricName("accuracy");
MulticlassClassificationEvaluator evaluator2 =
new
MulticlassClassificationEvaluator().setMetricName("weightedPrecision");
MulticlassClassificationEvaluator evaluator3 =
new
MulticlassClassificationEvaluator().setMetricName("weightedRecall");
System.out.println("Accuracy = " +
evaluator.evaluate(predictionAndLabels));
System.out.println("Precision = " +
evaluator2.evaluate(predictionAndLabels));
```

```
System.out.println("Recall = " +
evaluator3.evaluate(predictionAndLabels));
```

The output should appear as follows:

```
Accuracy = 0.9545454545454546
Precision = 0.9595959595959596
Recall = 0.9545454545454546
```

Step 11: Stop the Spark session

The following code is used to stop the Spark session:

```
spark.stop();
```

From the preceding prediction metrics, it is clear that the classification task is quite impressive. Now it's your turn to make your model adaptable. Now try training and testing with the new dataset and make your ML model adaptable.

Incremental Bayesian networks

As we discussed earlier, Naive Bayes is a simple multiclass classification algorithm with the assumption of independence between each pair of features. The Naive Bayes based model can be trained very efficiently. The model can compute the conditional probability distribution of each feature, given the label, since a pass to the training data. After that, it applies the Bayes theorem to compute the conditional probability distribution of the labels for making the prediction.

However, there is still no implementation of the incremental version of the Bayesian network into Spark yet. According to the API documentation provided at http://spark.apache.org/docs/latest/mllib-naive-bayes.html, each observation is a document and each feature represents a term. The value of an observation is the frequency of the term or a zero or one. This value indicates if the term has been found in the document for the multinomial Naive Bayes and Bernoulli Naive Bayes respectively for the document classification.

Note that as with linear SVM-based learning, here the feature values must be non-negative too. The type of the model is selected with an optional parameter, multinomial or Bernoulli. The default model type is multinomial. Furthermore, additive smoothing (that is, lambda) can be used by setting the parameter Î». Note the default of lambda is 1.0.

More technical details on the big data approach of Bayesian network based learning can be found in the paper: *A Scalable Data Science Workflow*

Approach for Big Data Bayesian Network Learning b y Jianwu W., et al., (http://users.sdsc.edu/~jianwu/JianwuWang_files/A_Scalable_Data_Science_Workflow_Appro

Interested readers also should refer to the following publications for more insight into the incremental Bayesian networks:

- http://www.jmlr.org/papers/volume11/henderson10a/henderson10a.pdf
- http://www.machinelearning.org/proceedings/icml2007/papers/351.pdf
- https://tel.archives-ouvertes.fr/tel-01284332/document

Classification using Naive Bayes with Spark

The current implementation in Spark MLlib supports both the multinomial Naive Bayes and Bernoulli Naive Bayes. However, the incremental version has not been implemented yet. Therefore, in this section, we will show you how to perform the classification using the Spark MLlib version of NaÃ¯ve Bayes on the Vehicle Scale dataset to provide you with some concepts of the NaÃ¯ve Bayes based learning.

Note, due to the low accuracy and precision using Spark ML, we did not provide the Pipeline version but implemented the same using only Spark MLlib. Moreover, if you have suitable and better data, you can try to implement the Spark ML version with ease.

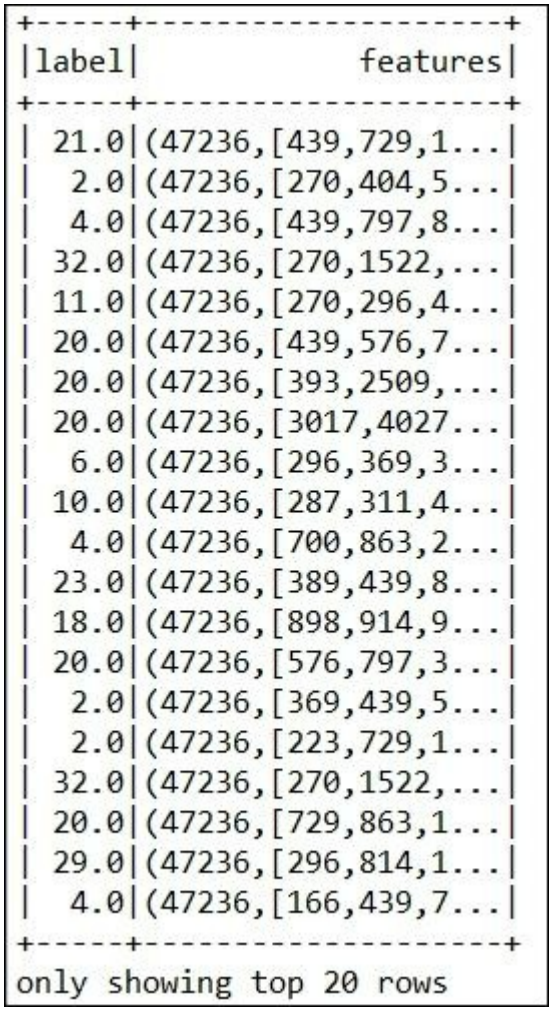
Step 1: Data collection, pre-processing, and exploration

The dataset was downloaded from https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass.html#aloi and provided by David D. Lewis, Yiming Yang, Tony G. Rose, and Fan Li. RCV1: A new benchmark collection for text categorization research. *Journal of Machine Learning Research*, 5:361-397, 2004.

Pre-processing: For the pre-processing, two steps were considered as follows:

- The label hierarchy is reorganized by mapping the data set to the second level of RCV1 (that is, revision) topic hierarchy. The documents, having the third or fourth level, are mapped to their parent category of the second level only. Consequently, documents having the first level are not considered for creating the mapping.
- Multi-labeled instances were removed since the current implementation of multi-level classifier in Spark is not robust enough.

After performing these two steps, there are finally 53 classes and 47,236 features collected. Here is a snapshot of the dataset shown in *Figure 7:*



```
+-----+--------------------+
|label|            features|
+-----+--------------------+
| 21.0|(47236,[439,729,1...|
|  2.0|(47236,[270,404,5...|
|  4.0|(47236,[439,797,8...|
| 32.0|(47236,[270,1522,...|
| 11.0|(47236,[270,296,4...|
| 20.0|(47236,[439,576,7...|
| 20.0|(47236,[393,2509,...|
| 20.0|(47236,[3017,4027...|
|  6.0|(47236,[296,369,3...|
| 10.0|(47236,[287,311,4...|
|  4.0|(47236,[700,863,2...|
| 23.0|(47236,[389,439,8...|
| 18.0|(47236,[898,914,9...|
| 20.0|(47236,[576,797,3...|
|  2.0|(47236,[369,439,5...|
|  2.0|(47236,[223,729,1...|
| 32.0|(47236,[270,1522,...|
| 20.0|(47236,[729,863,1...|
| 29.0|(47236,[296,814,1...|
|  4.0|(47236,[166,439,7...|
+-----+--------------------+
only showing top 20 rows
```

Figure 7: RCV1 topic hierarchy dataset

Step 2: Load the required library and packages

Here is the code to load the library and packages:

```
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import
org.apache.spark.api.java.function.Function;
import
org.apache.spark.api.java.function.PairFunction;
import
org.apache.spark.mllib.classification.NaiveBayes;
import
org.apache.spark.mllib.classification.NaiveBayesModel;
import
org.apache.spark.mllib.regression.LabeledPoint;
import org.apache.spark.mllib.util.MLUtils;
import org.apache.spark.sql.SparkSession;
importscala.Tuple2;
```

Step 3: Initiate a Spark session

The following code helps us to create the Spark session:

```
static SparkSession spark = SparkSession
      .builder()

 .appName("JavaLDAExample").master("local[*]")
      .config("spark.sql.warehouse.dir",
"E:/Exp/")
      .getOrCreate();
```

Step 4: Prepare LabeledPoint RDDs

Parse the dataset in the libsvm format and prepare `LabeledPoint` RDDs:

```
static String path =
"input/rcv1_train.multiclass.data";
JavaRDD<LabeledPoint> inputData =
MLUtils.loadLibSVMFile(spark.sparkContext(),
path).toJavaRDD();
```

For document classification, the input feature vectors are usually sparse, and sparse vectors should be supplied as input to take advantage of sparsity. Since the training data is only used once, it is not necessary to cache it.

Step 5: Prepare the training and test set

Here is the code to prepare the training and test set:

```
JavaRDD<LabeledPoint>[] split =
inputData.randomSplit(new double[]{0.8, 0.2},
12345L);
JavaRDD<LabeledPoint> training = split[0];
JavaRDD<LabeledPoint> test = split[1];
```

Step 6: Train the Naive Bayes model

Train a Naive Bayes model by specifying the model type as multinomial and lambda = 1.0, which is the default and suitable for the multiclass classification of any features. However, note that Bernoulli naive Bayes requires 0 or 1 feature values:

```
final NaiveBayesModel model =
NaiveBayes.train(training.rdd(), 1.0,
"multinomial");
```

Step 7: Calculate the prediction on the test dataset

Here is the code to calculate the prediction:

```
JavaPairRDD<Double,Double> predictionAndLabel =
test.mapToPair(new PairFunction<LabeledPoint,
Double, Double>() {
@Override
public Tuple2<Double, Double> call(LabeledPoint
p) {
return new Tuple2<>(model.predict(p.features()),
p.label());
          }
        });
```

Step 8: Calculate the prediction accuracy

Here is the code to calculate the prediction accuracy:

```
double accuracy = predictionAndLabel.filter(new
Function<Tuple2<Double, Double>, Boolean>() {
@Override
public Boolean call(Tuple2<Double, Double>pl) {
returnpl._1().equals(pl._2());
        }
      }).count() / (double) test.count();
```

Step 9: Print the accuracy

Here is the code to print the accuracy:

```
System.out.println("Accuracy of the
classification: "+accuracy);
```

This provides the following output:

```
Accuracy of the classification:
0.5941753719531497
```

This is pretty low, right? This is as we discussed when we tuned the ML models in Chapter 7, *Tuning Machine Learning Models*. There are further opportunities to improve the prediction accuracy by selecting appropriate algorithms (that is, classifier or regressor) via cross-validation and train split.

Adapting through reusing ML models

In this section, we will describe how to make a machine learning model adaptable for new datasets. An example will be shown for the prediction of heart disease. At first we will describe the problem statement, and then we will explore the heart diseases dataset. Following the dataset exploration, we will train and save the model to local storage. After that the model will be evaluated to see how it performs. Finally, we will reuse/reload the same model trained to work for the new data type.

More specifically, we will show how to predict the possibility of future heart disease by using the Spark machine learning APIs including Spark MLlib, Spark ML, and Spark SQL.

Problem statements and objectives

Machine learning and big data together are a radical combination that has created some great impacts in the field of research to academia and the industry as well as in the biomedical sector. In the area of biomedical data analytics, this carries a better impact on a real dataset for diagnosis and prognosis for better healthcare. Moreover, life science research is also entering into big data since datasets are being generated and produced in an unprecedented way. This imposes great challenges to machine learning and bioinformatics tools and algorithms to find the VALUE from big data criteria such as volume, velocity, variety, veracity, visibility, and value.

Data exploration

In recent times, biomedical research has advanced enormously and more and more life sciences datasets are being generated making many of them open source. However, for simplicity and ease, we have decided to use the Cleveland database. To date, most of the researchers who have applied the machine learning technique to biomedical data analytics have used this dataset. According to the dataset description at https://archive.ics.uci.edu/ml/machine-learning-databases/heart-disease/heart-disease.names, this heart disease dataset is one of the most used and well-studied datasets by researchers from biomedical data analytics and machine learning fields, respectively.

The dataset is freely available at the UCI machine learning dataset repository at https://archive.ics.uci.edu/ml/machine-learning-databases/heart-disease/. This data contains a total of 76 attributes, however, most of the published research papers refer to using a subset of only 14 features in the field. The "goal" field is used to refer to if the heart

diseases are present or absent. It has five possible values ranging from 0 to 4. The value 0 signifies no presence of heart diseases. The values 1 and 2 signify that the disease is present but in the primary stage. The values 3 and 4, on the other hand, indicate the strong possibility of heart disease. Biomedical laboratory experiments with the Cleveland dataset have simply attempted to distinguish presence (values 1, 2, 3, 4) from absence (value 0). In short, the higher the value the more the disease is possible and the more evidence of the presence there is. Another thing is that privacy is an important concern in the area of biomedical data analytics as well as all kinds of diagnosis and prognosis. Therefore, the names and social security numbers of the patients were recently removed from the dataset to avoid the privacy issue. Consequently, those values have been replaced with dummy values instead.

It is to be noted that three files have been processed, containing the Cleveland, Hungarian, and Switzerland datasets altogether. All four unprocessed files also exist in this directory. To demonstrate the example, we will use the Cleveland dataset for training and evaluating the models. However, the Hungarian dataset will be used to re-use the saved model. As we have said already, although the number of attributes is 76 (including the predicted attribute), like other ML/Biomedical researchers, we will also use only 14 attributes with the following attribute information:

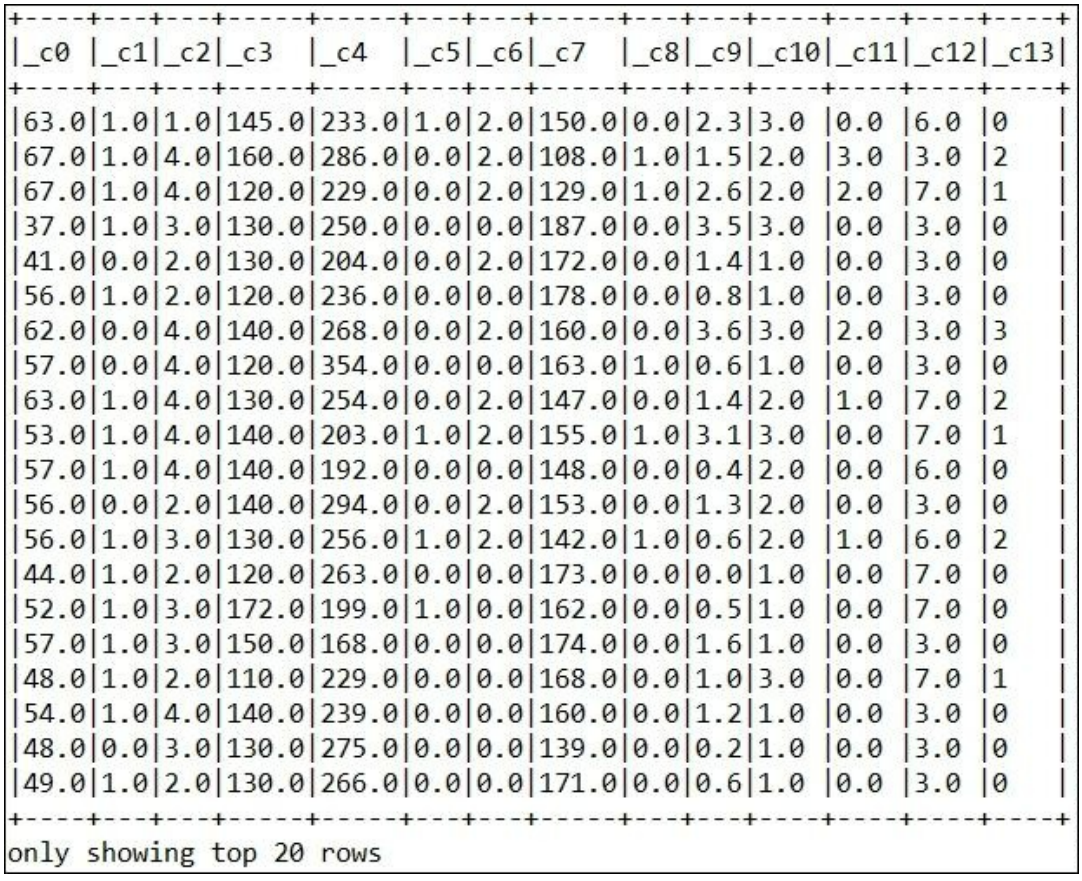
No.	**Attribute name**	**Explanation**
1	age	Age in years
2	sex	Either male or female: sex (1 = male; 0 = female)
3	cp	Chest pain type: — Value 1: typical angina — Value 2: atypical angina — Value 3: non-angina pain — Value 4: asymptomatic

4	trestbps	Resting blood pressure (in mm Hg on admission to the hospital)
5	chol	Serum cholesterol in mg/dl
6	fbs	Fasting blood sugar. If > 120 mg/dl)(1 = true; 0 = false)
7	restecg	Resting electrocardiographic results: — Value 0: normal — Value 1: having ST-T wave abnormality — Value 2: showing probable or definite left ventricular hypertrophy by Estes' criteria.
8	thalach	Maximum heart rate achieved
9	exang	Exercise induced angina (1 = yes; 0 = no)
10	oldpeak	ST depression induced by exercise relative to rest
11	slope	The slope of the peak exercise ST segment — Value 1: upsloping — Value 2: flat — Value 3: down-sloping
12	ca	Number of major vessels (0-3) colored by fluoroscopy
13	thal	Heart rate: —Value 3 = normal; —Value 6 = fixed defect

		—Value 7 = reversible defect
14	num	Diagnosis of heart disease (angiographic disease status) — Value 0: < 50% diameter narrowing — Value 1: > 50% diameter narrowing

Table 1: Dataset characteristics

A sample snapshot of the dataset is given as follows:



```
+----+---+---+-----+-----+---+---+-----+---+---+----+----+----+----+
|_c0 |_c1|_c2|_c3  |_c4  |_c5|_c6|_c7  |_c8|_c9|_c10|_c11|_c12|_c13|
+----+---+---+-----+-----+---+---+-----+---+---+----+----+----+----+
|63.0|1.0|1.0|145.0|233.0|1.0|2.0|150.0|0.0|2.3|3.0 |0.0 |6.0 |0   |
|67.0|1.0|4.0|160.0|286.0|0.0|2.0|108.0|1.0|1.5|2.0 |3.0 |3.0 |2   |
|67.0|1.0|4.0|120.0|229.0|0.0|2.0|129.0|1.0|2.6|2.0 |2.0 |7.0 |1   |
|37.0|1.0|3.0|130.0|250.0|0.0|0.0|187.0|0.0|3.5|3.0 |0.0 |3.0 |0   |
|41.0|0.0|2.0|130.0|204.0|0.0|2.0|172.0|0.0|1.4|1.0 |0.0 |3.0 |0   |
|56.0|1.0|2.0|120.0|236.0|0.0|0.0|178.0|0.0|0.8|1.0 |0.0 |3.0 |0   |
|62.0|0.0|4.0|140.0|268.0|0.0|2.0|160.0|0.0|3.6|3.0 |2.0 |3.0 |3   |
|57.0|0.0|4.0|120.0|354.0|0.0|0.0|163.0|1.0|0.6|1.0 |0.0 |3.0 |0   |
|63.0|1.0|4.0|130.0|254.0|0.0|2.0|147.0|0.0|1.4|2.0 |1.0 |7.0 |2   |
|53.0|1.0|4.0|140.0|203.0|1.0|2.0|155.0|1.0|3.1|3.0 |0.0 |7.0 |1   |
|57.0|1.0|4.0|140.0|192.0|0.0|0.0|148.0|0.0|0.4|2.0 |0.0 |6.0 |0   |
|56.0|0.0|2.0|140.0|294.0|0.0|2.0|153.0|0.0|1.3|2.0 |0.0 |3.0 |0   |
|56.0|1.0|3.0|130.0|256.0|1.0|2.0|142.0|1.0|0.6|2.0 |1.0 |6.0 |2   |
|44.0|1.0|2.0|120.0|263.0|0.0|0.0|173.0|0.0|0.0|1.0 |0.0 |7.0 |0   |
|52.0|1.0|3.0|172.0|199.0|1.0|0.0|162.0|0.0|0.5|1.0 |0.0 |7.0 |0   |
|57.0|1.0|3.0|150.0|168.0|0.0|0.0|174.0|0.0|1.6|1.0 |0.0 |3.0 |0   |
|48.0|1.0|2.0|110.0|229.0|0.0|0.0|168.0|0.0|1.0|3.0 |0.0 |7.0 |1   |
|54.0|1.0|4.0|140.0|239.0|0.0|0.0|160.0|0.0|1.2|1.0 |0.0 |3.0 |0   |
|48.0|0.0|3.0|130.0|275.0|0.0|0.0|139.0|0.0|0.2|1.0 |0.0 |3.0 |0   |
|49.0|1.0|2.0|130.0|266.0|0.0|0.0|171.0|0.0|0.6|1.0 |0.0 |3.0 |0   |
+----+---+---+-----+-----+---+---+-----+---+---+----+----+----+----+
only showing top 20 rows
```

Figure 8: A sample snapshot of the heart diseases dataset

Developing a heart diseases predictive model

Step 1: Loading the required packages and APIs

The following packages and APIs need to be imported for our purpose. We believe the packages are self-explanatory if you have the minimum working experience with Spark 2.0.0:

```
import java.util.HashMap;
import java.util.List;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import
org.apache.spark.api.java.function.Function;
import
org.apache.spark.api.java.function.PairFunction;
import
org.apache.spark.ml.classification.LogisticRegression;
import
org.apache.spark.mllib.classification.LogisticRegressionModel;
import
org.apache.spark.mllib.classification.NaiveBayes;
import
org.apache.spark.mllib.classification.NaiveBayesModel;
import org.apache.spark.mllib.linalg.DenseVector;
import org.apache.spark.mllib.linalg.Vector;
import
org.apache.spark.mllib.regression.LabeledPoint;
import
org.apache.spark.mllib.regression.LinearRegressionModel;
import
org.apache.spark.mllib.regression.LinearRegressionWithSGD;
import org.apache.spark.mllib.tree.DecisionTree;
import org.apache.spark.mllib.tree.RandomForest;
import
org.apache.spark.mllib.tree.model.DecisionTreeModel;
import
org.apache.spark.mllib.tree.model.RandomForestModel;
import org.apache.spark.rdd.RDD;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;
import
com.example.SparkSession.UtilityForSparkSession;
import javassist.bytecode.Descriptor.Iterator;
import scala.Tuple2;
```

Step 2: Create an active Spark session

The following code helps us to create the Spark session:

```
SparkSession spark =
UtilityForSparkSession.mySession();
```

Here is the `UtilityForSparkSession` class that creates and returns an active Spark session:

```
import org.apache.spark.sql.SparkSession;
public class UtilityForSparkSession {
  public static SparkSession mySession() {
    SparkSession spark = SparkSession
                         .builder()

 .appName("UtilityForSparkSession")
                         .master("local[*]")

 .config("spark.sql.warehouse.dir", "E:/Exp/")
                         .getOrCreate();
    return spark;
  }
}
```

Note that here in the Windows 7 platform, we have set the Spark SQL warehouse as `E:/Exp/`, but set your path accordingly, based on your operating system.

Step 3: Data parsing and RDD of Labelpoint creation

Take the input as a simple text file, parse them as a text file, and create an RDD of the label point that will be used for the classification and regression analysis. Also specify the input source and number of partition. Adjust the number of partition based on your dataset size. Here the number of partition has been set to 2:

```
String input =
"heart_diseases/processed_cleveland.data";
Dataset<Row> my_data =
spark.read().format("com.databricks.spark.csv").load(input);
my_data.show(false);
RDD<String> linesRDD =
spark.sparkContext().textFile(input, 2);
```

Since `JavaRDD` cannot be created directly from the text files, we have created a simple RDDs so that we can convert them to `JavaRDD` when necessary. Now let's create the `JavaRDD` with a Label Point. However, we first need to convert the RDD to `JavaRDD` to serve our purpose as follows:

```
JavaRDD<LabeledPoint> data =
linesRDD.toJavaRDD().map(new Function<String,
LabeledPoint>() {
      @Override
  public LabeledPoint call(String row) throws
Exception {
```

```
      String line = row.replaceAll("\\?",
"999999.0");
      String[] tokens = line.split(",");
      Integer last =
Integer.parseInt(tokens[13]);
      double[] features = new double[13];
      for (int i = 0; i < 13; i++) {
      features[i] =
Double.parseDouble(tokens[i]);
      }
      Vector v = new DenseVector(features);
      Double value = 0.0;
      if (last.intValue() > 0)
        value = 1.0;
      LabeledPoint lp = new LabeledPoint(value,
v);
    return lp;
      }
    });
```

Using the `replaceAll()` method, we have handled the invalid values such as the missing values that are specified in the original file using the *?* character. To get rid of the missing or invalid values we have replaced them with a very large value that has no side effect to the original classification or predictive results. The reason for this is that missing or sparse data can lead you to highly misleading results.

Step 4: Splitting the RDD of the label point into training and test sets

In the previous step, we created RDD label point data that can be used for the regression or classification task. Now we need to split the data into training and test sets as follows:

```
double[] weights = {0.7, 0.3};
long split_seed = 12345L;
JavaRDD<LabeledPoint>[] split =
data.randomSplit(weights, split_seed);
JavaRDD<LabeledPoint> training = split[0];
JavaRDD<LabeledPoint> test = split[1];
```

If you look at the preceding code segments, you will find that we have split the RDD label point as 70% for the training and 30% for the test set. The `randomSplit()` method performs this split. Note that we have set this RDD's storage level to persist its values across operations after the first time it is computed. This can only be used to assign a new storage level if the RDD does not have a storage level set yet. The split seed value is a long integer that signifies that the split would be random, but the result would not be a change in each run or iteration during the model building or training.

Step 5: Train the model

First we will train the linear regression model, which is the simplest regression classifier:

```
final double stepSize = 0.0000000009;
final int numberOfIterations = 40;
LinearRegressionModel model =
LinearRegressionWithSGD.train(JavaRDD.toRDD(training),
numberOfIterations, stepSize);
```

As you can see, the preceding code trains a linear regression model with no regularization using the Stochastic Gradient Descent. This solves the least squares regression formulation f *(weights) = 1/n ||A weights-y||^2^*, which is the mean squared error. Here the data matrix has *n* rows, and the input RDD holds the set of rows of A, each with its corresponding right-hand side label y. Also, to train the model, it takes the training set, number of iterations, and the step size. We provide some random values for the last two parameters here.

Step 6: Model saving for future use

Now let's save the model that we just created for future use. It's pretty simple - just use the following code by specifying the storage location as follows:

```
String model_storage_loc = "models/heartModel";
model.save(spark.sparkContext(),
model_storage_loc);
```

Once the model is saved in your desired location, you will see the following output in your Eclipse console:

```
21-Aug-2016 23:57:13 INFO: org.apache.parquet.hadoop.codec.CodecConfig: Compression: SNAPPY
21-Aug-2016 23:57:13 INFO: org.apache.parquet.hadoop.ParquetOutputFormat: Parquet block size to 134217728
21-Aug-2016 23:57:13 INFO: org.apache.parquet.hadoop.ParquetOutputFormat: Parquet page size to 1048576
21-Aug-2016 23:57:13 INFO: org.apache.parquet.hadoop.ParquetOutputFormat: Parquet dictionary page size to 1048576
21-Aug-2016 23:57:13 INFO: org.apache.parquet.hadoop.ParquetOutputFormat: Dictionary is on
21-Aug-2016 23:57:13 INFO: org.apache.parquet.hadoop.ParquetOutputFormat: Validation is off
21-Aug-2016 23:57:13 INFO: org.apache.parquet.hadoop.ParquetOutputFormat: Writer version is: PARQUET_1_0
21-Aug-2016 23:57:13 INFO: org.apache.parquet.hadoop.InternalParquetRecordWriter:Flushing mem columnStore to file.
allocated memory: 240
21-Aug-2016 23:57:13 INFO: org.apache.parquet.hadoop.ColumnChunkPageWriteStore: written 45B for [weights, type]
INT32: 1 values, 10B raw, 12B comp, 1 pages, encodings: [PLAIN, RLE, BIT_PACKED]
21-Aug-2016 23:57:13 INFO: org.apache.parquet.hadoop.ColumnChunkPageWriteStore: written 30B for [weights, size]
INT32: 1 values, 7B raw, 9B comp, 1 pages, encodings: [PLAIN, RLE, BIT_PACKED]
21-Aug-2016 23:57:13 INFO: org.apache.parquet.hadoop.ColumnChunkPageWriteStore: written 36B for [weights, indices,
list, element] INT32: 1 values, 13B raw, 15B comp, 1 pages, encodings: [PLAIN, RLE]
21-Aug-2016 23:57:13 INFO: org.apache.parquet.hadoop.ColumnChunkPageWriteStore: written 163B for [weights, values,
list, element] DOUBLE: 13 values, 117B raw, 120B comp, 1 pages, encodings: [PLAIN, RLE]
21-Aug-2016 23:57:13 INFO: org.apache.parquet.hadoop.ColumnChunkPageWriteStore: written 51B for [intercept] DOUBLE: 1
values, 8B raw, 10B comp, 1 pages, encodings: [PLAIN, BIT_PACKED]
```

Figure 9: The log after the model is saved to the storage

Step 7: Evaluate the model with a test set

Now let's calculate the prediction score on the test dataset:

```
JavaPairRDD<Double,Double> predictionAndLabel =
  test.mapToPair(new PairFunction<LabeledPoint,
Double, Double>() {
            @Override
    public Tuple2<Double, Double>
call(LabeledPoint p) {
       return new
Tuple2<>(model.predict(p.features()), p.label());
            }
          });
```

Predict the accuracy of the prediction:

```
double accuracy = predictionAndLabel.filter(new
Function<Tuple2<Double, Double>, Boolean>() {
          @Override
          public Boolean call(Tuple2<Double,
Double> pl) {
            return pl._1().equals(pl._2());
          }
        }).count() / (double) test.count();
System.out.println("Accuracy of the
classification: "+accuracy);
```

The output appears as follows:

```
Accuracy of the classification: 0.0
```

Step 8: Predictive analytics using a different classifier

Unfortunately, there is no prediction accuracy at all, right? There might be several reasons for that, including the following:

- The dataset characteristic
- Model selection
- Parameters selection - also called hyperparameter tuning

For simplicity, we assume the dataset is okay since, as we have already said, it is a widely used dataset used for machine learning research used by many researchers around the globe. Now, what next? Let's consider another classifier algorithm, for example, a Random forest or decision tree classifier. What about the Random forest? Let's go for the random forest classifier at second place. Just use the following code to train the model using the training set:

```
Integer numClasses = 26; //Number of classes
```

Now use the `HashMap` to restrict the delicacy in the tree construction:

```
HashMap<Integer, Integer> categoricalFeaturesInfo
= new HashMap<Integer, Integer>();
```

Now declare the other parameters needed to train the Random Forest classifier:

```
Integer numTrees = 5; // Use more in practice
String featureSubsetStrategy = "auto"; // Let
algorithm choose the best
String impurity = "gini"; // info. gain &
variance also available
Integer maxDepth = 20; // set the value of
maximum depth accordingly
Integer maxBins = 40; // set the value of bin
accordingly
Integer seed = 12345; //Setting a long seed value
is recommended
final RandomForestModel model =
RandomForest.trainClassifier(training,
numClasses,categoricalFeaturesInfo, numTrees,
featureSubsetStrategy, impurity, maxDepth,
maxBins, seed);
```

We believe the parameters used by the `trainClassifier()` method are self-explanatory and so we'll leave it to the readers to get to know the significance of each parameter. Fantastic! We have trained the model using the Random forest classifier and managed the cloud to save the model for future use. Now if you reuse the same code that we described in the *Evaluate the model with test set* step, you should have the following output:

```
Accuracy of the classification:
0.7843137254901961
```

Now the predictive accuracy should be much better. If you are still not satisfied, you can try with another classifier model such as the Naive Bayes classifier and carry out the hyperparameter tuning discussed in Chapter 7, *Tuning Machine Learning Models*.

Step 9: Making the model adaptable for a new dataset

We already mentioned that we have saved the model for future use, now we should take the opportunity to use the same model for new datasets. The reason, if you recall the steps, is that we have trained the model using the training set and evaluated it using the test set. Now, if you have more data or new data available to be used, what will you do? Will you go for re-

training the model? Of course not, since you will have to iterate several steps and you will have to sacrifice valuable time and cost too.

Therefore, it would be wise to use the already trained model and predict the performance on a new dataset. Well, now let's reuse the stored model. Note that you will have to reuse the same model that is to be trained for the same model. For example, if you have done the model training using the Random forest classifier and saved the model while reusing it, you will have to use the same classifier model to load the saved model. Therefore, we will use the Random forest to load the model while using the new dataset. Use the following code to do that. Now create an RDD label point from the new dataset (that is, the Hungarian database with the same 14 attributes):

```
String new_data =
"heart_diseases/processed_hungarian.data";
RDD<String> linesRDD =
spark.sparkContext().textFile(new_data, 2);
JavaRDD<LabeledPoint> data =
linesRDD.toJavaRDD().map(new Function<String,
LabeledPoint>() {
      @Override
  public LabeledPoint call(String row) throws
Exception {
  String line = row.replaceAll("\\?",
"999999.0");
  String[] tokens = line.split(",");
  Integer last = Integer.parseInt(tokens[13]);
    double[] features = new double[13];
              for (int i = 0; i < 13; i++) {
          features[i] =
Double.parseDouble(tokens[i]);
                }
      Vector v = new DenseVector(features);
      Double value = 0.0;
      if (last.intValue() > 0)
        value = 1.0;
      LabeledPoint p = new LabeledPoint(value,
v);
      return p;
      } });
```

Now let's load the saved model using the Random forest model algorithm as follows:

```
RandomForestModel model2 =
RandomForestModel.load(spark.sparkContext(),
model_storage_loc);
```

Now let's calculate the prediction on the test set:

```
JavaPairRDD<Double, Double> predictionAndLabel =
```

```
  data.mapToPair(new PairFunction<LabeledPoint,
Double, Double>() {
          @Override
          public Tuple2<Double, Double>
call(LabeledPoint p) {
      return new
Tuple2<>(model2.predict(p.features()),
p.label());
            }
          });
```

Now calculate the accuracy of the prediction as follows:

```
double accuracy = predictionAndLabel.filter(new
Function<Tuple2<Double, Double>, Boolean>() {
          @Override
          public Boolean call(Tuple2<Double,
Double> pl) {
            return pl._1().equals(pl._2());
          }
        }).count() / (double) data.count();
System.out.println("Accuracy of the
classification: "+accuracy);
```

We should have the following output:

```
Accuracy of the classification:
0.9108910891089109
```

Now train the NaÃ¯ve Bayesian classifier and see the predictive performance. Just download the source code for the Naive Bayesian classifier and run the code as a Maven-friendly project using the `pom.xml` file that includes all the dependencies of the required JARs and APIs.

The following table shows a comparison of the predictive accuracies among three classifiers (that is, Linear Regression, Random Forest, and the Naive Bayesian classifier). Note that, depending upon the training, the model you get might have different output since we randomly split the dataset into training and testing:

Classifier	**Model building time**	**Model saving time**	**Accuracy**
Linear regression	1199 ms	2563 ms	0.0%
NaÃ¯ve Bayes	873 ms	2514 ms	45%

Random forest	2120 ms	2538 ms	91%

Table 2: Comparison between three classifiers

> **Note**
>
> We avail the preceding output in a machine with a Windows 7(64-bit), Core i7 (2.90GHz) processor, and 32GB of main memory. Therefore, depending upon your OS type and hardware configuration, you might receive different results.

This way the ML model can be made adaptable for the new data type. However, make sure that you use the same classifier or regressor to train and reuse the model to make the ML application adaptable.

Machine learning in dynamic environments

Making a prediction in dynamic environments does not always succeed in producing desired outcomes, particularly in complex and unstructured data.

There are several reasons for that. For example, how do you infer a realistic outcome from a bit of data or deal with unstructured and high dimensional data that has been found too tedious? Moreover, model revision with efficient strategies to control the realistic environments is also costly.

Furthermore, sometimes the dimensionality of the input dataset is high. Consequently, data might be too dense or very sparse. In that case, how you deal with very large settings and how to apply the static models in emerging application areas such as robotics, image processing, deep learning, computer vision, or web mining is challenging. On the other hand, ensemble methods are becoming more popular for selecting and combining models from existing models to make the ML model more adaptable. A hierarchical and dynamic environment-based learning is shown in *Figure 10:*

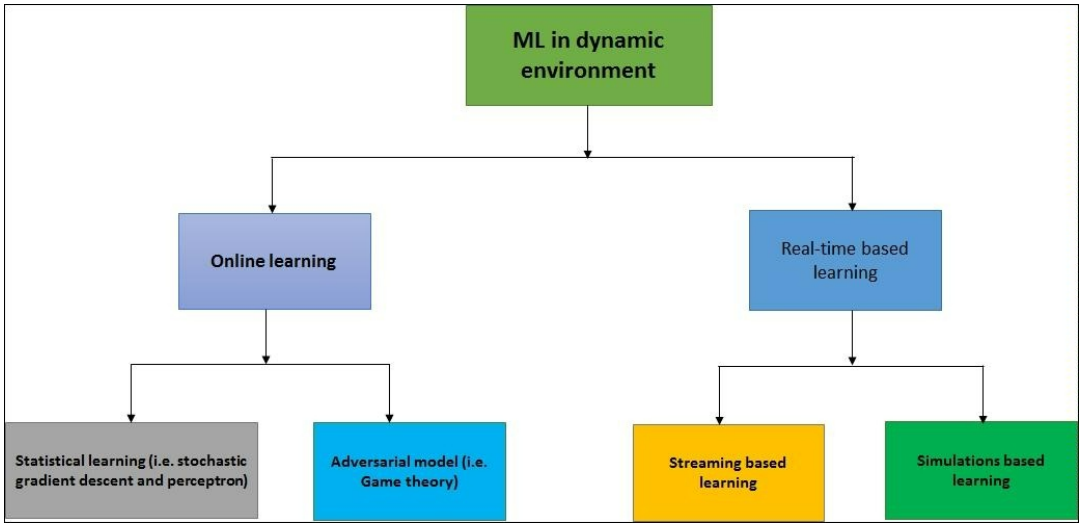


Figure 10: The hierarchy of machine learning in a dynamic environment

In this case, ML techniques such as neural networks and statistically-based learning are also becoming popular for their success with numerous applications in industry and research such as biological systems. In particular, classical learning algorithms such as neural networks, decision trees, or vector quantizes are often restricted to purely feedforward settings, and simple vectorial data, instead of dynamic environments. The feature of vectorization often provides a better prediction because of the rich

structure. In summary, there are three challenges in developing ML applications in a dynamic environment:

- How does the data structure emerge to shape in an autonomous environment?
- How do we deal with input data that is statistically sparse and high dimensional? More specifically, what about making predictive analysis using online algorithms for large-scale datasets, applying the dimensionality reduction and so on?

With only limited reinforcement signals, ill-posed domains, or partially underspecified settings, how do we develop controlled and effective strategies in dynamic environments? Considering these issues and promising advancement in the research, in this section, we will provide some insights into online learning techniques through a statistical and adversarial model. Since learning in a dynamic environment such as streaming will be discussed in Chapter 9, *Advanced Machine Learning with Streaming and Graph Data*, we will not discuss streaming-based learning in this chapter.

Online learning

Batch learning techniques generate the best predictor by learning on the entire training dataset at once and are often called static learning. Static learning algorithms take batches of training data to train a model, then a prediction is made using the test sample and the found relationship, whereas online learning algorithms take an initial guess model and then pick up a one-one observation from the training population and recalibrate the weights on each input parameter. Data usually becomes available in a sequential order as batches. The sequential data is used to update the best predictor of the outcome at each step as outlined in *Figure 11*. There are three use cases of online-based learning:

- Firstly, where it is computationally infeasible to train an ML model over the entire dataset, an online learning is commonly used
- Secondly, it is also used in a situation where it is necessary for the algorithm to dynamically adapt to new patterns in the data
- Thirdly, it is used when the data itself is generated as a function of time, for example, the stock price prediction

Online learning, therefore, requires out-of-core algorithms, that is, algorithms that can perform considering the constraints of networks. There are two general modeling strategies that exist for online learning models:

- **Statistical learning models**: For example, stochastic gradient descent and perceptron

- **Adversarial models**: For example, spam filtering falls into this category, as the adversary will dynamically generate new spam based on the current behavior of the spam detector

Although online and incremental learning techniques are similar, they also differ slightly. In online, it's generally a single pass (epoch=1) or a number of epochs that could be configured, whereas, incremental would mean that you already have a model. No matter how it is built, the model can be mutable by new examples. Also, a combination of online and incremental is often what is required.

Data is being generated in an unprecedented way everywhere, every day. This huge data imposes an enormous challenge to building ML tools that can handle data with high volume, velocity, and veracity. In short, data generated online is also big data. Therefore, we need to know the technique by which to learn about the online learning algorithms that are meant to handle data with such high volume and velocity with limited performance machines.

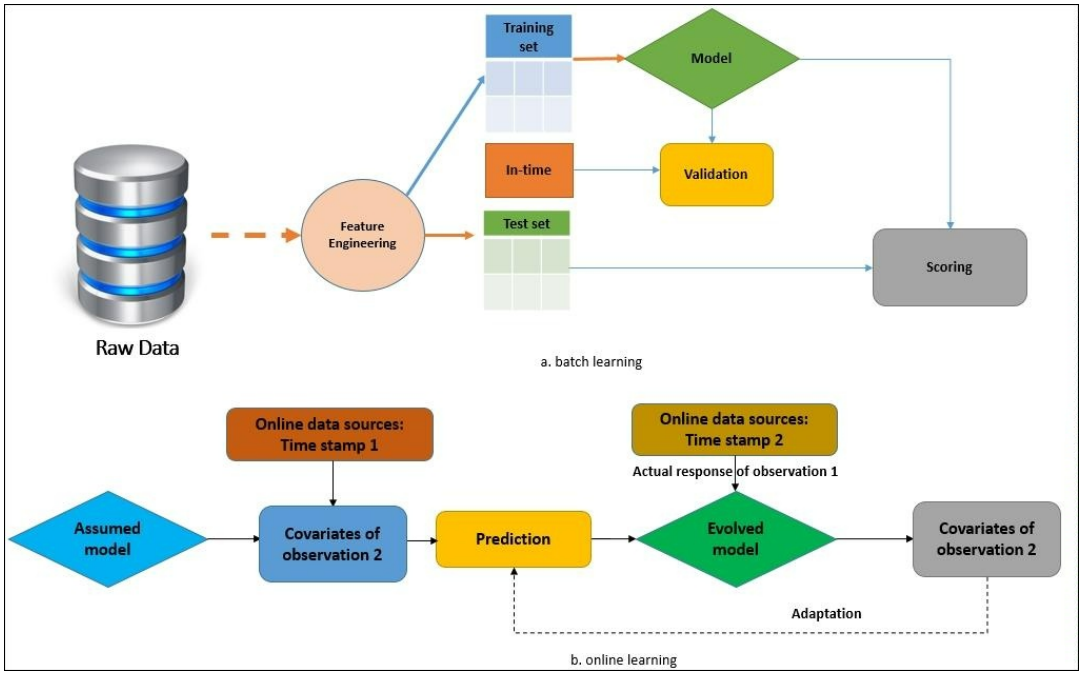


Figure 11: Batch (static) versus online learning, an overview

Statistical learning model

As already outlined, in statistically-based learning models such as **stochastic gradient descents (SGD)** and artificial neural networks or perceptron, data samples are assumed to be independent of each other. In addition to this, it is also assumed that the dataset is identically distributed as random variables. In other words, they don't adapt with time. Therefore, an ML algorithm has a limited access to the data.

In the field of the statistical learning model there are two interpretations that are considered significant:

- **First interpretation**: This considers the stochastic gradient descent method as applied to the problem of minimizing the expected risks. In an infinite stream of data, the predictive analytics is assumed to be drawn from the normal distribution. Therefore only the stochastic gradient descent method is used to bind the deviation. This interpretation is also valid for the finite training set.
- **Second interpretation:** This applies to the case of a finite training set and considers the SGD algorithm as an instance of incremental gradient descent method. In this case, one instead looks at the empirical risk: Since the gradients of in the incremental gradient descent, iterations are also stochastic estimates of the gradient of, this interpretation but applied to minimize the empirical risk as opposed to the expected risk. For why multiple passes through the data are readily allowed and actually lead to tighter bounds on the deviations.

Adversarial model

Classical machine learning, which is especially taught in classes, emphasizes a static environment where usually unchanging data is used to make predictions. It is, therefore, formally easier compared to a statistical or causal inference or dynamic environment. On the other hand, finding and solving the learning problem as a game between two players, for example, learner versus data generator in a dynamic environment, is an example of an adversarial model. This kind of modeling and making predictive analytics is critically tedious since the world does not know that you are trying to model it formally.

Furthermore, your model does not have any positive or negative effect on the world. Therefore, the ultimate goal of this kind of model is to minimize losses prevailing from circumstances generated by the move made and played by the other player. The opponent can adapt the data generated based on the output of the learning algorithm in run-time or dynamically. Since no distributional assumptions are made about the data, performing well for the entire sequence that could be viewed ahead of time becomes the ultimate goal. Additionally, regret is to be minimized on the hypothesis at the last pace. According to Cathy O. et al (*Weapons of Math Destruction*, Cathy O'Neil, and Crown, September 6, 2016) t adversarial-based machine learning can be defined as follows:

Adversarial machine learning is the formal name for studying what happens when conceding even a slightly more realistic alternative to assumptions of these types (harmlessly called **relaxing assumptions**).

Tip

Up to the Spark 2.0.0 release, there was no formal algorithm implemented in the release. Therefore, we were unable to provide any concrete examples that could be further explained elaborately. Interested readers should check the latest Spark release to understand the updates.

Summary

In this chapter, we tried to cover some advanced machine learning techniques to make machine learning models and applications adaptable for new problem and data types.

We have shown several examples of machine learning algorithms that learn from batch or static-based learning over the data of models that are updated each time they see a new training instance.

We have also discussed how to make the models adaptable through generalization, through incremental learning, through model reusing, and in dynamic environments.

In Chapter 9, *Advanced Machine Learning with Streaming and Graph Data*, we will guide you on how to apply machine learning techniques with the help of Spark MLlib and Spark ML on streaming and graph data, for example, topic modeling.

Chapter 9. Advanced Machine Learning with Streaming and Graph Data

This chapter guides the reader on how to apply machine learning techniques with the help of Spark MLlib, Spark ML and Spark Streaming to streaming and graph data using GraphX. For example, topic modeling from the real-time tweets data from Twitter. The readers will be able to use available APIs to build real-time and predictive applications from streaming data sources such as Twitter. Through the Twitter data analysis, we will show how to perform large scale social sentiment analysis. We will also show how to develop a large-scale movie recommendation using Spark MLlib, which is an implicit part of social network analysis. In a nutshell, the following topics will be covered throughout this chapter:

- Developing real-time ML pipelines
- Time series and social network analysis
- Movie recommendation using Spark
- Developing a real-time ML pipeline from streaming
- ML pipeline on graph data and semi-supervised graph-based learning

However, what it really needs in order to be an effective and emerging ML application is a continuous flow of labeled data. Consequently, preprocessing the large-scale unstructured data and accurate labeling to that data essentially introduces many unwanted latencies.

Nowadays, we hear and read a lot about real-time machine learning. More or less, people usually provide this appealing business scenario when discussing sentiment analysis from **Social Network Services** (**SNS**), credit card fraud detection systems, or mining purchase rules related to the customer from business oriented transactional data.

According to many ML experts, it is possible to continuously update the credit card fraud detection model in real time. It's fantastic, but not realistic to me, for several reasons. Firstly, ensuring the continuous flow of this kind of data is not needed for model retraining. Secondly, creating labeled data would probably be the slowest and the most expensive step in most of the machine learning systems.

Developing real-time ML pipelines

In order to develop a real-time machine learning application, we need to have access to a continuous flow of data. The data might include transactional data, simple texts, tweets from Twitter, messaging or

streaming from Flume or Kafka, and so on, as this is mostly unstructured data.

To deploy these kinds of ML applications, we need to go through a series of steps. The most unreliable source of the data that would serve our purpose is the real-time data from several sources. Often networks are a performance bottleneck.

For example, it's not guaranteed that you will always receive a bunch of tweets from Twitter. Moreover, labeling this data towards building an ML model on the fly is not a realistic idea. Nevertheless, here we provide a real insight on how we could develop and deploy an ML pipeline from real-time streaming data. *Figure 1* shows the workflow of a real-time ML application development.

Streaming data collection as unstructured text data

We would like to stress here that real time stream data collection depends on:

- The purpose of the data collection. If the purpose is to develop a credit card fraud detection online, then the data should be collected from your own network through the web API. If the purpose is to collect social media sentiment analysis then data could be collected from Twitter, LinkedIn, Facebook, or newspaper sites, and if the purpose is to network anomaly detection, data could be collected from the network data.
- Data availability is an issue since not all social media platforms provide public APIs for collecting data. The network condition is important since stream data is huge and needs very fast network connectivity.
- Storage capability is an important consideration since a collection of a few minutes of tweets data, for example, could contribute to several GB of data.

Moreover, we should wait at least a couple of days before marking the transactions as *Fraud* or *Not Fraud*, for example. In contrast, if somebody reported a fraud transaction, we can immediately label this transaction as *Fraud* for the sake of simplicity.

Labeling the data towards making the supervised machine learning

The labeled data set plays a central role in the whole process. It ensures that it is very easy to change the parameters of an algorithm such as the feature normalization or loss function. In that case, we would have several

options of choosing the algorithm itself from logistic regression, to **Support Vector Machine** (**SVM**), or random forest, for example.

However, we cannot change the labeled data set since this information is predefined and your model should predict the labels that you already have. In previous chapters, we have shown that labeling the structured data takes a considerable amount of time.

Now think about the fully unstructured stream data that we would be receiving from streaming or real-time sources. In that case, labeling the data would take a considerable amount of time. Nevertheless, we will also have to do the pre-processing, such as tokenization, cleaning, indexing, removing stop words, and removing special characters from the unstructured data.

Now, essentially, there would be a question of *how long does the data labeling process take?* The final thing about the labeled dataset is that we should understand that the labeled dataset might be biased sometimes if we don't do the labeling carefully, which might lead to a lot of issues with the model's performance.

Creating and building the model

For training the sentiment analytics, the credit card fraud detection model, and the association rule mining model, we need to have a lot of examples of transaction data that is as accurate as possible. Once we have the labeled dataset, we are ready to train and build the model:

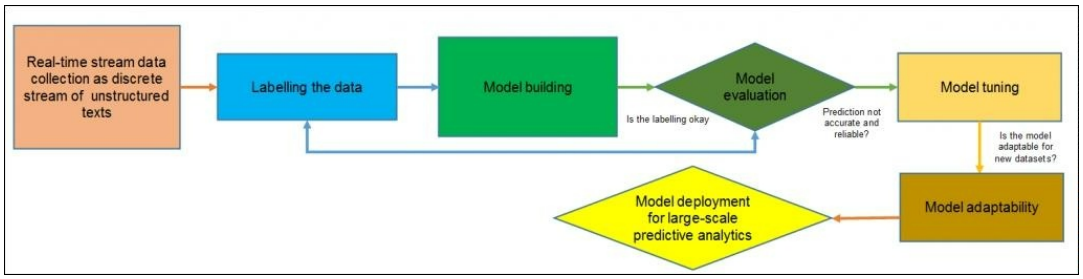


Figure 1: Real-time machine learning workflow.

In Chapter 7, *Tuning Machine Learning Models*, we discussed how to choose appropriate models and ML algorithms in order to produce better predictive analytics. The model can be presented as a binary or multiclass classifier with several classes. Alternatively, use an LDA model for the sentiment analysis using the topic modeling concept. In a nutshell, *Figure 1* shows a real-time machine learning workflow.

Real-time predictive analytics

When your ML model is properly trained and built, your model is ready for doing the real-time predictive analytics. If you get a good prediction from the

model it would be fantastic. However, as we previously mentioned when discussing some accuracy issues such as true positives and false positives, if the number of false positives is high then that means the performance of the model is not satisfactory.

This essentially means three things: we have not properly labeled the stream dataset, in that case iterate step two (labeling the data in *Figure 1*), or have not selected the proper ML algorithm to train the model, and finally we have not tuned what would eventually help us to find the appropriate hyperparameters or model selection; in that case, go straight to step seven (model deployment in *Figure 1*).

Tuning the ML model for improvement and model evaluation

As mentioned in step four (model evaluation in *Figure 1*), if the performance of the model is not satisfactory or convincing enough, then we need to tune the model. As discussed in Chapter 7, *Tuning Machine Learning Models*, we learned about how to choose the appropriate model and ML algorithms in order to produce better predictive analytics. There are several techniques for tuning the models, performance and we can go for them based on requirements and the situation. When we have done the tuning, finally we should do the model evaluation.

Model adaptability and deployment

When we have the tuned and found best model, the machine learning model has to be prepared in order to learn incrementally over the new data types when the model is updated each time it sees a new training instance. When we have our model ready for making the accurate and reliable prediction for the large-scale streaming data, we can deploy it in real life.

Time series and social network analysis

In this section, we will try to provide some insights and challenges of dealing and developing a large-scale ML pipeline from the time series and social network data.

Time series analysis

Time series data often arises, when, for example, monitoring industrial processes or tracking corporate business metrics. One of the fundamental differences between modeling data via time series methods is that time series analysis accounts for the fact that data points taken over time may have an internal structure.

This might include autocorrelation, trends, or seasonal variation that should be taken into account. Regression analysis in this regard is mostly used to test the theories. The target is to test to make sure that the current values of one or more independent times series parameters are correlated to the current properties of other time series data.

To develop large scale predictive analytics applications, time series analysis techniques can be applied to real-valued, categorical variables, continuous data, discrete numeric data, or even discrete symbolic data. A time series is a sequence of floating-point values, each linked to a timestamp. In particular, we try as hard as possible to stick with *time series* as meaning a univariate time series, although in other contexts, it sometimes refers to a series of multiple values at the same timestamp.

An instant in the time series data is the vector of values in a collection of time series corresponding to a single point in time. An observation is a tuple (timestamp, key, value), that is, a single value in a time series or instant. In a nutshell, a time series has mainly four characteristics:

- **Series with trends**: Since observations increase or decrease over time, although the trends are persistent and have long term movement.
- **Series data with seasonality**: Since observations stay high then drop off and some patterns repeat from one period to the next, and contain regular periodic fluctuations, say within a 12 month period.
- **Series data with cyclic component**: Since the business model changes periodically, that is, recessions in the business occur in a cyclic order sometimes. It also might contain the repeating swings or movements over more than one year.

- **Random variation**: An unpredictable component that gives time series graphs an irregular or zigzag appearance. It also contains erratic or residual fluctuations.

Because of these kinds of challenging characteristics, it really becomes difficult to develop practical machine learning applications for practical purposes. Hence, until now, there is only one package available for time series data analysis, developed by Cloudera; it is called the Spark-TS library. Here each time series is typically labeled with a key that enables identifying it among a collection of time series.

However, the current implementation of Spark does not provide any implemented algorithm for the time series data analysis. However, since it is an emerging and trending topic of interest, hopefully, we will have at least some algorithms implemented in Spark in coming releases. In Chapter 10, *Configuring and Working with External Libraries*, we will provide more insight into how to use these kinds of third-party packages with Spark.

Social network analysis

A social network is made up of **nodes** (points) and associated **links**,where nodes, links, or edges are then identifiable categories of analysis. These nodes might include the information about the people, groups, and organizations. Typically, this information is usually the main priority and concern for any type of social experimentation and analysis. The links in this kind of analysis focus on the collective way to include social contacts and exchangeable information to expand social interaction, such as Facebook, LinkedIn, Twitter, and so on. Therefore, it is obvious that organizations that are embedded in networks of larger social processes, links, and nodes, influence the others.

On the other hand, according to Otte E.et al. (*Social network analysis: A powerful strategy, also for the information sciences*, Journal of Information Science, 28: 441-453), **Social Network Analysis** (**SNA**) is the study of finding the mapping and measuring the relationships between the connected people or groups. It is also used to find the flows between people, groups, organizations, and information processing entities.

A proper SNA analysis could be used to show the distinction between the three most popular individual centrality measures: degree centrality, betweenness centrality, and closeness centrality. **Degree centrality** signifies how many links or incidents a node has or how many ties a node has.

The **betweenness centrality** is a centrality measure of a vertex within a graph. This also considers the edge of betweenness. Moreover, the

betweenness centrality signifies the number of times a node acts as a bridge by considering the shortest paths between other nodes.

On the other hand, the **closeness centrality** of a node is the average length of the shortest path between a particular node and all other nodes in a connected graph, such as a social network.

> ## Tip
>
> Interested readers are recommended to read more about the eigenvector centrality, Katz centrality, PageRank centrality, Percolation centrality, Cross-clique centrality, and alpha centrality for the proper understanding of statistical as well as social network centrality.

The social network is often represented as connected graphs (directed or undirected). As a result, it also involves graph data analysis, where people act as nodes and the connections or links act as the edges. Moreover, collecting and analyzing large-scale data and later on developing predictive and descriptive analytics applications from the social network, such as Facebook, Twitter, and LinkedIn also involve social network data analysis, including: link prediction such as predicting relations or friendship, determining communities in social networks such as clustering on graphs, and determining opinion leaders in networks, which is essentially a PageRank problem if the proper structure is done on a graph data.

Spark has its dedicated API for the analysis of graphs, which is called GraphX. This API can be used, for instance, to search for spam, rank search results, determine communities in social networks, or search for opinion leaders, and it's not a complete list of applying methods for analyzing graphs. We will discuss using the GraphX later in this chapter in more detail.

Movie recommendation using Spark

Model-based collaborative filtering is commonly being used by many companies, such as Netflix, as a recommender system for a real-time movie recommendation. In this section, we will see a complete example of how it works towards recommending movies for new users.

Model-based movie recommendation using Spark MLlib

The implementation in Spark MLlib supports the model-based collaborative filtering. In the model based collaborative filtering technique, users and products are described by a small set of factors, also called the **latent factors** (**LFs**).The LFs are then used for predicting the missing entries. Spark API provides the implementation of the **Alternating Least Squares** (also known as the **ALS** widely) algorithm, which is used to learn these latent factors by considering six parameters, including:

- `numBlocks`
- `rank`
- `iterations`
- `lambda`
- `implicitPrefs`
- `alpha`

To learn more about these parameters, refer to the recommendation system section in Chapter 5, *Supervised and Unsupervised Learning by Examples*. Note that to construct an ALS instance with default parameters, you can set the value based on your requirements. The default values are as follows: `numBlocks`: -1, `rank`: 10, `iterations`: 10, `lambda`: 0.01, `implicitPrefs`: false, `alpha`: 1.0.

The construction of an ALS instance, in short, is as follows:

- At first, the ALS, which is an iterative algorithm, is used to model the rating matrix as the multiplication of low-ranked users and product factors
- After that, the learning task is done by using these factors by minimizing the reconstruction error of the observed ratings

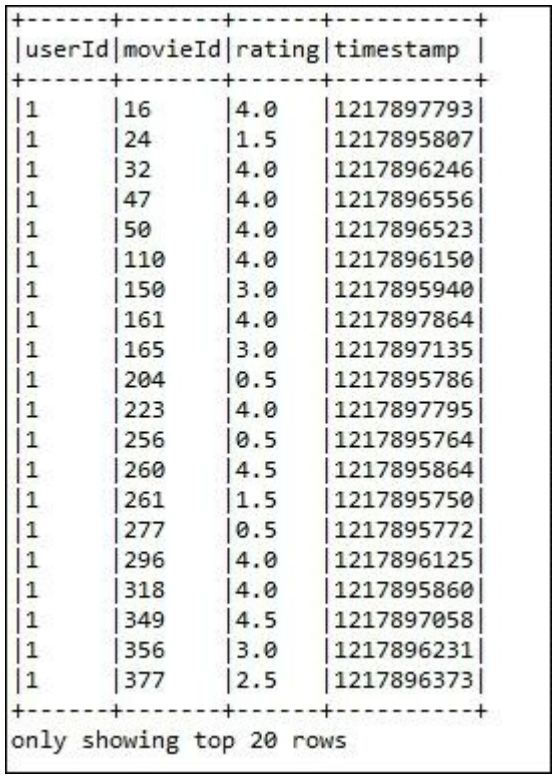
However, the unknown ratings can successively be calculated by multiplying these factors together.

The approach for a movie recommendation or any other recommendation based on the collaborative filtering technique used in the Spark MLlib has been proven to be a high performer with high prediction accuracy and scalable for the billions of ratings on commodity clusters used by companies such as Netflix. By following this approach, a company such as Netflix can recommend movies to its subscriber based on the predicted ratings. The ultimate target is to increase the sales, and of course, the customer satisfaction.

Data exploration

The movie and the corresponding rating dataset were downloaded from the MovieLens website (https://movielens.org). According to the data description on the MovieLens website, all the ratings are described in the `ratings.csv` file. Each row of this file followed by the header represents one rating for one movie by one user.

The CSV dataset has the following columns: `userId`, `movieId`, `rating`, and `timestamp` as shown in *Figure 2*. The rows are ordered first by the `userId` within the user, by `movieId`. Ratings are made on a five-star scale; with half-star increments (0.5 stars up to 5.0 stars). The timestamps represent the seconds since midnight **Coordinated Universal Time** (**UTC**) of January 1, 1970, where we have 105,339 ratings from the 668 users on 10,325 movies:



userId	movieId	rating	timestamp
1	16	4.0	1217897793
1	24	1.5	1217895807
1	32	4.0	1217896246
1	47	4.0	1217896556
1	50	4.0	1217896523
1	110	4.0	1217896150
1	150	3.0	1217895940
1	161	4.0	1217897864
1	165	3.0	1217897135
1	204	0.5	1217895786
1	223	4.0	1217897795
1	256	0.5	1217895764
1	260	4.5	1217895864
1	261	1.5	1217895750
1	277	0.5	1217895772
1	296	4.0	1217896125
1	318	4.0	1217895860
1	349	4.5	1217897058
1	356	3.0	1217896231
1	377	2.5	1217896373

only showing top 20 rows

Figure 2: Sample ratings for the top 20 movies.

On the other hand, the movie information is contained in the `movies.csv` file. Each row apart from the header information represents one movie containing the columns: `movieId`, `title`, and `genres`.

Movie titles are either created or inserted manually or imported from the website of the movie database at https://www.themoviedb.org/. The release year, however, is shown in the bracket.

Since movie titles are inserted manually, some errors or inconsistencies may exist in these titles. Readers are therefore recommended to check the IMDb database (http://www.ibdb.com/) to make sure if there are no inconsistencies or incorrect titles with their corresponding release year.

Genres are a separated list, and are selected from the following genre categories:

- Action, Adventure, Animation, Children's, Comedy, Crime
- Documentary, Drama, Fantasy, Film-Noir, Horror, Musical
- Mystery, Romance, Sci-Fi, Thriller, Western, War

```
+-------+-----------------------------------+-------------------------------------------+
|movieId|title                              |genres                                     |
+-------+-----------------------------------+-------------------------------------------+
|1      |Toy Story (1995)                   |Adventure|Animation|Children|Comedy|Fantasy|
|2      |Jumanji (1995)                     |Adventure|Children|Fantasy                 |
|3      |Grumpier Old Men (1995)            |Comedy|Romance                             |
|4      |Waiting to Exhale (1995)           |Comedy|Drama|Romance                       |
|5      |Father of the Bride Part II (1995) |Comedy                                     |
|6      |Heat (1995)                        |Action|Crime|Thriller                      |
|7      |Sabrina (1995)                     |Comedy|Romance                             |
|8      |Tom and Huck (1995)                |Adventure|Children                         |
|9      |Sudden Death (1995)                |Action                                     |
|10     |GoldenEye (1995)                   |Action|Adventure|Thriller                |
|11     |American President, The (1995)     |Comedy|Drama|Romance                       |
|12     |Dracula: Dead and Loving It (1995) |Comedy|Horror                              |
|13     |Balto (1995)                       |Adventure|Animation|Children               |
|14     |Nixon (1995)                       |Drama                                      |
|15     |Cutthroat Island (1995)            |Action|Adventure|Romance                   |
|16     |Casino (1995)                      |Crime|Drama                                |
|17     |Sense and Sensibility (1995)       |Drama|Romance                              |
|18     |Four Rooms (1995)                  |Comedy                                     |
|19     |Ace Ventura: When Nature Calls (1995)|Comedy                                   |
|20     |Money Train (1995)                 |Action|Comedy|Crime|Drama|Thriller       |
+-------+-----------------------------------+-------------------------------------------+
only showing top 20 rows
```

Figure 3: The title and genres for the top 20 movies.

Movie recommendation using Spark MLlib

In this subsection, we will show you how to recommend the movie for other users through a step-by-step example from data collection to movie recommendation. Download the `movies.csv` and `ratings.csv` files from Packt supplementary documents and place them in your project directory.

Step 1: Configure your Spark environment

Here is the code to configure your Spark environment:

```
static SparkSession spark = SparkSession
      .builder()
      .appName("JavaLDAExample")
         .master("local[*]")
         .config("spark.sql.warehouse.dir",
"E:/Exp/").
          getOrCreate();
```

Step 2: Load, parse, and explore the movie and rating Dataset

Here is the code illustrated:

```
String ratigsFile = "input/ratings.csv";
Dataset<Row> df1 =
spark.read().format("com.databricks.spark.csv").option("header",
"true").load(ratigsFile);
Dataset<Row> ratingsDF =
df1.select(df1.col("userId"), df1.col("movieId"),
df1.col("rating"),  df1.col("timestamp"));
ratingsDF.show(false);
```

This code segment should return you the `Dataset<Row>` of the ratings same as in *Figure 2*. On the other hand, the following code segment shows you the `Dataset<Row>` of movies, same as in *Figure 3*:

```
String moviesFile = "input/movies.csv";
Dataset<Row> df2 =
spark.read().format("com.databricks.spark.csv").option("header",
"true").load(moviesFile);
Dataset<Row> moviesDF =
df2.select(df2.col("movieId"), df2.col("title"),
df2.col("genres"));
moviesDF.show(false);
```

Step 3: Register both Datasets as temp tables

To register both Datasets, we can use the following code:

```
ratingsDF.createOrReplaceTempView("ratings");
moviesDF.createOrReplaceTempView("movies");
```

This will help to make the in-memory querying faster by creating a temporary view as a table in min-memory. The lifetime of the temporary

table using the `createOrReplaceTempView()` method is tied to the `[[SparkSession]]` that was used to create this Dataset.

Step 4: Explore and query for related statistics

Let's check the ratings related statistics. Just use the following code lines:

```
long numRatings = ratingsDF.count();
long numUsers =
ratingsDF.select(ratingsDF.col("userId")).distinct().count();
long numMovies =
ratingsDF.select(ratingsDF.col("movieId")).distinct().count();
System.out.println("Got " + numRatings + "
ratings from " + numUsers + " users on " +
numMovies + " movies.");
```

You should find 105,339 ratings from 668 users on 10,325 movies. Now, let's get the maximum and minimum ratings along with the count of users who have rated a movie.

However, you need to perform a SQL query on the rating table we just created in-memory in the previous step. Making a query here is simple, and it is similar to making a query from a MySQL database or RDBMS.

However, if you are not familiar with SQL-based queries, you are suggested to look at the SQL query specification to find out how to perform a selection using `SELECT` from a particular table, how to perform the ordering using `ORDER`, and how to perform a joining operation using the `JOIN` keyword.

Well, if you know the SQL query, you should get a new Dataset by using a complex SQL query as follows:

```
Dataset<Row> results = spark.sql("select
movies.title, movierates.maxr, movierates.minr,
movierates.cntu " + "from(SELECT
ratings.movieId,max(ratings.rating) as maxr,"  +
"min(ratings.rating) as minr,count(distinct
userId) as cntu "  + "FROM ratings group by
ratings.movieId) movierates " + "join movies on
movierates.movieId=movies.movieId " + "order by
movierates.cntu desc");
results.show(false);
```

```
Got 105339 ratings from 668 users on 10325 movies.
+-----------------------------------------------------------------------------+----+----+----+
|title                                                                        |maxr|minr|cntu|
+-----------------------------------------------------------------------------+----+----+----+
|Pulp Fiction (1994)                                                          |5.0 |0.5 |325 |
|Forrest Gump (1994)                                                          |5.0 |0.5 |311 |
|Shawshank Redemption, The (1994)                                             |5.0 |0.5 |308 |
|Jurassic Park (1993)                                                         |5.0 |1.0 |294 |
|Silence of the Lambs, The (1991)                                             |5.0 |0.5 |290 |
|Star Wars: Episode IV - A New Hope (1977)                                    |5.0 |0.5 |273 |
|Matrix, The (1999)                                                           |5.0 |0.5 |261 |
|Terminator 2: Judgment Day (1991)                                            |5.0 |0.5 |253 |
|Braveheart (1995)                                                            |5.0 |0.5 |248 |
|Schindler's List (1993)                                                      |5.0 |0.5 |248 |
|Fugitive, The (1993)                                                         |5.0 |1.0 |244 |
|Toy Story (1995)                                                             |5.0 |1.0 |232 |
|Star Wars: Episode V - The Empire Strikes Back (1980)                        |5.0 |0.5 |228 |
|Usual Suspects, The (1995)                                                   |5.0 |1.0 |228 |
|Raiders of the Lost Ark (Indiana Jones and the Raiders of the Lost Ark) (1981)|5.0 |1.0 |224 |
|Star Wars: Episode VI - Return of the Jedi (1983)                            |5.0 |0.5 |222 |
|Batman (1989)                                                                |5.0 |0.5 |217 |
|American Beauty (1999)                                                       |5.0 |1.0 |216 |
|Back to the Future (1985)                                                    |5.0 |1.5 |213 |
|Godfather, The (1972)                                                        |5.0 |1.0 |210 |
+-----------------------------------------------------------------------------+----+----+----+
only showing top 20 rows
```

Figure 4: Maximum and minimum ratings along with the count of users who have rated a movie.

To get an insight, we need to know more about the users and their ratings. Now let's find the top most active users and how many times they rated a movie:

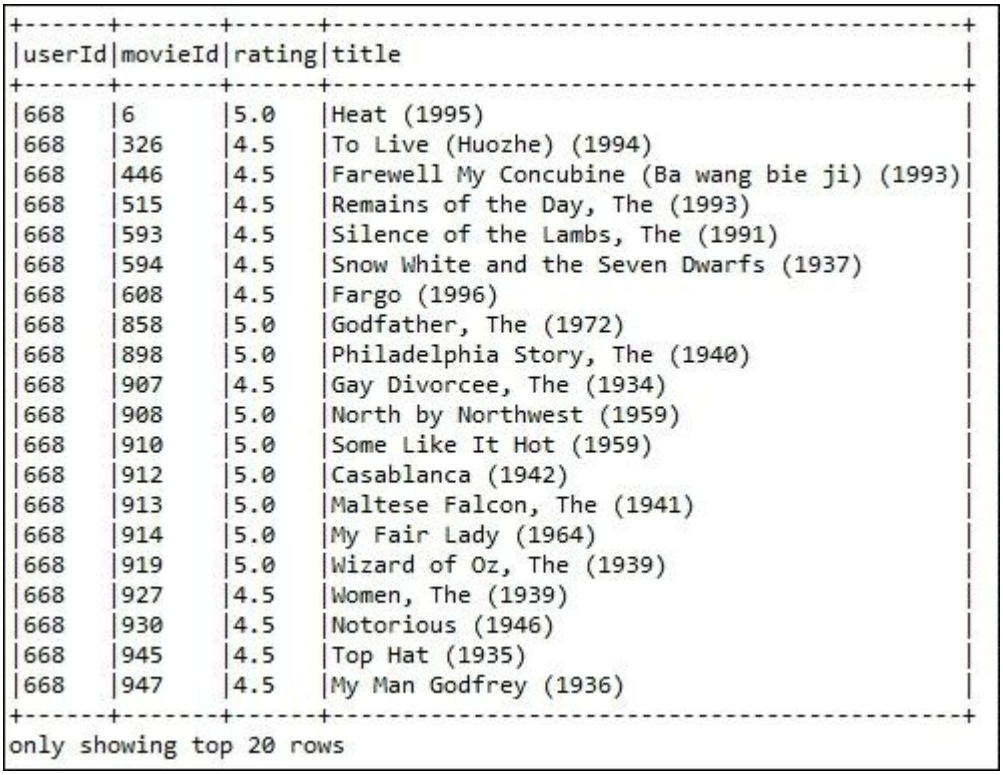
```
Dataset<Row> mostActiveUsersSchemaRDD =
spark.sql("SELECT ratings.userId, count(*) as ct
from ratings " + "group by ratings.userId order
by ct desc limit 10");
mostActiveUsersSchemaRDD.show(false);
```

```
+------+----+
|userId|ct  |
+------+----+
|668   |5678|
|575   |2837|
|458   |2086|
|232   |1421|
|310   |1287|
|475   |1249|
|128   |1231|
|224   |1182|
|607   |1176|
|63    |1107|
+------+----+
```

Figure 5: Number of ratings provided by an individual user.

Now let's have a look at a particular user, and find the movies that, say user 668, rated higher than 4:

```
Dataset<Row> results2 = spark.sql("SELECT
ratings.userId, ratings.movieId," +
"ratings.rating, movies.title FROM ratings JOIN
movies "+ "ON movies.movieId=ratings.movieId " +
"where ratings.userId=668 and ratings.rating >
4");
results2.show(false);
```



userId	movieId	rating	title
668	6	5.0	Heat (1995)
668	326	4.5	To Live (Huozhe) (1994)
668	446	4.5	Farewell My Concubine (Ba wang bie ji) (1993)
668	515	4.5	Remains of the Day, The (1993)
668	593	4.5	Silence of the Lambs, The (1991)
668	594	4.5	Snow White and the Seven Dwarfs (1937)
668	608	4.5	Fargo (1996)
668	858	5.0	Godfather, The (1972)
668	898	5.0	Philadelphia Story, The (1940)
668	907	4.5	Gay Divorcee, The (1934)
668	908	5.0	North by Northwest (1959)
668	910	5.0	Some Like It Hot (1959)
668	912	5.0	Casablanca (1942)
668	913	5.0	Maltese Falcon, The (1941)
668	914	5.0	My Fair Lady (1964)
668	919	5.0	Wizard of Oz, The (1939)
668	927	4.5	Women, The (1939)
668	930	4.5	Notorious (1946)
668	945	4.5	Top Hat (1935)
668	947	4.5	My Man Godfrey (1936)

only showing top 20 rows

Figure 6: The related rating for user 668.

Step 5: Prepare training and test rating data and see the counts

Here is the code illustrated:

```
Dataset<Row> [] splits =
ratingsDF.randomSplit(new double[] { 0.8, 0.2 });
Dataset<Row> trainingData = splits[0];
Dataset<Row> testData = splits[1];
long numTraining = trainingData.count();
long numTest = testData.count();
System.out.println("Training: " + numTraining + "
test: " + numTest);
```

You should find that there are 84,011 ratings in the training and 21,328 ratings in the test Dataset.

Step 6: Prepare the data for building the recommendation model using ALS

The following code illustrates for building the recommendation model using APIs:

```
JavaRDD<Rating> ratingsRDD =
trainingData.toJavaRDD().map(new Function<Row,
Rating>() {
      @Override
      public Rating call(Row r) throws Exception
{
        // TODO Auto-generated method stub
        int userId =
Integer.parseInt(r.getString(0));
        int movieId =
Integer.parseInt(r.getString(1));
        double ratings =
Double.parseDouble(r.getString(2));
        return new Rating(userId, movieId,
ratings);
      }
    });
```

The `ratingsRDD` RDD will contain the `userId`, `movieId`, and corresponding ratings from the training dataset that we prepared in the previous step. On the other hand, the following `testRDD` also contains the same information coming from the test Dataset we prepared in the previous step:

```
JavaRDD<Rating> testRDD =
testData.toJavaRDD().map(new Function<Row,
Rating>() {
      @Override
      public Rating call(Row r) throws Exception
{
        int userId =
Integer.parseInt(r.getString(0));
        int movieId =
Integer.parseInt(r.getString(1));
        double ratings =
Double.parseDouble(r.getString(2));
        return new Rating(userId, movieId,
ratings);
      }
    });
```

Step 7: Build an ALS user product matrix

Build an ALS user matrix model based on the `ratingsRDD` by specifying the rank, iterations, and lambda:

```
int rank = 20;
int numIterations = 10;
double lambda = 0.01;
MatrixFactorizationModel model =
ALS.train(JavaRDD.toRDD(ratingsRDD), rank,
numIterations, 0.01);
```

- Note that we have randomly selected the value of rank as `20` and have iterated the model for learning for 10 times and the lambda as `0.01`. With this setting, we got a good prediction accuracy. Readers are suggested to apply the hyper-parameter tuning to get to know the most optimum values for these parameters.
- However, readers are suggested to change the value of these two parameters based on their dataset. Moreover, as mentioned earlier, they also can use and specify other parameters such as `numberblock`, `implicitPrefs`, and `alpha` if the prediction performance is not satisfactory. Furthermore, set the number of blocks for both user blocks and product blocks to parallelize the computation into `pass -1` for an auto-configured number of blocks. The value is `-1`.

Step 8: Making predictions

Let's get the top six movie predictions for user 668:

```
System.out.println("Rating:
(UserID, MovieId, Rating)");
Rating[] topRecsForUser =
model.recommendProducts(668, 6);
for (Rating rating : topRecsForUser)
System.out.println(rating.toString());
```

```
Rating:(UserID, MovieId, Rating)
---------------------------------
Rating(668,5304,5.02697270645148)
Rating(668,93040,5.02697270645148)
Rating(668,101862,5.02697270645148)
Rating(668,25961,5.02697270645148)
Rating(668,116136,5.02697270645148)
Rating(668,25795,5.023343634020313)
```

Figure 7: Top six movies rated by the user 668.

Step 9: Get the predicted ratings to compare with the test ratings

Here is the code illustrated:

```
JavaRDD<Tuple2<Object, Object>>
testUserProductRDD = testData.toJavaRDD()
        .map(new Function<Row, Tuple2<Object,
Object>>() {
          @Override
          public Tuple2<Object, Object> call(Row
r) throws Exception {

            int userId =
Integer.parseInt(r.getString(0));
            int movieId =
Integer.parseInt(r.getString(1));
            double ratings =
Double.parseDouble(r.getString(2));
            return new Tuple2<Object,
Object>(userId, movieId);
          }
        });
JavaRDD<Rating> predictionsForTestRDD =
model.predict(JavaRDD.toRDD(testUserProductRDD)).toJavaRDD();
```

Now let's check the top 10 prediction for 10 users:

```
System.out.println(predictionsForTestRDD.take(10).toString());
```

Step 10: Prepare predictions

Here we will prepare the predictions related to RDDs in two steps. The first step includes preparing predictions for comparison from the `predictionsForTestRDD` RDD structure. It goes as follows:

```
JavaPairRDD<Tuple2<Integer, Integer>, Double>
predictionsKeyedByUserProductRDD =
JavaPairRDD.fromJavaRDD(
        predictionsForTestRDD.map(new
Function<Rating, Tuple2<Tuple2<Integer, Integer>,
Double>>() {
          @Override
          public Tuple2<Tuple2<Integer, Integer>,
Double> call(Rating r) throws Exception {
            return new Tuple2<Tuple2<Integer,
Integer>, Double>(
                new Tuple2<Integer,
Integer>(r.user(), r.product()), r.rating());
          }
        }));
```

The second step includes preparing the test for comparison:

```
JavaPairRDD<Tuple2<Integer, Integer>, Double>
testKeyedByUserProductRDD = JavaPairRDD
```

```
 .fromJavaRDD(testRDD.map(new Function<Rating,
Tuple2<Tuple2<Integer, Integer>, Double>>() {
          @Override
          public Tuple2<Tuple2<Integer, Integer>,
Double> call(Rating r) throws Exception {
            return new Tuple2<Tuple2<Integer,
Integer>, Double>(
                new Tuple2<Integer,
Integer>(r.user(), r.product()), r.rating());
          }
        }));
```

Step 11: Join the test with predictions and see the combined ratings

Join `testKeyedByUserProductRDD` and `predictionsKeyedByUserProductRDD` RDDs to get the combined test as well as predicted ratings against each user and `movieId`:

```
JavaPairRDD<Tuple2<Integer, Integer>,
Tuple2<Double, Double>>
testAndPredictionsJoinedRDD =
testKeyedByUserProductRDD
        .join(predictionsKeyedByUserProductRDD);
System.out.println("(UserID, MovieId) => (Test
rating, Predicted rating)");
System.out.println("---------------------------------");
for (Tuple2<Tuple2<Integer, Integer>,
Tuple2<Double, Double>> t :
testAndPredictionsJoinedRDD.take(6)) {
      Tuple2<Integer, Integer> userProduct =
t._1;
      Tuple2<Double, Double>
testAndPredictedRating = t._2;
      System.out.println("(" + userProduct._1() +
"," + userProduct._2() + ") => (" +
testAndPredictedRating._1()
          + "," + testAndPredictedRating._2() +
")");
    }
```

```
(UserID, MovieId) => (Test rating, Predicted rating)
---------------------------------
(133,185) => (3.0,3.0317963327109716)
(184,509) => (4.0,5.3608016642395775)
(147,3826) => (3.0,3.3262596066608334)
(193,2968) => (1.0,3.26583464418645)
(611,2028) => (5.0,2.366458515822674)
(177,2710) => (2.5,2.420442977636526)
```

Figure 8: Combined test as well as predicted ratings against each user and movieId.

Step 12: Evaluating the model against prediction performance

Let's check the performance of the ALS model by checking the number of true positives and false positives:

```
JavaPairRDD<Tuple2<Integer, Integer>,
Tuple2<Double, Double>> truePositives =
testAndPredictionsJoinedRDD
        .filter(new
Function<Tuple2<Tuple2<Integer, Integer>,
Tuple2<Double, Double>>, Boolean>() {
          @Override
          public Boolean
call(Tuple2<Tuple2<Integer, Integer>,
Tuple2<Double, Double>> r) throws Exception {
            return (r._2._1() <= 1 && r._2._2() <
5);
          }
        });
```

Now print the number of true positive predictions. We have considered that a prediction is a true prediction when the predicted rating is less than the highest rating (that is, `5`). Consequently, if the predicted rating is more or equal to `5`, consider that prediction as a false positive:

```
for (Tuple2<Tuple2<Integer, Integer>,
Tuple2<Double, Double>> t :
truePositives.take(2)) {
      Tuple2<Integer, Integer> userProduct =
t._1;
      Tuple2<Double, Double>
testAndPredictedRating = t._2;
    }
System.out.println("Number of true positive
prediction is: "+ truePositives.count());
```

You should find the value as follows.

The number of true positive prediction is 798. Now it's time to print the statistics for the false positives:

```
JavaPairRDD<Tuple2<Integer, Integer>,
Tuple2<Double, Double>> falsePositives =
testAndPredictionsJoinedRDD
        .filter(new
Function<Tuple2<Tuple2<Integer, Integer>,
Tuple2<Double, Double>>, Boolean>() {
          @Override
          public Boolean
call(Tuple2<Tuple2<Integer, Integer>,
Tuple2<Double, Double>> r) throws Exception {
            return (r._2._1() <= 1 && r._2._2()
>= 5);
```

```
          }
        });
for (Tuple2<Tuple2<Integer, Integer>,
Tuple2<Double, Double>> t :
falsePositives.take(2)) {
      Tuple2<Integer, Integer> userProduct =
t._1;
      Tuple2<Double, Double>
testAndPredictedRating = t._2;
    }
System.out.println("Number of false positive
prediction is: "+ falsePositives.count());
```

In this particular example, we have got only 14 false positive predictions, which is outstanding. Now let's check the performance of the prediction in terms of mean absolute error calculation between the test and predictions:

```
double meanAbsoluteError = JavaDoubleRDD
 .fromRDD(testAndPredictionsJoinedRDD.values().map(new
Function<Tuple2<Double, Double>, Object>() {
          public Object call(Tuple2<Double,
Double> pair) {
            Double err = pair._1() - pair._2();
            return err * err;
          }
        }).rdd()).mean();
    System.out.printing("Mean Absolute Error:
"+meanAbsoluteError);
```

Which returns the value as follows:

```
Mean Absolute Error: 1.5800601618477566
```

Developing a real-time ML pipeline from streaming

According to the API guidelines provided by Spark at http://spark.apache.org/docs/latest/streaming-programming-guide.html, technically, Spark Streaming receives live input data streams as objects (objects could be `Java/Python/R` objects). Later on, the streams are divided into batches, which are then processed by the Spark engine to generate the final input stream in batches. To make this process even easier, Spark Streaming provides a high-level abstraction, which is also called a discretized stream or DStream.

The **DStream** represents a continuous stream of data coming from real-time streaming sources such as Twitter, Kafka, Fume, Kinesis, Sensors, or any other sources. Discretized streams can be created from these sources, alternatively, high-level operations on other DStreams can also be applied for doing that. Internally, a DStream is represented as a sequence of RDDs, that means the RDD abstraction has been reused for processing the stream of RDDs.

As already discussed in Chapter 6, *Building Large Scale Machine Learning Pipelines*, a topic modeling technique automatically infers the topics discussed and inherently places them in a collection of documents as hidden resources. This is commonly used in the **Natural Language Processing** (**NLP**) and text mining tasks. These topics can be used to analyze, summarize, and organize those documents. Alternatively, those topics can be used for featurization and dimensionality reduction in later stages of a **machine learning** (**ML**) pipeline development. The most popular topic modeling algorithms are **Latent Dirichlet Allocation** (**LDA**) and **Probabilistic Latent Semantic Analysis** (**pLSA**). Previously we discussed how to apply the LDA algorithm for the static dataset that is already available. However, if the topic modeling is prepared from the real-time streaming data, that would be great and would be more live in knowing the trends in social media such as Twitter, LinkedIn, or Facebook.

However, due to the limited API facility by Facebook or LinkedIn, it would be difficult to collect the real-time data from those social media platforms. Spark also provides the API for accessing data from Twitter, Kafka, Flume, and Kinesis. The workflow of a near real-time ML application development from streaming data should follow the workflows as presented in *Figure 9*:

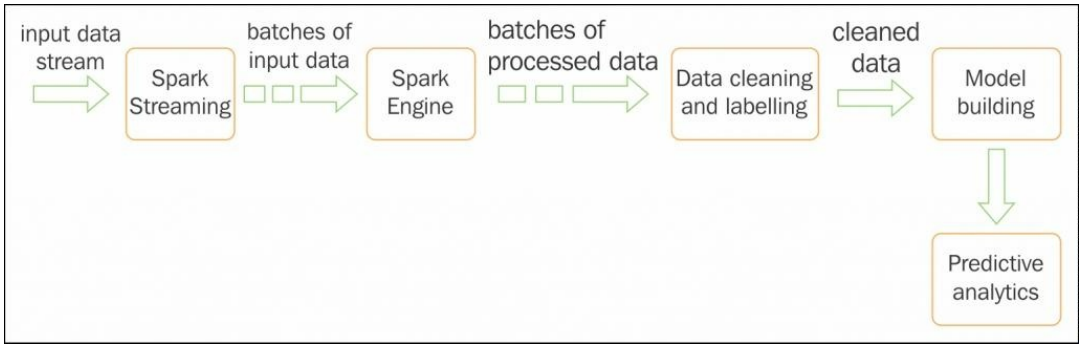


Figure 9: Real-time predictive ML model development from streaming data using Spark.

In this section, we will show you how to develop a real-time ML pipeline that handles streaming data. More specifically, we will show a step-by-step topic modeling from Twitter streaming data. The topic modeling here has two steps: Twitter data collection and topic modeling using LDA.

Real-time tweet data collection from Twitter

Spark provides APIs to access real-time tweets from the Twitter timeline. The tweets can be further made by using keywords or hashtags. Alternatively, tweets can also be downloaded from someone's Twitter timeline. However, before accessing the tweets data, you will have to create a sample Twitter application on Twitter and generate four keys: `consumerKey`, `consumerSecret`, `accessToken`, and `accessTokenSecret`.

> **Tip**
>
> Note that Twitter (and some other) driver support has been removed during Spark upgrade from 1.6.2 to 2.0.0. This means streaming data collection support using Spark from less used streaming connectors, including Twitter, Akka, MQTT, and ZeroMQ has been removed in Spark 2.0.0. Therefore, it i€™s not possible to develop an application for Twitter data collection using Spark 2.0.0. Consequently, Spark 1.6.1 will be used for the demonstration for Twitter data collection in this section. Readers are suggested to create a Maven project on Eclipse using the provided Maven friendly `pom.xml` file.

After authenticating your Spark ML application to collect data from Twitter, you will have to define the `JavaStreamingContext` by specifying the `SparkConf` and duration for collecting tweets. After that, tweets data can be downloaded as DStream or discrete stream through the `TwitterUtils` API of Spark once you revoke the `start()` method using the `JavaStreamingContext` object.

Upon starting to receive the tweets, you can save the tweets data on your local machine or HDFS or any other filesystem where applicable. However, the streaming will be continued until you terminate the streaming using `awaitTermination()`.

The received tweets, however, can be also be pre-processed or cleaned by using the `foreachRDD` design pattern and then can be saved to your desired location. Due to page limitation, we have limited our discussion here.

> **Tip**
>
> Moreover, interested readers should follow the API guidelines for Spark Streaming in the following web page of Spark:http://spark.apache.org/docs/latest/streaming-programming-guide.html.

Tweet collection using TwitterUtils API of Spark

In this sub-section, at first, we will show you how to collect real-time tweets data from Twitter using the `TwitterUtils` API. Then the same tweets data will be used for topic modeling in the next sub-section.

Step 1: Load required packages and APIs

Here is the code to load the required packages:

```
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaRDD;
import
org.apache.spark.api.java.function.Function;
import org.apache.spark.streaming.Duration;
import
org.apache.spark.streaming.api.java.JavaDStream;
import
org.apache.spark.streaming.api.java.JavaStreamingContext;
import
org.apache.spark.streaming.twitter.TwitterUtils;
import twitter4j.Status;
```

Step 2: Setting the Logger level

For setting the Logger level, we use the following code:

```
Logger.getLogger("org").setLevel(Level.OFF);
Logger.getLogger("akka").setLevel(Level.OFF);
```

```
Logger.getLogger("org.apache.spark").setLevel(Level.WARN);
Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF);
```

Step 3: Spark streaming environment setting

Here is the codes for Spark streaming illustrated:

```
SparkConf conf = new
SparkConf().setMaster("local[*]").setAppName("TwitterExample");
JavaStreamingContext jssc = new
JavaStreamingContext(conf, new Duration(1000));
```

Step 4: Setting the authentication for accessing Twitter data

Get the authentications values from the sample Twitter application by visiting the following URL: https://apps.twitter.com/:

```
String consumerKey = "VQINrM6ZcNqaCAawA6IN4xRTP";
String consumerSecret =
"F2OsVEuJypOZSAoNDFrWgoCHyNJNXbTr8T3yEbp9cWEYjTctye";
String accessToken =
"475468363-
IfRcZnbkEVPRw6bwXovMnw1FsbxuetvEF2JvbAvD";
String accessTokenSecret =
"vU7VtzZVyugUHO7ddeTvucu1wRrCZqFTPJUW8VAe6xgyf";
```

Note that here we have provided the same values for these four secret keys. Replace these values with your own keys accordingly. Well, now we need to set the system property using `twitter4j.oauth` for the previous four keys:

```
System.setProperty("twitter4j.oauth.consumerKey",
consumerKey);
System.setProperty("twitter4j.oauth.consumerSecret",
consumerSecret);
System.setProperty("twitter4j.oauth.accessToken",
accessToken);
System.setProperty("twitter4j.oauth.accessTokenSecret",
accessTokenSecret);
```

Step 5: Enable the check pointing

The following code shows how to enable the check pointing:

```
jssc.checkpoint("src/main/resources/twitterdata/");
```

The metadata check pointing is primarily needed for recovery from driver failures, whereas data or RDD check pointing is necessary even for basic

functioning if stateful transformations are used. For more details, please visit the following web page of Spark: http://spark.apache.org/docs/latest/streaming-programming-guide.html#checkpointing.

Step 6: Start accepting stream of tweets as a discrete stream

Let's collect only 100 tweets for simplicity and but can collect as much as you want:

```
JavaDStream<Status> tweets =
TwitterUtils.createStream(jssc);
final String
outputDirectory="src/main/resources/twitterdata/";
final long numTweetsToCollect = 100;
```

Step 7: Filter tweets and save as regular text files

The code for filter tweets is shown here:

```
tweets.foreachRDD(new Function<JavaRDD<Status>,
Void>() {
      public long numTweetsCollected = 0;
      @Override
      public Void call(JavaRDD<Status> status)
throws Exception {
        long count = status.count();
        if (count > 0) {
          status.saveAsTextFile(outputDirectory +
"/tweets_" + System.currentTimeMillis());
          numTweetsCollected += count;
          if (numTweetsCollected >=
numTweetsToCollect) {
               System.exit(0);
          }
        }
        return null;
      }
    });
```

Here we are pre-processing the tweets using the singleton method, `foreachRDD`, which accepts only filtered tweets, that is, if the status count is at least 1. When the number of collected tweets is equal or more than the number of tweets to be collected, then we exit the collection. Finally, we save the tweets as texts in the output directory.

Step 8: Controlling the streaming switch

The code for controlling the streaming switch is shown here:

```
jssc.start();
jssc.awaitTermination();
```

Eventually, we will use these texts of tweets for the topic modeling in the next step. If you recall the topic modeling in Chapter 6, *Building Scalable Machine Learning Pipelines*, we saw the corresponding term weight, topic name, and term indices. However, we also need to have the actual terms. In the next step, we will show the detailed technique of retrieving the terms extensively dependent on the vocabulary that needs to be created for that.

Topic modeling using Spark

In this sub-section, we represented a semi-automated technique of topic modeling using Spark. The following steps show the topic modeling from data reading to printing the topics along with their term-weights.

Step 1: Load necessary packages and APIs

Here is the code to load the necessary packages:

```
import java.io.Serializable;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import
org.apache.spark.api.java.JavaSparkContext;
import
org.apache.spark.api.java.function.Function;
import
org.apache.spark.api.java.function.Function2;
import
org.apache.spark.api.java.function.PairFlatMapFunction;
import
org.apache.spark.api.java.function.PairFunction;
import
org.apache.spark.ml.feature.StopWordsRemover;
import org.apache.spark.mllib.clustering.LDA;
import
org.apache.spark.mllib.clustering.LDAModel;
import org.apache.spark.mllib.linalg.Vector;
import org.apache.spark.mllib.linalg.Vectors;
import org.apache.spark.sql.SQLContext;
import scala.Tuple2;
```

Step 2: Configure the Spark environment

Here is the code to configure the Spark:

```
private transient static SparkConf sparkConf =
new
SparkConf().setMaster("local[*]").setAppName("TopicModelingLDA");
private transient static JavaSparkContext jsc =
new JavaSparkContext(sparkConf);
private transient static SQLContext sqlContext =
new org.apache.spark.sql.SQLContext(jsc);
```

Step 3: Setting the logging level

Here is the code to set the logging level:

```
Logger.getLogger("org").setLevel(Level.OFF);
Logger.getLogger("akka").setLevel(Level.OFF);
Logger.getLogger("org.apache.spark").setLevel(Level.WARN);
Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF);
```

> **Note**
>
> Note that setting the logging level that was just previously shown is optional.

Step 4: Create Java RDD and cache them in-memory

Create Java RDD and cache them for the tweets data from the previous step:

```
JavaRDD<String> data =
jsc.wholeTextFiles("src/main/resources/test/*.txt")
        .map(new Function<Tuple2<String, String>,
String>() {
          @Override
          public String call(Tuple2<String,
String> v1) throws Exception {
            return v1._2;
          }
        }).cache();
```

Step 5: Tokenize the terms

Load the list of stop-words provided by Spark and tokenize the terms by filtering them by applying three constraints: text length at least 4, not a stop

word, and making them each lower case. Note that we discussed stop-words in Chapter 6, *Building Scalable Machine Learning Pipelines*:

```
public static String[] stopwords = new
StopWordsRemover().getStopWords();
JavaRDD<String[]> tokenized = data.map(new
Function<String, String[]>() {
list.toArray(new String[0]);
      }      @Override
      public String[] call(String v1) throws
Exception {
        ArrayList<String> list = new
ArrayList<>();
        for (String s : v1.split("\\s")) {
          if (s.length() > 3 && !isStopWord(s) &&
isOnlyLetter(s))
            list.add(s.toLowerCase());
        }
        return
    });
```

Step 6: Prepare the term counts

Prepare the term counts by filtering them by applying four constraints: text length at least 4, not stop words, selecting only characters, and making them all lower case:

```
JavaPairRDD<String, Integer> termCounts = data
        .flatMapToPair(new
PairFlatMapFunction<String, String, Integer>() {
          @Override
          public Iterable<Tuple2<String,
Integer>> call(String t) throws Exception {
            ArrayList<Tuple2<String, Integer>> tc
= new ArrayList<>();
            for (String s : t.split("\\s")) {

              if (s.length() > 3
&& !isStopWord(s) && isOnlyLetter(s))
                tc.add(new Tuple2<String,
Integer>(s.toLowerCase(), 1));
            }
            return tc;
          }
        }).reduceByKey(new Function2<Integer,
Integer, Integer>() {
          @Override
          public Integer call(Integer v1, Integer
v2) throws Exception {
            return v1 + v2;
          }
        });
```

Note that here `isStopWords()` and `isOnlyLetters()` are two user defined methods that will be discussed at the end of this step.

Step 7: Sort the term counts

Sort the terms counts by applying two transformations, `sortByKey()` and `mapToPair()`:

```
JavaPairRDD<String, Integer> termCountsSorted =
termCounts
        .mapToPair(new
PairFunction<Tuple2<String, Integer>, Integer,
String>() {
          @Override
          public Tuple2<Integer, String>
call(Tuple2<String, Integer> t) throws Exception
{
            return t.swap();
          }
        }).sortByKey().mapToPair(new
PairFunction<Tuple2<Integer, String>, String,
Integer>() {
          @Override
          public Tuple2<String, Integer>
call(Tuple2<Integer, String> t) throws Exception
{
            return t.swap();
          }
        });
```

Step 8: Create the vocabulary

Create a vocabulary RDD by mapping the sorted term counts. Finally, print the key value pairs:

```
JavaRDD<String> vocabArray =
termCountsSorted.map(new Function<Tuple2<String,
Integer>, String>() {
      @Override
      public String call(Tuple2<String, Integer>
v1) throws Exception {
        return v1._1;
      }
    });
final Map<String, Long> vocab =
vocabArray.zipWithIndex().collectAsMap();
    for (Map.Entry<String, Long> entry :
vocab.entrySet()) {
      System.out.println(entry.getKey() + "/" +
entry.getValue());
    }
```

Let's see a screenshot of the vocabulary terms and their indices shown in *Figure 10*:



Figure 10: Vocabulary terms and their indices.

Step 9: Create the document matrix from the tokenized words/terms

Create the document matrix as `JavaPairRDD` from the tokenized terms by mapping the vocabulary we created in the previous step. After that, cache the RDD in-memory for faster processing:

```
JavaPairRDD<Long, Vector> documents = JavaPairRDD

 .fromJavaRDD(tokenized.zipWithIndex().map(new
Function<Tuple2<String[], Long>, Tuple2<Long,
Vector>>() {
          @Override
          public Tuple2<Long, Vector>
call(Tuple2<String[], Long> v1) throws Exception
{
            String[] tokens = v1._1;
            Map<Integer, Double> counts = new
HashMap();

            for (String s : tokens) {
              if (vocab.containsKey(s)) {
                long idx = vocab.get(s);
                int a = (int) idx;
                if (counts.containsKey(a)) {
```

```
                    counts.put(a, counts.get(a) +
1.0);
                  } else
                    counts.put(a, 0.0);
                }
              }
              ArrayList<Tuple2<Integer, Double>> ll
= new ArrayList<>();
              ArrayList<Double> dd = new
ArrayList<>();

              for (Map.Entry<Integer, Double>
entry : counts.entrySet()) {
                ll.add(new Tuple2<Integer,
Double>(entry.getKey(), entry.getValue()));
                dd.add(entry.getValue());
              }

              return new Tuple2<Long,
Vector>(v1._2, Vectors.sparse(vocab.size(), ll));
            }
          })).cache();
```

Step 10: Train the LDA model

Train the LDA model using the documents matrix from step 9 and describe 10 topic terms against four topics for simplicity.

Note that here we have used **Latent Dirichilet Allocation** (**LDA**), which is one of the most popular topic modeling algorithms commonly used for text mining. We could use more robust topic modeling algorithms such as **Probabilistic Latent Sentiment Analysis** (**pLSA**), **Pachinko Allocation Model** (**PAM**), or **Hierarchical Dirichilet Process** (**HDP**) algorithms. However, pLSA has the overfitting problem.

On the other hand, both HDP and PAM are more complex topic modeling algorithms used for complex text mining such as mining topics from high dimensional text data or documents of unstructured text. Moreover, to this date, Spark has implemented only one topic modeling algorithm, that is LDA. Therefore, we have to use LDA reasonably:

```
LDAModel ldaModel = new
LDA().setK(4).setMaxIterations(10).run(documents);
Tuple2<int[], double[]>[] topicDesces =
ldaModel.describeTopics(10);
int topicCount = topicDesces.length;
```

Note that to keep the topic generation simple, we have set the number of the topic as 4 and have iterated the LDA 10 times. Another reason is that in the next section we want to show how to connect these four topics through

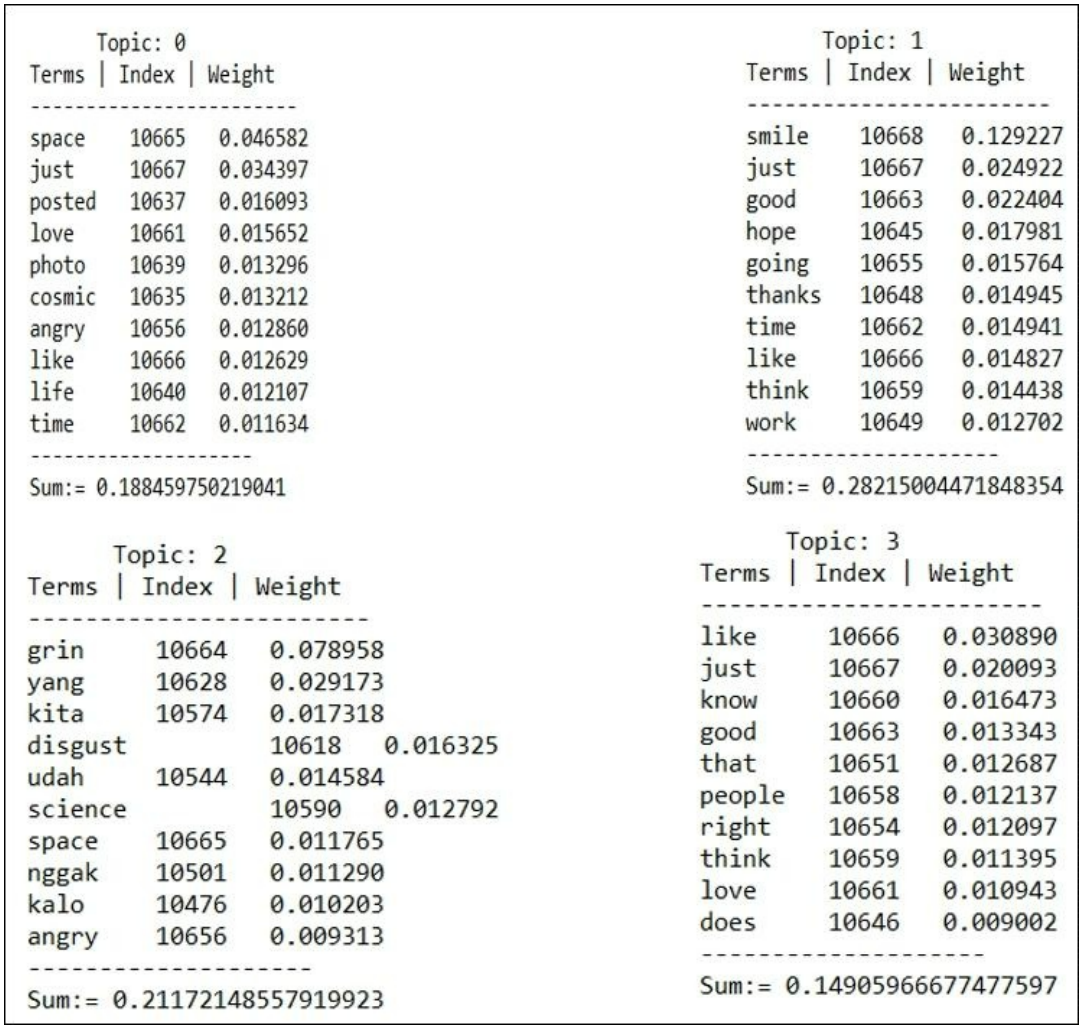
their common terms. Readers are recommended to change the value based on their requirements.

Step 11: Get the topic terms, index, term weights, and total sum across each topic

Get these statistics from the vocabulary and topic description described in step 10 and step 8:

```
for (int t = 0; t < topicCount; t++) {
      Tuple2<int[], double[]> topic =
topicDesces[t];
      System.out.println("      Topic: " + t);
      int[] indices = topic._1();
      double[] values = topic._2();
      double sum = 0.0d;
      int wordCount = indices.length;
      System.out.println("Terms |\tIndex
|\tWeight");

 System.out.println("------------------------");
      for (int w = 0; w < wordCount; w++) {
        double prob = values[w];
        int vocabIndex = indices[w];
        String vocabKey = "";
        for (Map.Entry<String, Long> entry :
vocab.entrySet()) {
          if (entry.getValue() == vocabIndex) {
            vocabKey = entry.getKey();
            break;
          } }
System.out.format("%s \t %d \t %f \n", vocabKey,
vocabIndex, prob);
        sum += prob;
      }
      System.out.println("--------------------");
      System.out.println("Sum:= " + sum);
      System.out.println();
    }  }
```

If you look at the preceding code segment carefully, the `vocabKey` indicates the corresponding topic term, `vocabIndex` is the index, and `prob` indicates the term weight for each term in a topic. The print statements have been used to format the outputs. Now let's see the output that describes four topics for simplicity in *Figure 11*:



```
      Topic: 0
Terms | Index | Weight
-----------------------
space    10665   0.046582
just     10667   0.034397
posted   10637   0.016093
love     10661   0.015652
photo    10639   0.013296
cosmic   10635   0.013212
angry    10656   0.012860
like     10666   0.012629
life     10640   0.012107
time     10662   0.011634
-------------------
Sum:= 0.188459750219041
```

```
      Topic: 1
Terms | Index | Weight
-----------------------
smile    10668   0.129227
just     10667   0.024922
good     10663   0.022404
hope     10645   0.017981
going    10655   0.015764
thanks   10648   0.014945
time     10662   0.014941
like     10666   0.014827
think    10659   0.014438
work     10649   0.012702
-------------------
Sum:= 0.28215004471848354
```

```
      Topic: 2
Terms | Index | Weight
-----------------------
grin      10664   0.078958
yang      10628   0.029173
kita      10574   0.017318
disgust           10618   0.016325
udah      10544   0.014584
science           10590   0.012792
space     10665   0.011765
nggak     10501   0.011290
kalo      10476   0.010203
angry     10656   0.009313
--------------------
Sum:= 0.21172148557919923
```

```
      Topic: 3
Terms | Index | Weight
-----------------------
like     10666   0.030890
just     10667   0.020093
know     10660   0.016473
good     10663   0.013343
that     10651   0.012687
people   10658   0.012137
right    10654   0.012097
think    10659   0.011395
love     10661   0.010943
does     10646   0.009002
--------------------
Sum:= 0.14905966677477597
```

Figure 11: Describing four topics.

As we mentioned in *step 6*, here we will show how we develop the `isStopWord()` method. Just use the following code:

```
public static boolean isStopWord(String word) {
    for (String s : stopwords) {
      if (word.equals(s))
        return true;
    }
    return false;
  }
```

And the `isOnlyLetters()` method goes as follows:

```
public static boolean isOnlyLetter(String word) {
    for (Character ch : word.toCharArray()) {
      if (!Character.isLetter(ch))
        return false;
    }
    return true;
```

```
}
```

In the next section, we will introduce how to parse and handle large-scale graph data using GraphX API of Spark to find the connected components from the topics data that we got from *Figure 10*.

ML pipeline on graph data and semi-supervised graph-based learning

Due to the big data deluge, there has been a large amount of unlabeled data, and very small amounts of labeled data. As already discussed, labeling and annotating this data is computationally expensive and an obstacle in finding real insight from the data. Also, the increasing growth of the social network and media producers graph data at scale. These data thrive to develop real-time and large-scale supervised learning methods that can use information in the input distribution.

The idea behind the graph-based semi-supervised learning is to construct a graph connecting similar data points or components. That lets the hidden and unobserved labels be random variables on the nodes of this graph. In this type of learning, similar data points can have similar labels and the information *propagates* from labeled data points to other data points. A similar restriction also limits the ability to express many important steps in typical processing and analytical pipelines. However, the graph-based learning is not optimized for an iterative diffusion technique such as PageRank since lots of computational aspects are underlying and involved.

Moreover, due to API restriction and unavailability, we will not discuss this graph-based machine learning in detail with suitable examples.

However, in this section, we will provide a graph-based semi-supervised application development, which is basically a continuation of the topic modeling that we presented in the previous section.

Introduction to GraphX

GraphX is a comparatively new component in Spark for graphs processing, graph analytics, graph visualization, and graph-parallel computation. Actually, the original computational aspect of the Spark RDD was extended by introducing a new graph abstraction layer as a resilient distributed graph computation that provides resilient properties in the graph processing and storage.

To provide the graph related computation, a set of basic operators such as subgraph, `jointVertices`, and `aggregateMessages` are exposed by the GraphX. In addition to this, it also inherited the optimized variant of the Pregel API in the GraphX implementation.

Moreover, to simplify graph analytics tasks, GraphX is being enriched and growing with a collection of graph algorithms and builders.

In the next section, we will introduce how to parse and handle large-scale graph data using the GraphX API of Spark to find the connected components from the topics data that we got from *Figure 11*.

Getting and parsing graph data using the GraphX API

In this subsection, we will show you how to parse graph data using the GraphX API and then the connected components from the graph will be described in the next subsection. Due to API limitation in GraphX, we were unable to provide the same implementation in Java, but we did so for Scala implementation.

To run the following source code, go to your Spark distribution and start the Spark shell by providing the following commands:

```
$ cd home/spark-2.0.0-bin-hadoop2.7/bin
$./spark-shell
```

Then the Spark shell will be available with the Spark session. We assume your Spark distribution is saved in the `home/spark-2.0.0-bin-hadoop2.7` path. Please change the path accordingly in order to run the Spark shell. Also, please save the topic terms shown in *Figure 11* to separate text files so that you will be able to use those terms for analyzing as graph data before you proceed to follow the next steps:

Step 1: Loading required packages and APIs

Here is the code to load the required packages:

```
package com.examples.graphs
import org.apache.spark._
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD
```

Step 2: Prepare the Spark environment

Here is the code to prepare the Spark environment:

```
val conf = new
SparkConf().setAppName("GraphXDemo").setMaster("local[*]")
val sc = new SparkContext(conf)
```

Step 3: Parse the topic's terms and tokenize them

The following code illustrates to parse he topic's terms:

```
val corpus: RDD[String] =
sc.wholeTextFiles("home/ /topics/*.txt").map(_._2)
val tokenized: RDD[Seq[String]] =
corpus.map(_.toLowerCase.split("\\s"))
tokenized.foreach { x => println(x) }
```

Step 4: Create RDDs of documents

Create the RDDs of documents as a RDD[(`DocumentID`, (`nodeName`, `wordCount`))] notation. For example, RDD[(1L, (Topic_0, 4))]:

```
val nodes: RDD[(VertexId, (String, Long))] =
tokenized.zipWithIndex().map{
      case (tokens, id) =>
        val nodeName="Topic_"+id;
        (id, (nodeName, tokens.size))

    }
    nodes.collect().foreach{
      x =>
        println(x._1+":
("+x._2._1+","+x._2._2+")")
    }
```

The preceding print method generates the output as shown in Figure 12:

```
0: (Topic_0,10)
1: (Topic_1,10)
2: (Topic_2,10)
3: (Topic_3,10)
```

Figure 12: The nodes.

Step 5: Make a word document pair

Here is the code to make a word document pair:

```
val wordPairs: RDD[(String, Long)] =
      tokenized.zipWithIndex.flatMap {
        case (tokens, id) =>
          val list = new Array[(String,
Long)](tokens.size)

          for (i <- 0 to tokens.length - 1) {
            //tokens.foreach { term =>
            list(i) = (tokens(i), id)
          }
```

```
            list.toSeq
          }
        wordPairs.collect().foreach(x => println(x))
        println(wordPairs.count())
```

Step 6: Create the graph relationships between nodes

The following code shows to create a graph relationships between nodes:

```
val relationships: RDD[Edge[String]] =
wordPairs.groupByKey().flatMap{
      case(edge, nodes)=>
        val nodesList = nodes.toArray
        val list = new
Array[Edge[String]]
(nodesList.length * nodesList.length)
        if (nodesList.length>1){

          var count:Int=0;
          for (i <- 0 to nodesList.length-2) {
            for(j<-i+1 to nodesList.length-1){
         list(count) = new Edge(nodesList(i),
nodesList(j), edge)
         //list(count+1) = new Edge(nodesList(j),
nodesList(i), edge)
              count += 1;
              //count += 2;
            }
          }
        }
        list.toSeq
    }.filter { x => x!=null }
    relationships.collect().foreach { x =>
println(x) }
```

Note; if you want to make the graph connected, but not undirected, then just enable the following line:

```
list(count
+1) = new Edge(nodesList(j), nodesList(i), edge)
```

Immediately after the following line:

```
list(count) = new Edge(nodesList(i),
nodesList(j), edge)
```

Increment the count by 2, that is, `count += 2` to make the changes consistent.

Step 7: Initialize the graph

The code illustrated here shows how to illustrate the graph:

```
val graph = Graph(nodes, relationships)
println(graph.edges.count)
```

Finding the connected components

According to the API documentation at http://spark.apache.org/docs/latest/graphx-programming-guide.html, each connected components of the graph are labeled by the connected components algorithm using the lowest-numbered vertex IDs. For example, in a social network analysis, the clusters are approximated by the connected components. To make this even easier and faster, the GraphX API contains an implementation of the algorithm as the `ConnectedComponents` object.

However, there's no Java, Python, or R-based implementation for finding the connected components. Therefore, it allows us to compute the connected components in the topics we have calculated using the LDA algorithm through one or more term, as follows:

```
val facts: RDD[String] =
      graph.triplets.map(triplet =>
        triplet.srcAttr._1 + " contains the terms
"" + triplet.attr + "" like as " +
triplet.dstAttr._1)
    facts.collect.foreach(println(_))
```

This should produce the output as shown in *Figure 13*:

```
Topic_1 contains the terms "just" like as Topic_2
Topic_1 contains the terms "zara" like as Topic_3
Topic_0 contains the terms "know" like as Topic_1
Topic_0 contains the terms "like" like as Topic_2
Topic_0 contains the terms "rihanna" like as Topic_2
Topic_0 contains the terms "fucking" like as Topic_2
Topic_1 contains the terms "beyonce" like as Topic_3
```

Figure 13: Connection between the topics using GraphX.

If you look the output in *Figure 13* carefully, we have printed the relationships with a triplet. It is to be noted that in addition to the support of the vertices and edges, Spark GraphX also has the notion of a triplet. More technically, a triplet is an object that broadens the Edge object. From the graph perspective, it stores the information about an edge and the related vertices adjacent to it in a graph.

Summary

In this chapter, we have shown how to develop large-scale machine learning applications from real-time Twitter stream data and graph data. We have discussed the social network and time-series data analysis. In addition, we also developed an emerging recommendation application by using the content-based collaborative filtering algorithms of Spark MLlib to make movie recommendations for users. These applications, however, can be extended and deployed for other use cases.

It is worth noting that the current implementation of Spark contains a few implemented algorithms for the streaming or network data analysis. However, we can hope that, for example, GraphX will be improved in the future and extended for not only Scala, but for Java, R, and Python too. In the next chapter, we will focus on how to interact with external data sources to make the Spark working environment more diverse.

Chapter 10. Configuring and Working with External Libraries

This chapter guides you on using external libraries to expand your data analysis to make the Spark more versatile. Examples will be given for deploying third-party-developed packages or libraries for machine learning applications with Spark core and ML/MLlib. We will also discuss how to compile and use external libraries with the core libraries of Spark for time series. As promised, we will also discuss how to configure SparkR to increase exploratory data manipulation and operations. In a nutshell, the following topics will be covered throughout this chapter:

- Third-party ML libraries with Spark
- Using external libraries when deploying Spark ML on a cluster
- Time series analysis using the Spark-TS package of Cloudera
- Configuring SparkR with RStudio
- Configuring Hadoop run-time on Windows

In order to provide a user-friendly environment for the developer, it is also possible to incorporate third-party APIs and libraries with Spark Core and other APIs such as Spark MLlib/ML, Spark Streaming, GraphX, and so on. Interested readers should refer to the following website that is listed on the Spark website as the **Third-Party Packages**: https://spark-packages.org/.

This website is a community index of third-party packages for Apache Spark. To date, there are total 252 packages registered on this site, as shown in *Table 1*:

Domain	**No. of packages**	**URL**
Spark core	9	https://spark-packages.org/?q=tags%3A%22Core%22
Data sources	39	https://spark-packages.org/?q=tags%3A%22Data%20Sources%22
Machine learning	55	https://spark-packages.org/?q=tags%3A%22Machine%20Learning%22

Streaming	36	https://spark-packages.org/?q=tags%3A%22Streaming%22
Graph processing	13	https://spark-packages.org/?q=tags%3A%22Graph%22
Spark with Python	5	https://spark-packages.org/?q=tags%3A%22PySpark%22
Cluster deployment	10	https://spark-packages.org/?q=tags%3A%22Deployment%22
Data processing example	18	https://spark-packages.org/?q=tags%3A%22Examples%22
Applications	10	https://spark-packages.org/?q=tags%3A%22Applications%22
Tools	24	https://spark-packages.org/?q=tags%3A%22Tools%22
Total Packages: 252		

Table 1: Third-party libraries for Spark based on application domain

Third-party ML libraries with Spark

The 55 third-party machine learning libraries include libraries for neural data analysis, generalized clustering, streaming, topic modelling, feature selection, matrix factorization, distributed DataFrame for distributed ML, model matrix, Stanford Core NLP wrapper for Spark, social network analysis, deep learning module running, assembly of fundamental statistics, binary classifier calibration, and tokenizer for DataFrame.

Table 2 provides a summary of the most useful packages based on use cases and application areas of machine learning. Interested readers should visit the respective websites for more insights:

Third party ML library for Spark	**Use cases**
thunder ScalaNetwork	Neural network Large-scale neural data analysis with Spark where the neural network implementation is done with Scala.
generalized-kmeans-clustering patchwork bisecting-kmeans spark-knn	Clustering This project generalizes the Spark MLLIB K-means cluster to support arbitrary distance functions.Highly scalable grid-density clustering algorithm for Spark MLlib. This is a prototype implementation of Bisecting K-Means Clustering on Spark. k-nearest neighbors algorithm on Spark.
spark-ml-streaming streaming-matrix-factorization twitter-stream-ml	Streaming Visualize streaming machine learning in Spark. Streaming Recommendation engine using matrix factorization with user and product bias. Machine learning over Twitter's stream. Using Apache Spark, Web Server and Lightning Graph server.
pipeline	Docker-based pipelining End-to-End, real-time, advanced analytics big data reference pipeline using Spark, Spark SQL, Spark Streaming, ML, MLlib, GraphX, Kafka,

	Cassandra, Redis, Apache Zeppelin, Spark-Notebook, iPython/Jupyter Notebook, Tableau, H2O Flow, and Tachyon.
dllib CaffeOnSpark`dl4j-spark-ml`	Deep learning dllib is a deep learning tool running on Apache Spark. Users need to download the tools as .jar and then can integrate with Spark and develop deep-learning-based applications. CaffeOnSpark is a scalable deep learning running with the Spark executors. It is based on peer-to-peer (P2P) communication. `dl4j-spark-ml` can be used to develop deep-learning-based ML applications by integrating with Spark ML.
kNN_IS sparkboostspark-calibration	Classification kNN-IS: An Iterative Spark-based design of the k-Nearest Neighbours classifier for big data.A distributed implementation of AdaBoost.MH and MP-Boost using Apache Spark. Assesses binary classifier calibration (that is, how well classifier outputs match observed class proportions) in Spark.
Zen	Regression Zen provides a platform for large-scale and efficient machine learning on top of Spark. For example, logistic regression, linear regression, Latent Dirichlet Allocation (LDA), factorization machines and Deep Neural Network (DNN) are implemented in the current release.
modelmatrix spark-infotheoretic-feature-selection	Feature engineering spark-infotheoretic-feature-selection tools provide an alternative to Spark for developing large-scale machine learning applications. They provides robust feature engineering through the pipelining including

	the feature extractors, feature selectors. It is focused on building sparse feature-vector-based pipelines. On the other hand, it can be used as a feature selection framework based on Information Theory. Algorithms based on Information Theory include mRMR, InfoGain, JMI, and other commonly used FS filters.
spark-knn-graphs	Graph processing Spark algorithms for building and processing k-nn graphs
TopicModeling	Topic modelling Distributed Topic Modelling on Apache Spark
Spark.statistics	Statistics Apart from SparkR, Spark.statistics works as an assembler of basic statistics implementation based on the Spark core

Table 2: Summary of the most useful third-party packages based on use cases and application areas of machine learning with Spark

Using external libraries with Spark Core

In order to work with these external libraries, instead of placing the jars in any specific folder, a simple fix would be to start the `pyspark` shell or spark-shell with the following arguments:

```
bin/pyspark --packages
com.databricks:spark-
csv_2.10:1.0.3
bin/spark-
shell --packages
com.databricks:spark-csv_2.10:1.0.3
```

This will automatically load the required `spark-csv` jars. However, these two jar files have to be downloaded to the Spark distribution using the following command in Ubuntu:

```
wget
http://search.maven.org/remotecontent?
filepath=org/apache/commons/commons-
csv/1.1/commons-csv-
1.1.jar
wget
http://search.maven.org/remotecontent?
filepath=com/databricks/spark-
csv_2.10/1.0.0/spark-csv_2.10-1.0.0.jar
```

Then, to create an active Spark session, use the following line of codes:

```
static SparkSession spark = SparkSession
        .builder()
        .appName("JavaLDAExample")
          .master("local[*]")
          .config("spark.sql.warehouse.dir",
"C:/Exp/")
          .getOrCreate();
```

Once you have instantiated an active Spark session, use the following lines of code to read the csv input file:

```
String input = "input/letterdata.data";
Dataset<Row> df =
spark.read().format("com.databricks.spark.csv").option("header",
"true").load(input);
```

```
df.show();
```

Note that here we define the `com.databricks.spark.csv` input format by using the `format()` method, dedicatedly developed by Databricks for faster CSV file reading and parsing, and by setting the auxiliary option for the header as true using the `option()` method. Finally, the `load()` method loads the input data from the `input/letterdata.data` location, for example.

As a continuation, in the next section, we will discuss configuring the Spark-TS library for time series data analysis.

Tip

Interested readers should visit the third-party ML packages web page for Spark at https://spark-packages.org/?q=tags%3A%22Machine%20Learning%22 for the package-specific discussion, updates, and configuration procedures.

Time series analysis using the Cloudera Spark-TS package

As discussed in *Chapter 9*, *Advanced Machine Learning with Streaming and Graph Data*, we will see how to configure the Spark-TS package developed by Cloudera. Mainly, we will talk about the TimeSeriesRDD in this section.

Time series data

Time series data consists of sequences of measurements, each occurring at a point in time. A variety of terms are used to describe time series data, and many of them apply to conflicting or overlapping concepts. In the interest of clarity, in Spark-TS, Cloudera sticks to a particular vocabulary. Three objects are important in time series data analysis: time series, instant, and observation:

- A time series is a sequence of real (that is, floating-point) values, each linked to a specific timestamp. Particularly, this sticks with time series as meaning a univariate time series. In Scala, a time series is usually represented by a Breeze presented at https://github.com/scalanlp/breeze vector, and in Python, a 1-D NumPy array (refer to http://www.numpy.org/ for more), and has a `DateTimeIndex` as shown at https://github.com/sryza/spark-timeseries/blob/master/src/main/scala/com/cloudera/sparkts/DateTimeIndex.scala.
- On the other hand, an instant is the vector of values in a collection of time series corresponding to a single point in time. In the Spark-TS library, each time series is typically labeled with a key that enables it to be identified among a collection of time series.
- Finally, an observation is a tuple of (timestamp, key, value), that is, a single value in a time series or instant.

However, not all data with timestamps are time series data. For example, logs don't fit directly into time series since they consist of discrete events, not scalar measurements taken at intervals. However, measurements of log messages per hour would constitute a time series.

Configuring Spark-TS

The most straightforward way to access Spark-TS from Scala is to depend on it in a Maven project. Do this by including the following repo in `pom.xml`:

```
<repositories>
    <repository>
      <id>cloudera-repos</id>
```

```
      <name>Cloudera Repos</name>

 <url>https://repository.cloudera.com/artifactory/cloudera-
repos/</url>
    </repository>
</repositories>
 And including the following dependency in the
pom.xml:
<dependency>
      <groupId>com.cloudera.sparkts</groupId>
      <artifactId>sparkts</artifactId>
      <version>0.1.0</version>
</dependency>
```

To get the raw `pom.xml` file, interested readers should go to the following URL:

https://github.com/sryza/spark-timeseries/blob/master/pom.xml

Alternatively, to access it in a spark-shell, download the JAR from https://repository.cloudera.com/cloudera/libs-release-local/com/cloudera/sparkts/sparkts/0.1.0/sparkts-0.1.0-jar-with-dependencies.jar, and then launch the shell with the following command as discussed in the *Using external libraries with Spark Core* section in this chapter:

```
spark-shell \
      --jars
sparkts-0.1.0-jar-with-dependencies.jar \
      --driver-class-path
sparkts-0.1.0-jar-with-dependencies.jar
```

TimeSeriesRDD

According to the Spark-TS engineering blog written on the Cloudera website at http://blog.cloudera.com/blog/2015/12/spark-ts-a-new-library-for-analyzing-time-series-data-with-apache-spark/, TimeSeriesRDD is central to Spark-TS, where each object in the RDD stores a full univariate series. Operations that tend to apply exclusively to time series are much more efficient. For example, if you want to generate a set of lagged time series from your original collection of time series, each lagged series can be computed just by looking at a single record in the input RDD.

Similarly, with imputing missing values based on surrounding values, or fitting time series models to each series, all of the data needed is present in a single array. Therefore, the central abstraction of the Spark-TS library is TimeSeriesRDD, which is simply a collection of time series on which you can operate in a distributed fashion. This approach allows you to avoid storing timestamps for each series and instead store a single

`DateTimeIndex` to which all the series vectors conform. `TimeSeriesRDD[K]` extends `RDD[(K, Vector[Double])]`, where K is the key type (usually a string), and the second element in the tuple is a Breeze vector representing the time series.

A more technical discussion can be found in the GitHub URL: https://github.com/sryza/spark-timeseries. Since this is a Third Party Package, a detailed discussion is out of the scope of this book, we believe.

Configuring SparkR with RStudio

Let's assume you have RStudio installed on your machine. Follow the steps mentioned here:

1. Now open RStudio and create a new R script; then write the following code:

```
      SPARK_HOME =
"/home/spark-2.0.0-bin-hadoop2.7/R/lib"
      Sys.setenv(SPARK_MEM="8g")
      Sys.setenv(SPARK_HOME =
"/home/spark-2.0.0-bin-hadoop2.7")

 .libPaths(c(file.path(Sys.getenv("SPARK_HOME"),
"R",
      "lib"),.libPaths()))
```

2. Load the necessary package for SparkR by using this code:

```
      library(SparkR, lib.loc = SPARK_HOME)
      ibrary(SparkR)
```

3. Configure the SparkR environment as follows:

```
      sc <- sparkR.init(appName =
"SparkR-DataFrame-example", master =
      "local")
      sqlContext <- sparkRSQL.init(sc)
```

4. Now let's create the first DataFrame and print the first few rows, as follows:

```
      df <- createDataFrame(sqlContext,
faithful)
      head(df)
```

5. You might need to install the following packages in order to make the `devtools` package work:

```
      install.packages("xml2", dependencies =
TRUE)
      install.packages("Rcpp", dependencies =
TRUE)
      install.packages("plyr", dependencies =
TRUE)
      install.packages("devtools",
dependencies = TRUE)
      install.packages("MatrixModels",
dependencies = TRUE)
      install.packages("quantreg",
dependencies = TRUE)
```

```
      install.packages("moments",
dependencies = TRUE)
      install.packages("xml2")
      install.packages(c("digest", "gtable",
"scales", "rversions",
      "lintr"))
```

6. Morever, you might need to install `libcurl` for RCurl, which devtools depends on. To do this, just run this command:

```
      sudo apt-get -y build-dep
libcurl4-gnutls-dev
      sudo apt-get install
libcurl4-gnutls-dev
      sudo apt-get install r-cran-plyr
      sudo apt-get install r-cran-reshape2
```

7. Now configure the `ggplot2.SparkR` package from GitHub using the following code:

```
      library(devtools)

 devtools::install_github("SKKU-
SKT/ggplot2.SparkR")
```

8. Now let's compute the skewness and kurtosis for the sample DataFrame that we have just created. Before that, load the necessary packages:

```
      library(moments)
      library(ggplot2)
```

9. Let's create the DataFrame for the daily exercise example shown in the *Feature engineering and data exploration* section in *Chapter 4*, *Extracting Knowledge through Feature Engineering*, and show the first few rows using `head` command:

```
      time_taken <- c (15, 16, 18, 17.16,
16.5, 18.6, 19.0, 20.4, 20.6,
      25.15, 27.27, 25.24, 21.05, 21.65,
20.92, 22.61, 23.71, 35, 39, 50)
      df_new <- data.frame(time_taken)
      head(df_new)
      df<- createDataFrame(sqlContext, data =
df_new)
      head(df)
```

10. Now calculate the skewness and kurtosis, as follows:

```
      skewness(df)
      kurtosis(df_new)
```

You are probably aware that we used the two terms `skewness` and `kurtosis` in Chapter 4, *Extracting Knowledge through Feature Engineering*. If you are not familiar with these two terms, here is a bit of definition of them. Well, from the statistical perspective, `skewness` is a measure of symmetry. Alternatively and more precisely, it signifies the lack of symmetry in a distribution of the dataset.

Now you might be wondering what symmetric is. Well, a distribution of the dataset is symmetric if it looks the same to the left and right of the center point.

Kurtosis, on the other hand, is a measure of whether the data are heavy-tailed or light-tailed relative to a normal distribution:

11. Finally, let's plot the density plot graph by calling the `ggplot()` method of the `ggplot2.SparkR` package:

```
      ggplot(df, aes(x = time_taken)) +
stat_density(geom="line",
      col= "green", size = 1, bw = 4) +
theme_bw()
```

If you are not familiar with the `ggplot2` R package, note that `ggplot2` is a plotting system for R based on the grammar of graphics of base and lattice graphics. It provides many fiddly details of the graphics that make plotting a hassle, for example, placing or drawing legends in a graph, as well as providing a powerful model of graphics. This will make your life easier in order to produce simple as well as complex multi-layered graphics.

Tip

More info about `ggplot2` and its documentation can be found at the following website: http://docs.ggplot2.org/current/.

Configuring Hadoop run-time on Windows

If you are developing your machine learning application on windows using Eclipse (as Maven project of course), probably you will face a problem since Spark expects that there is a runtime environment for Hadoop on Windows too.

More specifically, suppose you are running a Spark project written in Java with main class as `JavaNaiveBayes_ML.java`, then you will experience an IO exception saying that:

```
16/10/04 11:59:52 ERROR Shell: Failed to locate
the winutils binary in the hadoop binary path
java.io.IOException: Could not locate executable
null\bin\winutils.exe in the Hadoop binaries.
```



Figure 1: IO exception due to the missing Hadoop runtime

The reason is that by default Hadoop is developed for the Linux environment and if you are developing your Spark applications on windows platform, a bridge is required that will provide the Hadoop environment for the Hadoop runtime for Spark to be properly executed.

Now, how to get rid of this problem then? The solution is straight forward. As the error message says, we need to have an executable namely `winutils.exe`. Now download the `winutils.exe` file from the code directory of Packt for this chapter and copy and paste it in the Spark distribution directory and configure Eclipse.

More specifically, suppose your Spark distribution containing Hadoop is located at `C:/Users/spark-2.0.0-bin-hadoop2.7`. Inside the Spark

distribution there is a directory named `bin.` Now, paste the executable there (that is, `path = C:/Users/spark-2.0.0-binhadoop2.7/bin/`).

The second phase of the solution is going to Eclipse, select the main class (that is, `JavaNaiveBayes_ML.java` in this case), and then go to the **Run** menu. From the **Run** menu go to the **Run Configurations** option and from this option select the **Environment** tab. If you select the tab, you a will have the option to create a new environmental variable for Eclipse suing the JVM.

Now create a new environmental variable and put the value as `C:/Users/spark-2.0.0-bin-hadoop2.7/`. Now press on **apply** and re-run your application and your problem should be resolved.

More technically, the details of the IO exception can be described as follows in Figure 1:

Summary

In this chapter, we showed how to use external libraries with Spark to expand data analyses.

More and more Spark as well as third-party packages are being developed by open source contributors. Readers should be updated with the latest news and release on the Spark website. They also should be notified of about the latest machine learning APIs, since the development of Spark is continuous and innovative and, of course, sometimes after a certain package becomes obsolete or deprecated.

Throughout this book, we have tried to guide you on how to use the most popular and widely used machine learning algorithms that have been developed by Spark. However, there are other algorithms too that we could not discuss, and more and more algorithms will be added to the Spark ML and MLlib packages.

This is more or less the end of our little journey with Spark. Now a general suggestion from our side to you as readers, or if you are relatively new to machine learning, Java, or Spark at first try to understand whether a problem is really a machine learning problem. If it is a machine learning problem, try to guess what type of learning algorithms should be the best fit, that is, classification, clustering, regression, recommendation, or frequent pattern mining.

Then define and formulate the problem. After that you should generate or download the appropriate data based on the feature engineering concept of Spark that we have discussed. Then you can select an ML model that will provide better results in terms of accuracy. However, as discussed earlier, the model selection really depends on your data and problem type.

Now that you have your data ready to train the model, go straight to train the model towards making predictive analytics.

When your model is trained, evaluate it to see how it goes and fulfills your prediction expectations. Well, if you are not happy with the performance, try changing to other ML algorithms towards model selection. As discussed in Chapter 7, *Tuning Machine Learning Models*, even proper model selection cannot provide the best result sometimes because of the nature of the data you have.

So what is to be done? It's simple. Tune your ML model using the available tuning algorithms to properly set the hyperparameters. You might also need to make your model adaptable for new data types, especially if you are

developing an ML application for a dynamic environment such as time series analysis or streaming analytics.

Finally, deploy your model and you have it as a robust ML application.

Our final recommendation to the readers is to browse the Spark website (at http://spark.apache.org/) regularly to get updates and also try to incorporate the regular Spark provided APIs with other third-party applications to get the best result of the collaboration.