Md. Rezaul Karim

Predictive Analytics with TensorFlow

Implement deep learning principles to predict valuable insights using TensorFlow



Packt>

Predictive Analytics with TensorFlow

Implement deep learning principles to predict valuable insights using TensorFlow

Md. Rezaul Karim



BIRMINGHAM - MUMBAI

Predictive Analytics with TensorFlow

Copyright © 2017 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 author, 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 2017

Production reference: 1251017

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78839-892-3

www.packtpub.com

Credits

Author
Md. Rezaul Karim

Reviewers
Andrea Mostosi
Meng-Chieh Ling

Commissioning Editor
Sunith Shetty

Acquisition Editor
Chandan Kumar

Content Development Editor
Amrita Noronha

Technical Editor
Sayali Thanekar

Copy Editor
Safis Editing

Project Coordinator
Shweta H Birwatkar

Proofreader
Safis Editing

Indexer
Pratik Shirodkar

Graphics
Tania Dutta

Production Coordinator
Aparna Bhagat

About the Author

Md. Rezaul Karim is a Research Scientist at Fraunhofer FIT, Germany. He is also a PhD candidate at RWTH Aachen University, Aachen, Germany. He holds a BSc and an MSc degree in Computer Science. Before joining Fraunhofer FIT, he worked as a Researcher at Insight Centre for Data Analytics, Ireland. Before this, he worked as a Lead Engineer at Samsung Electronics' distributed R&D Institutes in Korea, India, Turkey, and Bangladesh. Previously, he has worked as a Research Assistant at the database lab, Kyung Hee University, Korea. He also worked as an R&D engineer with BMTech21 Worldwide, Korea. Even before this, he worked as a Software Engineer with i2SoftTechnology, Dhaka, Bangladesh.

He has more than 8 years of experience in the area of research and development with solid understanding of algorithms and data structures in C, C++, Java, Scala, R, and Python. He has published several books, articles, and research papers concerning big data and virtualization technologies, such as Spark, Kafka, DC/OS, Docker, Mesos, Zeppelin, Hadoop, and MapReduce. He is also equally competent with deep learning technologies such as TensorFlow, DeepLearning4j, and H2O. His research interests include Machine Learning, Deep Learning, Semantic Web, Linked Data, Big Data, and Bioinformatics. Also, he is the author of the following book titles:

- *Large-Scale Machine Learning with Spark* (Packt Publishing Ltd.)
- *Deep Learning with TensorFlow* (Packt Publishing Ltd.)
- *Scala and Spark for Big Data Analytics* (Packt Publishing Ltd.)

Acknowledgments

I am very grateful to my parents, who have always encouraged me to pursue knowledge. I also want to thank my wife, Saroar; son, Shadman; brother, Mamtaz; sister, Josna; and friends who have endured my long monologs about the subjects in this book and always have encouraged and listened to me. Writing this book was made easier by the amazing efforts of the open source community and the great documentation of many projects out there related to TensorFlow and Python. Further, I would like to thank the acquisition, content development, and technical editors of Packt Publishing Ltd. (and, of course, others who were involved in this book title) for their sincere cooperation and coordination. Additionally, without the work of numerous researchers and deep learning practitioners who shared their expertise in publications, lectures, and source code, this book might not have existed at all! Finally, I appreciate the efforts of the TensorFlow community and all those who have contributed to APIs, whose work ultimately brought machine learning to the masses.

About the Reviewers

Andrea Mostosi is a technology enthusiast, a husband, and a father. During the last 10 years, he led the entire life cycle of several projects across different technologies, companies, and markets. He is now working on artificial intelligence, data mining, and a lot of other scary things.

I'd like to thank my wonderful son, Ryan, for every smile, every hug, and every sleepless night he gave me since his birth. When the machines finally take over humanity, you'll be able to say that your father has contributed to making this happen, my son.

Meng-Chieh Ling is a theoretical physics PhD from Karlsruhe Institute of Technology. After finishing his PhD, he attended The Data Incubator Reply to change his career from theoretical physics to data science.

www.PacktPub.com

eBooks, discount offers, and more

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 `customercare@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

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at `https://www.amazon.com/dp/1788398920`.

If you'd like to join our team of regular reviewers, you can email us at `customerreviews@packtpub.com`. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface **ix**

Chapter 1: Basic Python and Linear Algebra for Predictive Analytics **1**

A basic introduction to predictive analytics **2**
Why predictive analytics? 3
Working principles of a predictive model 3
A bit of linear algebra **7**
Programming linear algebra 8
Installing and getting started with Python **9**
Installing on Windows 9
Installing Python on Linux 11
Installing and upgrading PIP (or PIP3) 12
Installing Python on Mac OS 12
Installing packages in Python 13
Getting started with Python **13**
Python data types 14
Using strings in Python 14
Using lists in Python 15
Using tuples in Python 16
Using dictionary in Python 17
Using sets in Python 18
Functions in Python 19
Classes in Python 20
Vectors, matrices, and graphs **21**
Vectors 21
Matrices 25
Matrix addition 25
Matrix subtraction 25
Finding the determinant of a matrix 27

Finding the transpose of a matrix 27
Solving simultaneous linear equations 28
Eigenvalues and eigenvectors 29
Span and linear independence 29
Principal component analysis 31
Singular value decomposition 33
Data compression in a predictive model using SVD 34
Predictive analytics tools in Python 37
Summary 38
Chapter 2: Statistics, Probability, and Information Theory for Predictive Modeling 39
Using statistics in predictive modeling 40
Statistical models 41
Parametric versus nonparametric model 41
Population and sample 42
Random sampling 43
Expectation 43
Central limit theorem 44
Skewness and data distribution 46
Standard deviation and variance 47
Covariance and correlation 49
Interquartile, range, and quartiles 50
Hypothesis testing 51
Chi-square tests 52
Chi-square independence test 53
Basic probability for predictive modeling 54
Probability and the random variables 54
Generating random numbers and setting the seed 55
Probability distributions 55
Marginal probability 56
Conditional probability 56
The chain rule of conditional probability 57
Independence and conditional independence 58
Bayes' rule 58
Using information theory in predictive modeling 60
Self-information 60
Mutual information 61
Entropy 61
Shannon entropy 61
Joint entropy 62
Conditional entropy 62
Information gain 62
Using information theory 63

Using information theory in Python 64
Summary 67
Chapter 3: From Data to Decisions – Getting Started with TensorFlow 69
Taking decisions based on data - Titanic example 70
Data value chain for making decisions 70
From disaster to decision – Titanic survival example 71
General overview of TensorFlow 76
Installing and configuring TensorFlow 78
Installing TensorFlow on Linux 79
Installing Python and nVidia driver 80
Installing TensorFlow from source 85
Testing your TensorFlow installation 87
TensorFlow computational graph 87
TensorFlow programming model 90
Data model in TensorFlow 93
Tensors 94
Rank 96
Shape 97
Data type 97
Variables 100
Fetches 101
Feeds and placeholders 102
TensorBoard 103
How does TensorBoard work? 104
Getting started with TensorFlow – linear regression and beyond 105
Source code for the linear regression 111
Summary 113
Chapter 4: Putting Data in Place - Supervised Learning for Predictive Analytics 115
Supervised learning for predictive analytics 116
Linear regression - revisited 117
Problem statement 118
Using linear regression for movie rating prediction 119
From disaster to decision - Titanic example revisited 131
An exploratory analysis of the Titanic dataset 132
Feature engineering 137
Logistic regression for survival prediction 140
Using TensorFlow contrib 141
Linear SVM for survival prediction 146

Ensemble method for survival prediction: random forest 151
A comparative analysis 155
Summary 156
Chapter 5: Clustering Your Data - Unsupervised Learning for Predictive Analytics 157
Unsupervised learning and clustering 157
Using K-means for predictive analytics 160
How K-means works 160
Using K-means for predicting neighborhoods 162
Predictive models for clustering audio files 170
Using kNN for predictive analytics 182
Working principles of kNN 182
Implementing a kNN-based predictive model 183
Summary 192
Chapter 6: Predictive Analytics Pipelines for NLP 193
NLP analytics pipelines 194
Using text analytics 195
Transformers and estimators 196
Standard transformer 196
Estimator transformer 197
StopWordsRemover 199
N-gram 200
Using BOW for predictive analytics 201
Bag-of-words 201
The problem definition 202
The dataset description and exploration 202
Spam prediction using LR and BOW with TensorFlow 203
TF-IDF model for predictive analytics 214
How to compute TF, IDF, and TFIDF? 215
Implementing a TF-IDF model for spam prediction 218
Using Word2vec for sentiment analysis 225
Continuous bag-of-words 225
Continuous skip-gram 226
Using CBOW for word embedding and model building 227
CBOW model building 227
Reusing the CBOW for predicting sentiment 235
Summary 241
Chapter 7: Using Deep Neural Networks for Predictive Analytics 243
Deep learning for better predictive analytics 244
Artificial Neural Networks 245

Deep Neural Networks **248**
DNN architectures 248
Multilayer perceptrons **250**
Training an MLP 251
Using MLPs 252
DNN performance analysis **253**
Fine-tuning DNN hyperparameters **257**
Number of hidden layers 257
Number of neurons per hidden layer 258
Activation functions 258
Weight and biases initialization 258
Regularization 259
Using multilayer perceptrons for predictive analytics **260**
Dataset description 261
Preprocessing 263
A TensorFlow implementation of MLP 265
Deep belief networks **273**
Restricted Boltzmann Machines 274
Construction of a simple DBN 277
Unsupervised Pretraining 277
Using deep belief networks for predictive analytics **279**
Summary **286**

Chapter 8: Using Convolutional Neural Networks for Predictive Analytics **287**

CNNs and the drawbacks of regular DNNs **288**
CNN architecture **289**
Convolutional operations **290**
Applying convolution operations in TensorFlow 291
Pooling layer and padding operations **293**
Applying subsampling operations in TensorFlow 294
Tuning CNN hyperparameters **296**
CNN-based predictive model for sentiment analysis **300**
Exploring movie and product review datasets 301
Using CNN for predictive analytics about movie reviews 302
CNN model for emotion recognition **317**
Dataset description 317
CNN architecture design 318
Testing the model on your own image 328
Using complex CNN for predictive analytics 333
Dataset description 333

CNN predictive model for image classification **334**
Summary **356**
Chapter 9: Using Recurrent Neural Networks for Predictive Analytics **357**
RNN architecture **358**
Contextual information and the architecture of RNNs 358
BRNNs 359
LSTM networks 360
GRU cell 363
Using BRNN for image classification **364**
Implementing an RNN for spam prediction **371**
Developing a predictive model for time series data **378**
Description of the dataset 378
Preprocessing and exploratory analysis 379
LSTM predictive model 382
Model evaluation 384
An LSTM predictive model for sentiment analysis **388**
Network design 388
LSTM model training 389
Visualizing through TensorBoard 406
LSTM model evaluation 408
Summary **410**
Chapter 10: Recommendation Systems for Predictive Analytics **411**
Recommendation systems **411**
Collaborative filtering approaches 412
Content-based filtering approaches 413
Hybrid recommendation systems 413
Model-based collaborative filtering 414
Collaborative filtering approach for movie recommendations **414**
The utility matrix 414
Dataset description 416
Ratings data 416
Movies data 416
User data 417
Exploratory analysis of the dataset 418
Implementing a movie recommendation engine 424
Training the model with available ratings 424
Inferencing the saved model 433
Generating a user-item table 434
Clustering similar movies 435
Movie rating prediction by users 439
Finding the top K movies 440

Predicting top K similar movies 441
Computing the user-user similarity 442
Evaluating the recommendation system 442
Factorization machines for recommendation systems 445
Factorization machines 446
The cold start problem in recommendation systems 447
Problem definition and formulation 448
Dataset description 449
Preprocessing 450
Implementing an FM model 452
Improved factorization machines for predictive analytics 454
Neural factorization machines 455
Dataset description 456
Using NFM for movie recommendations 456
Summary 461
Chapter 11: Using Reinforcement Learning for Predictive Analytics 463
Reinforcement learning 464
Reinforcement learning in predictive analytics 464
Notation, policy, and utility in RL 465
Policy 466
Utility 466
Developing a multiarmed bandit's predictive model 468
Developing a stock price predictive model 477
Summary 487
Index 489

Preface

The continued growth in data, coupled with the need to make increasingly complex decisions against that data, is creating massive hurdles that prevent organizations from deriving insights in a timely manner using traditional approaches. Machine learning is concerned with algorithms that transform raw data into information and then into actionable intelligence. This fact makes machine learning well suited to the predictive analytics. Without machine learning, therefore, it would be nearly impossible to keep up with these massive streams of information altogether.

On the other hand, deep learning is a branch of machine learning algorithms based on learning multiple levels of representation. A deep learning algorithm is nothing more than the implementation of a complex and deep neural network so that it can learn through the analysis of large amounts of data. Thus, it took just a few years to develop powerful deep learning algorithms to recognize images, natural language processing, and perform a myriad of other complex tasks.

Considering these motivations and requirements, this book is dedicated to developers, data analysts, machine learning practitioners, and deep learning enthusiasts who want to build powerful, robust, and accurate predictive models with the power of TensorFlow from scratch, and combining other open source Python libraries.

The first section of this book covers applied math, statistics, and probability theory for predictive analytics. It will then cover useful Python packages to getting started with data science in a practical manner. The second section shows how to develop large-scale predictive analytics pipelines using supervised learning algorithms, for example, classification and regression; and unsupervised learning algorithms, for example, clustering. It'll then demonstrate how to develop predictive models for NLP.

Finally, reinforcement learning and a factorization machine-based recommendation system will be used to develop predictive models. The third section covers practical mastery of deep learning architectures for advanced predictive analytics, including deep neural networks and recurrent neural networks for high-dimensional and sequence data. Finally, it'll show how to develop convolutional neural networks-based predictive models for emotion recognition, image classification, and sentiment analysis.

Happy Reading!

What this book covers

Chapter 1, *Basic Python and Linear Algebra for Predictive Analytics*, discusses the basic concepts in linear algebra for predictive analytics, such as vectors, matrices, tensors, linear dependence, and span. Then, we move on to a brief introduction to **Principal Component Analysis** (**PCA**) and **Singular Value Decomposition** (**SVD**). Finally, some predictive modeling tools in Python will be discussed.

Chapter 2, *Statistics, Probability, and Information Theory for Predictive Modeling*, covers some statistic, probabilistic, and information theory concepts before getting started on predictive analytics: random sampling, hypothesis testing, chi-square test, correlation, expectation, variance, covariance and Bayes' rule, and so on. It then discusses the central objects of probability theory: random variables, stochastic processes, and events. Information theory, which studies the quantification, storage, and communication of information, will be discussed at the end of the chapter.

Chapter 3, *From Data to Decisions - Getting Started with TensorFlow,* provides a detailed description of the main TensorFlow features in a real-life problem, followed by detailed discussions about TensorFlow installation and configuration. It then covers computation graphs, data, and programming models before getting started with TensorFlow. The last part of the chapter contains an example of implementing linear regression model for predictive analytics.

Chapter 4, *Putting Data in Place - Supervised Learning for Predictive Analytics*, covers some TensorFlow-based supervised learning techniques from a theoretical and practical perspective. In particular, the linear regression model for regression analysis will be covered on a real dataset. It then shows how we could solve the Titanic survival problem using logistic regression, random forests, and SVMs for predictive analytics.

Chapter 5, *Clustering Your Data - Unsupervised Learning for Predictive Analytics*, digs deeper into predictive analytics and finds out how we can take advantage of it to cluster records belonging to the certain group or class for a dataset of unsupervised observations. It will then provide some practical examples of unsupervised learning. Particularly, clustering techniques using TensorFlow will be discussed with some hands-on examples.

Chapter 6, *Predictive Analytics Pipelines for NLP*, shows how to use TensorFlow for text analytics with a focus on text classification from an unstructured spam prediction and movie review dataset. Based on the spam filtering dataset, it shows how to develop predictive models using a linear regression algorithm with TensorFlow. Particularly, it will use the **bag-of-words** (**BOW**) and TF-IDF algorithms for spam prediction. Later on, it will also show how to develop large-scale predictive models for predicting sentiment from the movie review dataset using the **continuous bag-of-words** (**CBOW**) and continuous skip-gram algorithms.

Chapter 7, *Using Deep Neural Networks for Predictive Analytics*, demonstrates how to train DNNs and analyze the performance metrics that are needed to evaluate a DNN predictive model. It also shows how to tune the hyperparameters for DNNs for better and optimized performance. It will provide two examples on how to build very robust and accurate predictive models for predictive analytics as well, in particular, using **Deep Belief Networks** (**DBN**) and **Multilayer Perceptron** (**MLP**) on a bank marketing dataset.

Chapter 8, *Using Convolutional Neural Networks for Predictive Analytics*, discusses how to develop predictive analytics applications such as emotion recognition, image classification, and text classification using the convolutional neural network algorithm on real image/text datasets. Finally, it will provide some pointers on how to tune and debug CNN-based networks for optimized performance.

Chapter 9, *Using Recurrent Neural Networks for Predictive Analytics*, provides some theoretical background for RNNs. Then, it shows a few examples of implementing predictive models for image classification, sentiment analysis of movies, and products spam prediction for NLP. Finally, it shows how to develop predictive models for time-series data.

Chapter 10, *Recommendation System for Predictive Analytics*, provides several examples of how to develop recommendation systems for predictive analytics followed by some theoretical background of recommendation systems, for example, matrix factorization. Later in the chapter, an example of developing movie recommendation engine using SVD and K-means will be shown. Finally, the chapter shows how we could use factorization machines to develop a more accurate and robust recommendation system.

Chapter 11, *Using Reinforcement Learning for Predictive Analytics*, talks about designing machine learning systems driven by criticism and rewards. It will show several examples of how to apply reinforcement learning algorithms for developing predictive models on real-life datasets.

What you need for this book

All the examples have been implemented in Python 2 and 3 with TensorFlow 1.2.0+. You will also need some additional software and tools. To be more specific, the following tools and libraries are required, preferably the latest version:

- Python (2.7.x or 3.3+)
- TensorFlow (1.0.0+)
- Bazel (latest version)
- pip/pip3 (latest version for Python 2 and 3 respectively)
- matplotlib (latest version)
- pandas (latest version)
- NumPy (latest version)
- SciPy (latest version)
- sklearn (latest version)
- yahoo_finance (latest version)
- Bazel(latest version)
- CUDA (latest version)
- CuDNN (latest version)

Linux distributions are preferable (including Debian, Ubuntu, Fedora, RHEL, and CentOS) and to be more specific, for Ubuntu it is recommended to have the 14.04 (LTS) 64-bit (or later) complete installation or VMWare player 12 or VirtualBox. You can also run TensorFlow jobs on Windows (XP/7/8/10) or Mac OS X (10.4.7+).

Processor Core i5 or Core i7 with GPU support is recommended to get the best results. However, multicore processing would provide faster data processing and scalability of the predictive analytics jobs—at least 8 GB RAM (recommended) for a standalone mode and at least 32 GB RAM for a single VM and higher for a cluster. There is enough storage for running heavy jobs (depending on the dataset size you will be handling), preferably at least 50 GB of free disk storage.

Who this book is for

This book is dedicated to developers, data analysts, and deep learning enthusiasts who want to build powerful, robust, and accurate predictive models with the power of TensorFlow from scratch and in combination with other open source Python libraries. If you want to build your own extensive applications that work and can predict smart decisions in the future, then this book is what you need! A good command of object-oriented programming with Python is a prerequisite. Some competence in applied mathematics, statistics, linear algebra, and information theory is a plus and would help readers understand the concepts presented 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.

A block of code for importing necessary packages and libraries modules is set as follows:

```
#Import libraries (Numpy, Tensorflow, matplotlib)
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plot
```

When creating the session from the TensorFlow and do some computation, we used the following code segment:

```
with tf.Session() as sess:
      sess.run(tf.global_variables_initializer())
      writer = tf.summary.FileWriter(logs_path, graph=tf.get_default_
graph())
      print("done")
```

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

```
# cp /usr/src/asterisk-addons/configs/cdr_mysql.conf.sample
     /etc/asterisk/cdr_mysql.conf
```

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



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or 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 `www.packtpub.com/authors`.

Customer support

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

Downloading the example code

You can download the example code files for 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**.

You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged in to your Packt account.

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/Predictive-Analytics-with-TensorFlow`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Downloading the color images of this book

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

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you 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.

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

Piracy

Piracy of 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

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

1
Basic Python and Linear Algebra for Predictive Analytics

Predictive analytics (**PA**) is the use of data, statistical algorithms, and machine learning techniques to identify the likelihood of future outcomes based on historical data. The goal is to go beyond knowing what has happened to provide the best assessment of what will happen in the future. However, before we start developing predictive analytics models, knowing basic linear algebra, statistics, probability, and information theory with Python is a mandate. We will start with the basic concepts of linear algebra with Python.

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

- What are predictive analytics and why do we use them?
- What is linear algebra?
- Installing and getting started with Python
- Vectors, matrices, and tensors
- Linear dependence and span
- Principal component analysis (PCA)
- Singular value decomposition (SVD)
- Predictive modeling tools in Python

A basic introduction to predictive analytics

We will refer to a famous definition of machine learning by Tom Mitchell, where he explained what learning really means from a computer science perspective:

> *"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"*

Based on this definition, we can conclude that a computer program or machine can:

- Learn from data and histories
- Can be improved with experience
- Interactively enhance a model that can be used to predict an outcome

Typical machine learning tasks are concept learning, predictive modeling, clustering, and finding useful patterns. The ultimate goal is to improve the learning in such a way that it becomes automatic: so that no human interactions are needed anymore or reduce the level of human interaction as much as possible.

Predictive analytics on the other hand is the process of extracting useful information from historical facts, and stream data (consisting of live data objects) in order to determine hidden patterns and predict future outcomes and trends.



What doesn't predictive analytics do?

Predictive analytics does not tell you what will happen in the future, rather it is about creating predictive models that place a numerical value, or score, on the likelihood of a particular event to happen in the future with an acceptable level of reliability, and includes what-if scenarios and risk assessment.

Why predictive analytics?

In the area of business intelligence, with the right operations management platform, decision-makers are capable of managing all of the business-related inputs, events, and data that provide real-time insight to the enterprise level. Subsequently, predictive models can be used to identify useful patterns from historical, transactional, and recent data to identify potential risks and opportunities. Therefore, it is gaining much attention and wide acceptance. Furthermore, using the traditional reporting and monitoring tools, you have the ability to move from the reactive operations to proactive operations. PA helps move beyond this to plan for the future and identify new areas of business for profit and productivity.

Working principles of a predictive model

Being at the core of predictive analytics, many machine learning functions can be formulated as a convex optimization problem for finding a minimizer of a convex function *f* that depends on a variable vector *w* (weights), which has *d* records. Formally, we can write this as the optimization problem $\min_{\mathbf{w}\in\mathbb{R}^d} f(\mathbf{w})$, where the objective function is of the form:

$$f(\mathbf{w}) := \lambda R(\mathbf{w}) + \frac{1}{n}\sum_{i=1}^{n} L(\mathbf{w}; \mathbf{x}_i, y_i)$$

Here the vectors $\mathbf{x}_i \in \mathbb{R}^d$ are the training data points for $1 \le i \le n$, and $y_i \in \mathbb{R}$ are their corresponding labels that we want to predict eventually. We call the method linear if $L(w;x,y)$ can be expressed as a function of $w^T x$ and y.

The objective function *f* has two components: i) a regularizer that controls the complexity of the model, and ii) the loss that measures the error of the model on the training data. The loss function $L(w;)$ is typically a convex function in w. The fixed regularization parameter $\lambda \ge 0$ defines the trade-off between the two goals of minimizing the loss on the training error and minimizing model complexity to avoid overfitting. For more detailed discussion, interested readers should refer to *Chapter 7*, *Using Deep Neural Networks for Predictive Analytics*.

A more simplified understanding can be gained from figure 1: you have the current data or observations. Now it's your shot to use the black box to predict the future outcome based on the current data and historical facts. In this context, all the undecided values are called **parameters**, and the description–that is, the black box, is a PA model:

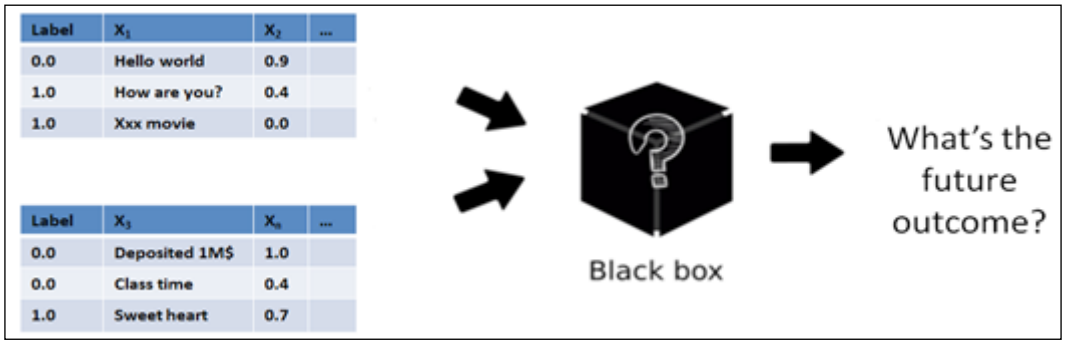


Figure 1: the main task in predictive analytics is predictive modeling–that is, using the black box

As an engineer or a developer, you have to write an algorithm that will observe existing parameters/data/samples/examples to train the black box and figure out how to tune parameters to achieve the best model for making predictions before the deployment. Wow, that's a mouthful! Don't worry; this concept will be clearer in upcoming chapters.

In machine learning, we observe an algorithm's performance in two stages: learning and inference. The ultimate target of the learning stage is to prepare and describe the available data, also called feature vector, which is used to train the model.

The learning stage is one of the most important stages, but it is also truly time-consuming. It involves preparing a list of vectors also called feature vectors (most of the time) from the training data after transformation so that we can feed them to the learning algorithms. On the other hand, training data also sometimes contains impure information that needs some pre-processing such as cleaning.

Once we have the feature vectors, the next step in this stage is preparing (or writing/reusing) the learning algorithm. The next important step is training the algorithm to prepare the predictive model. Typically, (and of course based on data size), running an algorithm may take hours (or even days) so that the features converge into a useful model as shown in the following figure:

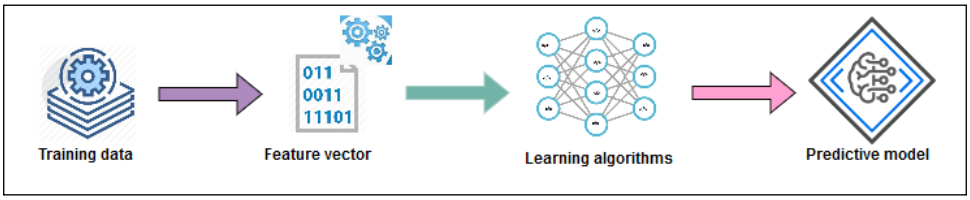


Figure 2: Learning and training a predictive model – it shows how to generate the feature vectors from the training data to train the learning algorithm that produces a predictive model



Common predictive analytics methods

Common predictive analytics methods include regression analysis, classification, time series forecasting, association rule mining, clustering, recommendation systems and text mining, sentiment analysis, and much more. Now to prepare the feature vectors, we need to know a little bit about mathematics, statistics, and so on.

The second most important stage is the inference that is used for making an intelligent use of the model such as predicting from the never-before-seen data, making recommendations, deducing future rules, and so on. Typically, it takes less time compared to the learning stage and sometimes even in real time, as shown in the following figure:

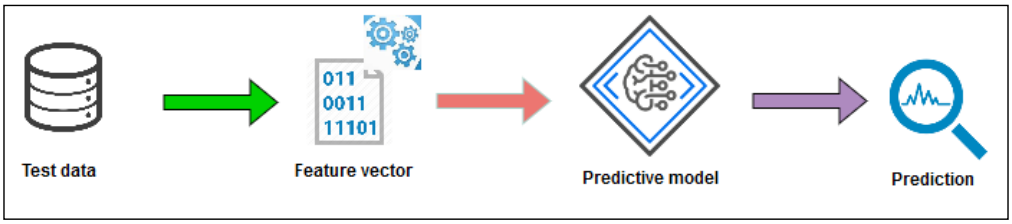


Figure 3: Inferencing from an existing model towards predictive analytics (feature vectors are generated from unknown data for making predictions)

Thus, inferencing (see figure 4 for more) is all about testing the model against new (that is, unobserved) data and evaluating the performance of the model itself. However, in the whole process and for making the predictive model a successful one, data acts as the first-class citizen in all machine learning tasks.

In reality, the data that we feed to our machine learning systems must be mathematical objects, such as vectors, matrices, or graphs (in later chapters, we will refer to them as tensors to make it clearer) so that they can consume such data:

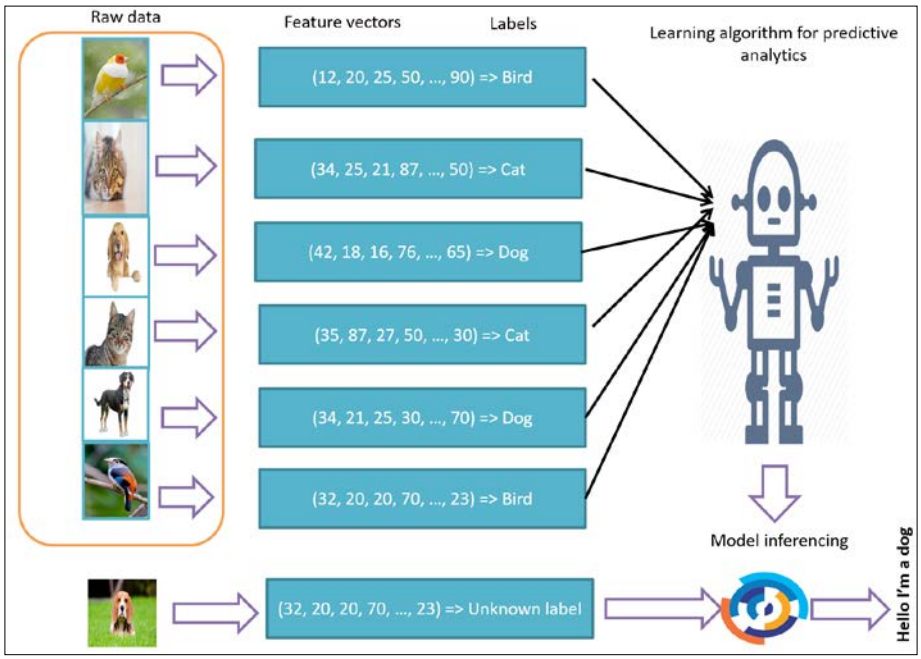


Figure 4: Feature vectors are everywhere - they are used in both learning and inferencing stages in predictive analytics

Depending on the available data and feature types, the performance of your predictive model can vacillate dramatically. Therefore, selecting the right features is one of the most important steps before the inferencing takes place. This is called feature engineering, which can be defined as follows:



Feature engineering

In this process, domain knowledge about the data is used to create only selective or useful features that help prepare the feature vectors to be used so that a machine learning algorithm works.

For example, buying a car; you often see features such as model name, color, horsepower, price, and a number of seats. Thus considering these features, buying a car is not a trivial problem. The general machine learning rule of thumb is that the more data there is, the better the predictive model. However, having more features often creates a mess so the performance degrades drastically: especially if the dataset is high-dimensional and this phenomenon is called the **curse of dimensionality**. We will see some examples in following sections.

In addition, we also need to know how to represent and use such objects through better representation and transformation. These include some basic (and sometimes advanced maths), statistics, probability, and information theory.

For now, this is enough learning. Let's focus on learning some non-trivial topics of linear algebra that could cover vectors, matrix, graphs, and so on. In *Chapter 2*, *Statistics, Probability and Information Theory for Predictive Modeling*, we will learn the basic statistics, probability, and information theory needed for developing PA models. These will be your helping hand as well as basic building blocks for the TensorFlow-based PA throughout subsequent chapters.

A bit of linear algebra

Linear algebra is a branch of pure mathematics that deals with linear sets of equations and their transformation properties such as the analysis of rotations in space, **least squares fitting** (**LSF**), solving linear and differential equations, matrix operation, determination of a circle passing through given points in a vector space over a field, and so on.

You might have heard about the linear regression, which is an example of solving a linear equation where data is represented in the form of linear equations: $y = Ax$. However, in contrast to classical algebra, linear algebra often deals with matrices and vectors. In practice, more complex operations are used in data representation and model building–that is, in a learning algorithm using the notation and formalisms from linear algebra.

Programming linear algebra

As a developer, data scientist, or engineer, you may wish to clutch a programming environment and start coding up vectors, matrix multiplication, PCA, SVD, and QR decompositions with test data. The following are some widely used options that you might like to consider and explore:

- **SciPy**:A Python-based ecosystem for open-source software for mathematics, science, and engineering. This is very easy and is lots of fun if you are a Python programmer with clean syntax.
- **Linear algebra package** (**LAPACK**): is a successor to LINPACK and a standard software library for **numerical linear algebra** (**NLA**). It offers numerous routines for solving systems such as linear equations, linear least squares, eigenvalue, eigenvector, singular value decomposition, matrix factorizations, and Schur decomposition.
- **Basic linear algebra subprograms** (**BLAS**): offers numerous routines as the standard building blocks for performing basic vector and matrix operations.
- **NumPy**: is the fundamental package for scientific computing in Python. It has a very powerful N-dimensional array object for a multi-dimensional container of generic data and numerical operation and broadcasting functions.
- **Pandas**: next to SciPy, BLAS, LAPACK, and NumPy, pandas is one of the most widely used Python libraries for data science. It has some expressive data structures straight away!

Well, enough has been said about linear algebra. Now it's time to discuss how to prepare our development environment for learning LA before getting started with Python and TensorFlow for the predictive analytics in upcoming chapters. From my personal experience, Python is a good candidate for learning and implementing LA. Thus, let's have a quick look at how to install and configure Python on different platforms.

Installing and getting started with Python

Python is one of the most popular programming languages. It is a high-level, interpreted, interactive, and object-oriented scripting language. Unfortunately, there has been a big split between Python versions: 2 versus 3, which could make things a bit confusing to newcomers. You can see the major difference between them at `https://wiki.python.org/moin/Python2orPython3`. But don't worry; I will lead you in the right direction for installing both major versions.

Installing on Windows

On the Python download page at `https://www.python.org/downloads/`, you'll find the latest release of Python 2 or Python 3 (2.7.13 and 3.6.1, respectively, at the time of writing). You can now select and download the installer (`.exe`) of either version. Installation is similar to installing other software on Windows.

Let's assume that you have installed both versions and now it's time to add the installation path to the environmental variables.

For doing so click on **Start**, and then type `advanced system settings`, then select the **View advanced system settings** | **System Properties** | **Advanced** | **Environment Variables...** button:

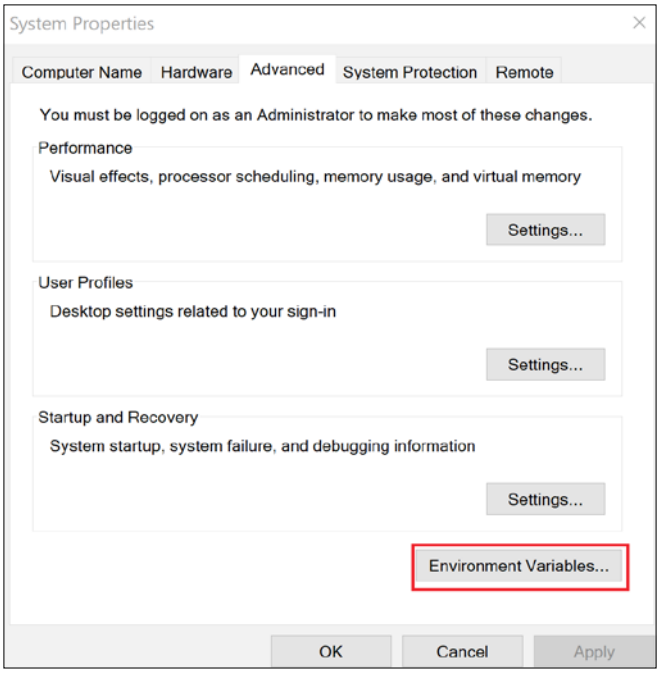


Figure 5: Creating a system variable for Python

Python 3 is usually listed in the **User variables for Jason**, but Python 2 is listed under the **System variables** as follows:

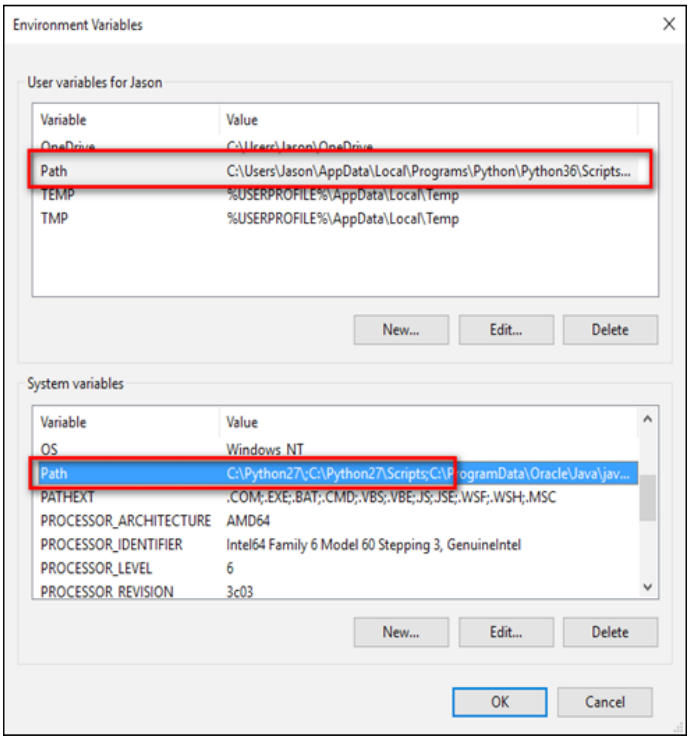


Figure 6: Showing how to add the Python installation location as system path

There are a few ways you can remedy this situation. The simplest way is to make changes that can give us access to `python` for Python 2 and `python3` for Python 3. For this, go to the folder where you have installed Python 3. It should be something like this: `C:\Users\[username]\AppData\Local\Programs\Python\Python36` by default.

Make a copy of the `python.exe` file, and rename that copy (not the original) to `python3.exe` as shown in the following screenshot:

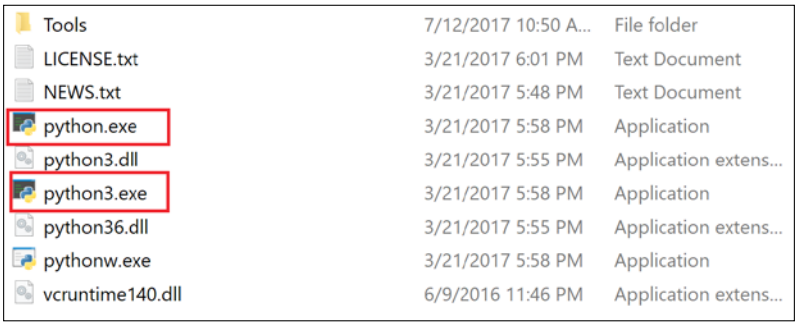


Figure 7: Fixing Python 2 versus Python 3 issue

Open a new Command Prompt (the environmental variables refresh with each new Command Prompt you open), and type `python3 --version`:

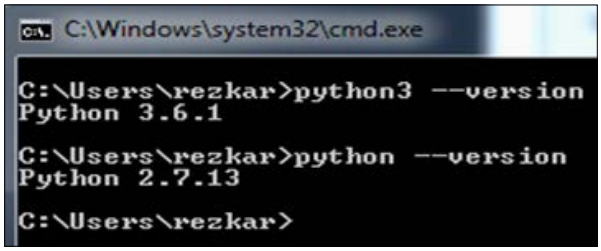


Figure 8: Showing Python 2 and Python 3 version

Fantastic, now you're ready for whatever Python project you want to tackle.

Installing Python on Linux

For those of you who are new to Python, Python 2.7.x and 3.x are automatically installed on Ubuntu. Make sure to check if `Python 2` or `Python 3` is installed using the following command:

```
$ python -V
>> Python 2.7.13
$ which python
>> /usr/bin/python
```

For Python 3.3+ use the following:

```
$ python3 -V
>> Python 3.6.1
```

If you want a very specific version:

```
$ sudo apt-cache show python3
$ sudo apt-get install python3=3.6.1*
```

Installing and upgrading PIP (or PIP3)

The `pip` or `pip3` package manager usually comes with your Ubuntu. Make check to sure if `pip` or `pip3` is installed using the following command:

```
$ pip -V
>> pip 9.0.1 from /usr/local/lib/python2.7/dist-packages/pip-9.0.1-
py2.7.egg (python 2.7)
```

For Python 3.3+ use the following:

```
$ pip3 -V
>> pip 1.5.4 from /usr/lib/python3/dist-packages (python 3.4)
```

It is to be noted that `pip` version 8.1+ or `pip3` version 1.5+ are strongly expected to give better results and smooth computation. If version 8.1+ for `pip` and 1.5+ for `pip3` are not installed, see the following command to either install or upgrade to the latest `pip` version:

```
$ sudo apt-get install python-pip python-dev
```

For Python 3.3+, use the following command:

```
$ sudo apt-get install python3-pip python-dev
```

Installing Python on Mac OS

Before installing the Python, you should install a C compiler. The fastest way of doing so is to install the `Xcode` command-line tools by running the following command:

```
xcode-select –install
```

Alternatively, you can also download the full version of `Xcode` from the Mac App Store.

If you already have `Xcode` installed on your Mac machine, do not install OSX-GCC-Installer. In combination, you can experience some unwanted issues that are really difficult to diagnose and get rid of.

Although Mac OS comes with a large number of Unix utilities, however, one key component called Homebrew is missing, which can be installed using the following command:

```
$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/
Homebrew/install/master/install)"
```

Set the Homebrew installation path to the `PATH` environment variable to the `~/.profile` file by issuing the following command:

```
export PATH=/usr/local/bin:/usr/local/sbin:$PATH
```

Now, you're ready to install Python 2.7.x or 3.x. For Python 2.7.x issue the following command:

```
$ brew install python
```

For Python 3 issue the following command:

```
$ brew install python3
```

Installing packages in Python

Additional packages (other than built-in packages) that will be used throughout this book can be installed via the `pip` installer program. We have already installed Python `pip` for Python 2.7.x and Python 3.x. Now to install a Python package or module, you can execute `pip` on the command line (Windows) or terminal (Linux/Mac OS):

```
$ sudo pip install PackageName # For Python3 use pip3
```

However, already installed packages can be updated via the `--upgrade` flag by issuing the following command:

```
$ sudo pip install PackageName –upgrade # For Python3, use pip3
```

Getting started with Python

In this sub-section, we will see some examples on how to get familiar with Python programming. I assume you already know the basic Python. However, I will provide some basic things in Python that will be needed in upcoming sections and chapters.

Python data types

Python has five standard data types as follows:

- Numbers
- String
- List
- Tuple
- Dictionary

Besides these, Python supports four different numerical types, such as:

- `int` (signed integers)
- `long` (long integers, can be represented in octal and hexadecimal too)
- `float` (floating point real values)
- `complex` (complex numbers)

Now you can assign values to the variables using the `=` sign as follows:

```
>>> counter = 100.50        # A floating point
>>> age   = 32              # An integer assignment
>>> name    = "Reza"        # A string
```

Python also allows you to assign a single value to several variables concurrently. For example:

```
>>> x = y = z = 50
```

Here we have created an integer with the value `50` and subsequently, all three variables are assigned to the same memory location. Furthermore, you can also assign multiple objects to multiple variables with ease.

For example:

```
>>> x,y,z = 50,30,"Reza"
```

Two integer objects (that is, `50` and `30`) will assign to variables `x` and `y` respectively. On the other hand, variable `z` will be assigned to string `Reza`.

Using strings in Python

Strings in Python are identified as a contiguous set of characters represented in quotation marks. Indexes start at `0` in the beginning of the string. The `+` sign is used as the string concatenation operator. Whereas the `*` is the repetition operator.

For example:

```
>>> message = 'Hello, world!'
>>> print message #complete string will be printed
>>> print message[0] #Only the first character will be printed
>>> print message[2:5] #Characters from 3rd to 5th will be printed
>>> print message * 2      # Prints string two times
>>> print message + "TEST" # Prints concatenated string
```

The preceding lines should produce the following output:

```
Hello, world!

H

llo

Hello, world!Hello, world!

Hello, world!TEST
```

Using lists in Python

Lists are one of the most versatile objects used in Python. A list contains items separated by commas enclosed within square brackets–that is, `[]`. Values in a list can be accessed using the slice operator (`[ ]` and `[:]`) with indexes starting at `0` in the beginning and the end at `n-1` considering the length of the list is n. The concatenation and repetition operation is similar to strings in Python. Let's see some examples:

```
>>> list1 = [ 'Ireland', 1985 , 4.5, 'John Rambo']
>>> list2 = ['USA', 1982 , 6.5, 'Sylvester Stallone']
>>> print list1     # Prints the complete list
>>> print list1[0]    # Prints only the first element of the list
>>> print list1[1:3]  # Prints elements starting from 2nd to 3rd
>>> print list1[2:]   # Prints elements starting from 3rd element
>>> print list1 * 2   # Prints the list 2 times
>>> print list1 + list2 # Prints the concatenated lists
```

This produces the following output:

```
        ['Ireland', 1985, 4.5, 'John Rambo']

        Ireland

        [1985, 4.5]

        [4.5, 'John Rambo']

        ['Ireland', 1985, 4.5, 'John Rambo', 'Ireland', 1985, 4.5,
'John Rambo']

        ['Ireland', 1985, 4.5, 'John Rambo', 'USA', 1982, 6.5,
'Sylvester Stallone']
```

Using tuples in Python

A tuple is another sequence data type similar to the list consisting of values separated by commas, but enclosed within parentheses. While the elements and size in a list can be changed, a tuple cannot be updated. Thus you can think of a tuple as a read-only list:

```
>>> tuple1 = ('Ireland', 1985, 4.5, 'John Rambo')
>>> tuple2 = ('USA', 1982, 6.5, 'Sylvester Stallone')
>>> print tuple1     # Prints the complete list
>>> print tuple1[0]  # Prints only the first element of the list
>>> print tuple1[1:3] # Prints elements starting from 2nd to 3rd
>>> print tuple1[2:]  # Prints elements starting from 3rd element
>>> print tuple1 * 2   # Prints the list 2 times
>>> print tuple1 + tuple2 # Prints the concatenated lists
```

This produces the following output:

```
>>> ('Ireland', 1985, 4.5, 'John Rambo')

        Ireland

        (1985, 4.5]

        (4.5, 'John Rambo')

        ('Ireland', 1985, 4.5, 'John Rambo', 'Ireland', 1985, 4.5,
'John Rambo')

        ('Ireland', 1985, 4.5, 'John Rambo', 'USA', 1982, 6.5,
'Sylvester Stallone')
```

Using dictionary in Python

Dictionaries are a kind of hash table. You can compare them with associative arrays or hashes in Perl. A dictionary consists of key-value pairs: where a key can be any Python type, but mostly are numbers and strings. A value on the other hand, can also be any arbitrary Python object. In Python, a dictionary is enclosed by curly braces ({}). The values are usually assigned and can be accessed using square braces ([]) or using the `get()` method. For example:

- An empty dictionary:
  ```
  >>> mydict = {}
  ```
- A dictionary with integer keys and string values:
  ```
  >>> mydict = {1: 'apple', 2: 'ball', 3: 'cat'}
  ```
- A dictionary with mixed keys:
  ```
  >>> mydict = {'name': 'John Rambo', 'numbers': [2, 4, 3]}
  ```
- Printing the whole dictionary:
  ```
  >>> print(mydict)
  ```
  Output: `{'name': 'John Rambo', 'numbers': [2, 4, 3]}`
- Accessing dictionary element using `[]`:
  ```
  >>> print(mydict['name'])
  ```
  Output: `John Rambo`
- Accessing dictionary element using the `get()` method:
  ```
  >>> print(mydict.get('numbers'))
  ```
  Output: `[2, 4, 3]`
- Updating a value:
  ```
  >>> mydict['name'] = 'Asif Karim'
  >>> print(mydict)
  ```
  Output: `{'name': 'Asif Karim', 'numbers': [2, 4, 3]}`
- Adding a new item:
  ```
  >>> mydict['address'] = 'Aachen, Germany'
  >>> print(mydict)
  ```
  Output: `{'address': 'Aachen, Germany', 'name': 'Asif Karim', 'numbers': [2, 4, 3]}`

- Removing an arbitrary item:

```
>>> mydict.popitem()
>>> print(mydict)
```

  Output: `{'name': 'Asif Karim', 'numbers': [2, 4, 3]}`

- Removing all items:

```
>>> mydict.clear()
>>> print(mydict)
```

  Output: `{}`

Using sets in Python

A set can be created by placing any number of items inside curly braces `{}`, separated by a comma. Items in a set can be of different types (integer, float, tuple, string, and so on). Alternatively, a set can be created using the built-in function `set()` of Python. For example:

- A set of integers:

```
>>>> mySet = {1, 2, 3, 4, 5}
>>> print(mySet)
```

  Output: `set([1, 2, 3, 4, 5])`

- A set of mixed datatypes:

```
>>> mySet = {4.0, "John R ambo", (1, 2, 3, 4, 5), 9}
>>> print(mySet)
```

  Output: `set([9, (1, 2, 3, 4, 5), 4.0, 'John R ambo'])`

- Inserting a single item to existing set:

```
>>> mySet.add(2.5)
>>> print(mySet)
```

  Output: `set([2.5, 9, (1, 2, 3, 4, 5), 4.0, 'John R ambo'])`

- Adding multiple elements:

```
>>> mySet.update([7,8,9])
>>> print(my_set)
```

  Output: `set([2.5, 4.0, 7, 8, 9, (1, 2, 3, 4, 5), 'John R ambo'])`

A particular item can be removed from set using `discard()` and `remove()`:

```
>>> mySet.remove(8)
>>> print(mySet)
```

Output: `set([2.5, 4.0, 7, 9, (1, 2, 3, 4, 5), 'John R ambo'])`

```
>>> mySet.discard(7)
>>> print(mySet)
```

Output: `set([2.5, 4.0, 9, (1, 2, 3, 4, 5), 'John R ambo'])`



Set and mutable elements: beware that a set cannot have mutable elements as its member such, list, set, or dictionary.

Functions in Python

In Python, a function is a first-class citizen: consisting of a group of related statements for performing a specific task. Functions help you gain modularity in your code. Since your program grows larger and larger, functions make it more organized and manageable. Thus it also helps us avoid repetition towards making the code reusable.

The basic syntax of declaring a function in Python is as follows:

```
def function_name(parameters):
... statement(s)
        return [expression_list]

def absolute_value(x):
  if x >= 0:
    return x
  else:
    return -x
```

Now the preceding function can be called as follows:

```
>>> absolute_value(10)
```

#Output: `10`

```
>>> absolute_value(-10)
```

#Output: `10`



Lines and indentation in Python: be aware that Python does not provide any brackets/braces (such as Java, C++, and so on.) to indicate blocks of code for a method or class definitions or flow control. Rather blocks of code are denoted by a line indentation. Fortunately or unfortunately, this convention is strictly enforced. The number of spaces in the indentation is variable. However, it is known that all statements within the block must be indented the same amount.

Now it's time to discuss some Object Oriented Programming (OOP) concepts. Like other OOP, classes in Python are also basic building blocks. However, for simplicity, we are not going to discuss most of the OOP concepts in this chapter but readers will get to know them in upcoming chapters.

Classes in Python

Similar to functions, a class can be defined using the keyword class. Once you create a class in Python, it creates a new local namespace; where all the attributes are defined. Well, an attribute can be data, set, list, dictionary, array, or a function:

```
class MyAbsClass:
  number = 20
  name = "John Rambo"

  def __init__(self, number=10):
    self.real = number

  def absolute_value( x):
    if x >= 0:      return x
    else:
      return -x
```

Now if we want to access the properties of the preceding class, we have to create an object of that class. This is also called instantiation of that class. Creating an object is similar to a function call:

```
>>> obj = MyAbsClass()
```



Instantiation

An object is used to call an instance of a class. This process is called instantiation.

Let's see the whole class containing some data and methods as follows:

```
class MyAbsClass:
  number = 20
  name = "John Rambo"

  def __init__(self, number=10):
    self.real = number

  def absolute_value( x):
    if x >= 0:
      return x
    else:
      return -x
  obj = MyAbsClass()
  value = obj.absolute_value(-10)
  print("The absolute value of -10 is : "+ str(value))
  print(obj.number)
  print(obj.name)
```

Output:

```
The absolute value of -10 is: 10

20

John Rambo
```

Now let's move to the next section, where we will discuss vectors, matrix, graph and tensors, and so on. Interested readers can refer to this URL for more extensive materials: `https://www.programiz.com/python-programming/`.

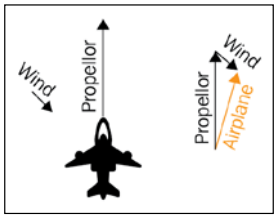
Vectors, matrices, and graphs

Learning how to perform several operations on matrices including inverse, eigenvalues, and determinants are some fundamental things before using some advanced topics such as (PCA, SVD, and so on. Thus, in this section, we will discuss vectors, metrics, and tensors, which are some fundamental topics for learning predictive analytics.

Vectors

The vector object is not a displayable object but is a powerful aid to 3D computations. Its properties are similar to vectors used in science and engineering. It can be used together with NumPy arrays.

For example, suppose an airplane is flying due north, there's a wind coming however from the north-west (see below figure). Now the question is how will the plane survive and move to the north?



If you look at the preceding figure carefully, there are two types of velocity that are active. The velocity caused by the wind and the propeller, respectively. The resultant velocity results in a marginally slower ground speed legend the plan east of north. Now the thing is if you observe the plane from the ground, it would seem that the plane is being moved sideways slightly as shown in the following figure:



Alternatively, you might have seen the birds struggling against the strong winds that seem to fly sideways. Using a vector of linear algebra, we can have a better explanation as to why that happens.

Python provides several modules for computing vector operations. For example, vectors is such a module that can be used to return a vector object with the given components, which are made to be floating-point (that is, 3 is converted to 3.0). Vectors can be added or subtracted from each other, or multiplied by an ordinary number. For example:

```
import numpy as np
from vectors import Point, Vector
v1 = Vector(1, 2, 3)
v2 = Vector(10, 20, 30)
```



Be aware that the Vectors module used for the preceding example code does not have support for Python3. So to install this module in Python 2, issue the following command on Linux:

```
$ pip install vectors
```

We can add a real number to a vector or compute the vector sum of two vectors as follows:

```
print(v1.add(10))
>> Vector(11.0, 12.0, 13.0)
print(v1.sum(v2))
>> Vector(11.0, 22.0, 33.0)
```

In the preceding cases, both methods return a vector instance. We can get the magnitude of the vector easily:

```
Print(v1.magnitude())
>> 3.7416573867739413
```

We can multiply a vector by a real number. The following returns a vector instance:

```
print(v1.multiply(4))
>> Vector(4.0, 8.0, 12.0)
```

To find the dot product of two vectors:

```
print v1.dot(v2)
>> 140.0
```

To use angle theta on the dot function, check the following case for which the dot product returns a real number:

```
print(v1.dot(v2, 180))
>> -4800.49306298
```

To perform the cross product of two vectors which returns a vector instance that is always perpendicular (90 degrees) to the other two vectors:

```
v1.cross(v2)
>> Vector(0, 0, 0)
```

We can also find the angle theta between two vectors. It is to be noted that the angle is measured in degrees:

```
v1.angle(v2)
>> 0.0
```

It is also possible to check if the two given vectors are parallel, perpendicular, or non-parallel to each other. For the following cases the result is either true or false:

```
v1.parallel(v2)
>> True
v1.perpendicular(v2)
>> False
```



For the mathematical explanation, please refer to this URL to get more insight: `https://www.mathsisfun.com/algebra/vectors.html`.

In the previous section, we mentioned buying a car that has some resemblance to feature engineering. Now let's see an example of how vectors could help us to select appropriate features.

Suppose you have the feature vectors of some potential cars. Now it's possible to figure out which two cars are similar by defining a distance function out of the feature vectors. One thing should be remembered: that comparing similarities and dissimilarities between data objects are one of the fundamental components in predictive analytics. Linear algebra helps us represent objects towards comparing.

One of the standard ways of doing so is calculating the Euclidian distance is an intuitive thinking of points in space. Suppose you have the following two feature vectors $X = (X_1, X_2 \ldots X_n)$ and $Y = (Y_1, Y_2 \ldots Y_n)$. Now the Euclidian distance can be calculated as follows:

$$\sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \ldots + (x_n - y_n)^2}$$

Thus if you have two points, for example *(0, 2)*, *(4, 0)*, the Euclidian distance would be:

= || *(0, 2)* – *(4, 0)* ||

= || *(-4, 2)* ||

= $\sqrt{(-4)^2 + (2)^2}$

= $\sqrt{20}$

= *4.4722*

This is called L2 norm and it is actually one of the many possible distance functions. In the real world, a more complex distance function is used. We will see it in upcoming chapters.

Matrices

A matrix is a 2D array for storing real or complex numbers. In a real matrix, all of its elements r belong to R. Similarly, a complex matrix has entries c in C.

Matrix addition

Given that two matrices have the same dimension, they can be added together, which results in a new matrix with the same dimensions: each element is the sum of the corresponding elements of the previous matrices. Suppose we have the following matrix `A` and `B` as follows:

```
A = np.matrix(
    [[3, 2],
    [4, 6]])
B = np.matrix(
    [[1, 4],
    [2, 0]])
```

Now the addition of the precoding matrices can be computed as follows:

```
C = A + B
print(C)

>> [[ 8 -5]
    [ 6 15]]
```

Matrix subtraction

Similar to addition, in matrix subtraction, each element of one matrix is subtracted from the corresponding element of the other. If a scalar is subtracted from a matrix, the former is subtracted from every element of the latter:

```
A = np.matrix(
    [[1, 4],
     [2, 9]])
```

```
B = np.matrix(
    [[7, -9],
     [4, 6]])

>> [[-6 13]
   [-2  3]]
```

Multiplying two matrices

Finding the product of two matrices is also required in many cases. When two matrices are multiplied, the result is simply expanded with each column of the result obtained by using the corresponding column of the second matrix. Suppose we have the following matrix `A` and `B` as follows:

```
A = np.matrix(
    [[1, 4],
     [2, 0]])
B = np.matrix(
    [[3, 2],
     [4, 6]])
```

Now the addtion of the preceding matrixes can be computed as follows:

```
C = A * B
print(C)
>> [[23 15]
   [50 36]]
```

Furthermore, it is also required sometimes to add a constant to each element in a matrix, this is called summing of a matrix and a scalar. Let's add a constant, say `8`, to matrix `A` as follows:

```
B =  A + 8
Print(B)
>> [[ 9 12]
   [10  8]]
```

Finding the determinant of a matrix

The determinant can be computed from the elements of a square matrix. The determinant of a matrix `A` is denoted `det(A)`, `det A`, or `|A|`. Often, determinant is viewed as the scaling factor of the transformation described by the matrix itself. One of the interesting facts is that the determinant of a product of matrices is always equal to the product of determinants:

```
A = np.matrix(
    [[3, 2],
     [4, 6]])
deter = np.linalg.det(A)
print(deter)
>> 10.0
```

Finding the transpose of a matrix

In matrix transpose, rows become columns and columns should be rows. Suppose we have the following matrix:

```
matrix = np.matrix(
    [[3, 6, 7],
     [2, 7, 9],
     [5, 8, 6]])
transpose = np.transpose(matrix)


>> [[3 2 5]
      [6 7 8]
      [7 9 6]]
Matrix inversion
```

We have seen addition, subtraction, and multiplication of matrixes, however, there is no such division. Fortunately, there is a matrix construct similar to that of division, and it is central to much of the work of the analyst. The key ingredient is the use of the inverse of a matrix.

Let's see an example:

```
matrix = np.matrix(
    [[3, 6, 7],
     [2, 7, 9],
     [5, 8, 6]])
```

```
inverse = np.linalg.inv(matrix)
print(inverse)
>> [[ 1.2  -0.8  -0.2 ]
 [-1.32  0.68  0.52]
 [ 0.76 -0.24 -0.36]]
```

It is to be noted that the multiplication of the original matrix and the inverse one always produces a square matrix also called identity matrix. More formally:

```
Inv(A) * A = I
```



Identity matrix

An identity matrix is a square matrix with ones on the diagonal from the upper left to lower right and zeros elsewhere. For example:

```
I = [[1 0 0],
     [0 1 0],
     [0 0 1]]
```

Solving simultaneous linear equations

Matrix inversion is often used to solve a set of simultaneous linear equations. For example, how to find the solution of $Ax=B$: that is the value of x that satisfies this equation. Suppose we have the following matrix `A` and `B` as follows:

```
A = np.matrix(
    [[1, 4],
     [2, 0]])

B = np.matrix(
    [[3, 2],
     [4, 6]])
```

Now the solution can be computed by calling the solve method as follows:

```
X = np.linalg.solve(A, B)
print(X)
>> [[ 2.    3.  ]
    [ 0.25 -0.25]]
```

Eigenvalues and eigenvectors

In the following figure, original matrix A acts by extending the vector x without changing its direction. Thus, x is an eigenvector of matrix A; whereas the scale factor λ is the eigenvalue corresponding to the eigenvector x:

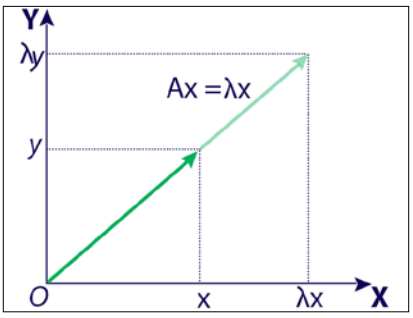


Figure 13: demonstrating eigenvalues and eigenvectors

For example:

```
matrix = np.matrix(
    [[3, 6, 7],
     [2, 7, 9],
     [5, 8, 6]])

eigvals = np.linalg.eig(matrix)
print(eigvals)
>> (array([ 18.03062661,   0.53948277,  -2.57010939]),
    matrix([[-0.52213277, -0.69701957, -0.23035157],
        [-0.59400273,  0.64684805, -0.6406727 ],
        [-0.6119952 , -0.3094371 ,  0.73244566]]))
```

In the preceding output, the array signifies eigenvalues and the matrix signifies the eigenvector.

Span and linear independence

The span of vectors v_1, v_2... v_n is the set of linear combinations such that: $c_1v_1 + c_2v_2 + ... + c_nv_n$, which is a vector space called V. Now if we further expand this idea such that $S = \{v_1, v_2... v_n\}$ is a subset of V, then **Span**(**S**) is equal to V. More formally, S spans V if and only if every vector v in V can be expressed as a linear combination of vectors in S–that is:

$$v = c_1v_1 + c_2v_2 + \ldots + c_nv_n$$

Let's see an example, suppose we have the following set $S = \{(0,1,1), (1,0,1), (1,1,0)\}$. Obviously, this set spans R_3 Therefore, vector (2, 4, 8) can be expressed as a linear combination of vectors in S.

To solve this, we can say that a vector in R_3 has the form $v = (x, y, z)$. Therefore, it would be enough showing that every such v can be expressed as follows:

```
(x,y,z) = c1(0, 1, 1) + c2(1, 0, 1) + c3(1, 1, 0)
        = (c2 + c3, c1 + c3, c1 + c2)
```

Now the preceding relation can be written more explicitly as follows:

```
c2 + c3 = x
      c1 + c3 = y
      c1 + c2 = z
```

If we can recall our undergraduate mathematics, the preceding relation can be written in matrix form as follows:

$$A^{-1} = \begin{pmatrix} -.5 & .5 & .5 \\ .5 & -.5 & .5 \\ .5 & .5 & -.5 \end{pmatrix}$$

Now the preceding relation is pretty expressible in the equation form as follows:

$Ax = B$

If you look carefully, the determinant of matrix A is 2, that is, $det(A) = 2$. This also signifies that A is non-singular. Therefore, there exists a solution such that $x = A\text{-}1B$. Now as asking, to write $(2, 4, 8)$ as a linear combination of vectors in S, we now find the following:

$$A^{-1} = \begin{pmatrix} -.5 & .5 & .5 \\ .5 & -.5 & .5 \\ .5 & .5 & -.5 \end{pmatrix}$$

We also find the following:

$$X = \begin{pmatrix} -.5 & .5 & .5 \\ .5 & -.5 & .5 \\ .5 & .5 & -.5 \end{pmatrix} \begin{pmatrix} 2 \\ 4 \\ 8 \end{pmatrix} = \begin{pmatrix} 5 \\ 3 \\ -1 \end{pmatrix}$$

Finally, we have:

(2,4,8) = 5(0,1,1) + 3(1,0,1) + (-1)(1,1,0)

So far, we know how to find out if a group of vectors spans over a vector space. Now the question is are there any redundancies in the vectors span? That is, is there a smaller subset of *S* such that it also Span(*S*) then one of the given vectors can be rewritten as a linear combination of the others, such that:

$$v_i = c_1 v_1 + c_2 v_2 + \ldots + c_{i-1} v_{i-1} + c_{i+1} v_{i+1} + c_n v_n$$

If the preceding relation is satisfied then *S* is a linearly dependent set, otherwise, *S* is linearly independent. There is another way of checking that a set of vectors are linearly dependent. Now let's see an example of the preceding definition.

Given a set $S = \left\{\cos^2 t, \sin^2 t, 4\right\}$. Now it can be seen that *S* is a linearly dependent set of vectors since $4 = 4\cos^2 t + 4\sin^2 t$

Now we know some basic concepts from linear algebra to construct a predictive analytics model, yet often we need to deal with high dimensional data to make the prediction more meaningful by taking out less significant or correlated features. PCA algorithm comes in handy to deal with the curse of dimensionality.

Principal component analysis

In predictive analytics, most often you will face an issue about the data dimensionality also called the curse of dimensionality. You need to deal with too many variables having less important ones as well. Thus when a dataset has too many variables, there are only a few possible situations you may encounter:

1. You find that most of the variables are correlated–that is, have a mutual relationship or connection, in which one thing affects or depends on another.
2. Then say you lose patience and decided to train the model using the whole data. This results in a very poor accuracy and your boss is unhappy.

3. Naturally, you might be indecisive about what to do next.
4. Finally, you start thinking to get rid of the issue by finding only important variables–that is, feature selection.

Believe it or not, handling this issue is not that difficult, but the usage of some statistical techniques such as factor analysis, singular value decomposition, and principal component analysis help overcome such difficulties.

PCA is a statistical method for extracting important variables from a high-dimensional dataset (having so many variables). In other words, PCA extracts a low-dimensional set. But it tries to capture as much information as possible called components form. This is not full of surprise, but with fewer variables, the interactive visualization becomes more meaningful. In particular, PCA is more useful when dealing with higher dimensional data–that is, at least three dimensions.



Principal components

In a PCA algorithm, a principal component is a normalized linear combination of the original predictors in a dataset.

The PCA is all about performing operations on a symmetric correlation or covariance matrix. Therefore, the matrix has to be numeric having standardized data. Let's say we have a dataset of dimension *300 (m) × 300 (n)*. Where *m* signifies a number of observations and *n* is the number of predictors–that is, response. Since the dataset is high-dimensional–that is, having *n = 300*, theoretically there could be *n(n-1)/2*–that is, *44850* scatter plots for analyzing the available relationship in the variable.

You're right, yes, performing an exploratory analysis on this type of data is really difficult–that is, the curse of dimensionality. One approach could be selecting a subset of *n (n << 300)* predictors to capture as much information as possible without sacrificing the quality much. Now if you plot such data, the observation in the resultant is a low dimensional space.

For example, the following figure shows the transformation of three-dimensional gene expression data, which is mainly located within a two-dimensional subspace. PCA is then used to visualize this data by reducing the dimensionality of the data. If you look at the graph carefully, you can observe that each subsequent dimension is a linear combination of *n* features:

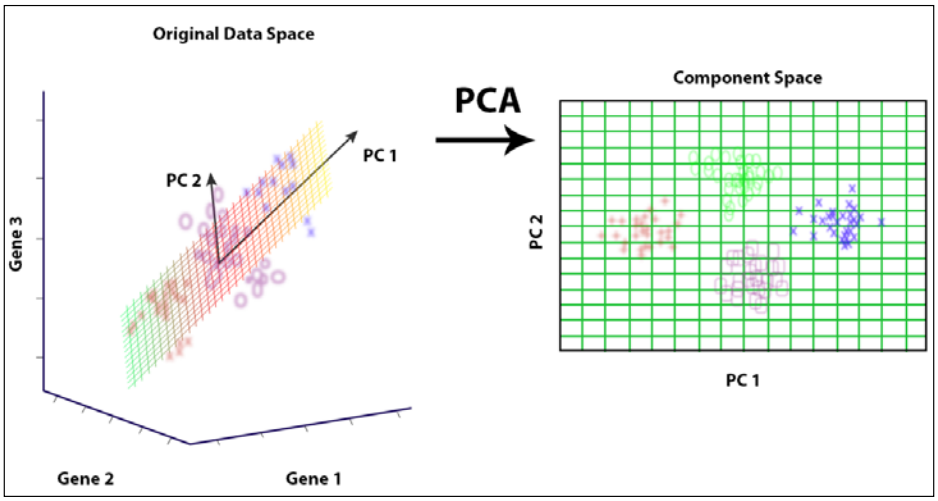


Figure 19: Using PCA in bioinformatics with high dimensional data

In the preceding figure, **PC1** and **PC2** are the principal components.

Singular value decomposition

If matrix *A* has a matrix of eigenvectors **P** that is not invertible, then *A* does not have Eigen decomposition too. However, if *A* is an *m x n* real matrix with *m>n*, then the original matrix *A* can be written using a so-called singular value decomposition of the form (as the product of three matrices) *U*, *Σ*, *V**. Suppose we have the following matrix:

```
matrix = np.matrix(
    [[6, 8],
    [5, 7]] )
```

Now the SVD can be computed by calling the `svd()` method from the NumPy module of Python as follows:

```
svd = np.linalg.svd(matrix)
```

This is an array that has three fields–that is, `u`, `sigma`, and `v`:

```
U = svd[0]
Sigma = svd[1]
V = svd[2]
```

For better interpretation of the preceding result, let's do some transformation–that is, converting each field as a list as follows:

```
U = U.tolist()
Sigma = Sigma.tolist()
V = V.tolist()
```

Let's compute the matrix production consisting of the three components:

```
matrix_prod = [[,$U$', ,', ,$\Sigma$', ,$V^*$', ,'],
    [U[0][0], U[0][1], Sigma[0], V[0][0], V[0][1]],
    [U[1][0], U[1][1], Sigma[1], V[1][0], V[1][1]]]
```

Let's convert the preceding matrix into a table for the SVD:

```
table = FF.create_table(matrix_prod)
```

Finally, display the components as follows:

```
py.plot(table, filename='Matrix_SVD')
```

The output is as follows:

U		Σ	V^*	
-0.758111069207	-0.652125453227	13.1900344373	-0.592060143475	-0.805893781157
-0.652125453227	0.758111069207	0.151629626861	-0.805893781157	0.592060143475

Figure 20: Showing each and singular value that are decomposed using SVD

Data compression in a predictive model using SVD

The SVD is a widely used decomposition technique in computer science, math, and other disciplines. In this section, I will provide a small glimpse of that in a data compression technique. Suppose we have a matrix A with rank 200–that is, the columns of this matrix span a 200-dimensional space. Representing and encoding this large matrix on your PC will take a pretty good amount of memory.

SVD comes at the front end to efficiently handle this issue without sacrificing by approximating the original matrix with one of lower rank. Suppose we approximate it as a matrix with rank 100. Now the question is how close can we get to this matrix by storing only 100 columns? Another question could be that can we use a matrix of rank 20? In other words, is it possible to summarize all of the information of this very dense, (that is, 200-rank) matrix with only a rank 20 matrix?

If we want to keep say 90% of the original information, it would be enough computing of the sums of singular values until we reach 90% of the sum. Consequently, the rest of the singular values will be discarded. Since the SVD algorithm only stores the columns of U and V we greatly reduce the memory usage since we set elements on the diagonal of Σ to 0. Images are represented in a rectangular array where each element corresponds to the grayscale value for that pixel.

For colored images, we have a three-dimensional array of size $m \times n \times 3$, where m and n represent the number of pixels vertically and horizontally, respectively, and for each pixel, we store the intensity for colors red, green, and blue. The three signifies that this is a three-dimensional space. Now are going to repeat the preceding low-rank approximation procedure on a larger matrix. The resulting three-dimensional array will be a pretty good approximation of the original image. Here's the original image:



Figure 21: Original image having 1000*1600 dimensions and takes 37500 KB

Now at first, we do the singular value decomposition using the SVD algorithm for all the red, green, and blue components. Then we try to reconstruct the whole image using the best rank 10 approximations, the size of the compression one is and see how much space we require to store the compressed one. We have observed that the compressed matrices have a total size of about 610 KB, which is about 61.5 times less. Let's see the compressed one:

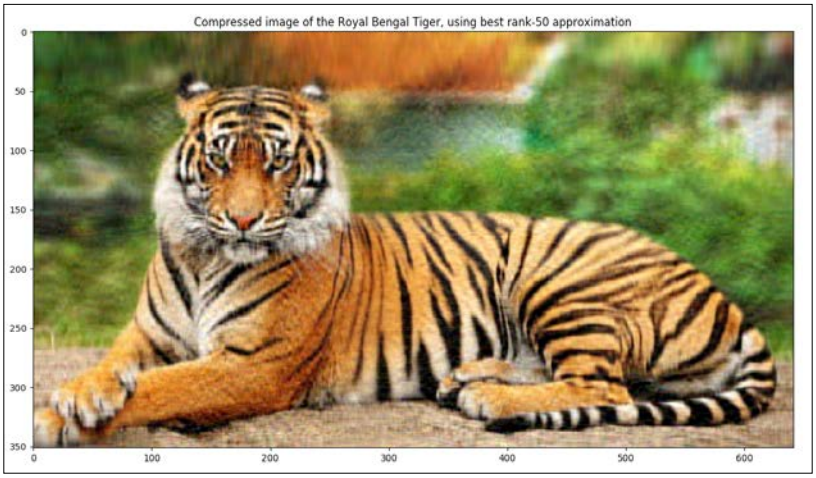


Figure 22: The compressed one after best rank 10 approximations has only 610KB in size with lower resolution

However, using the best rank 50 approximations, we have observed that the compressed matrices have a total size of about 3048 KB, which is about 12 times less. Let's see the compressed one:

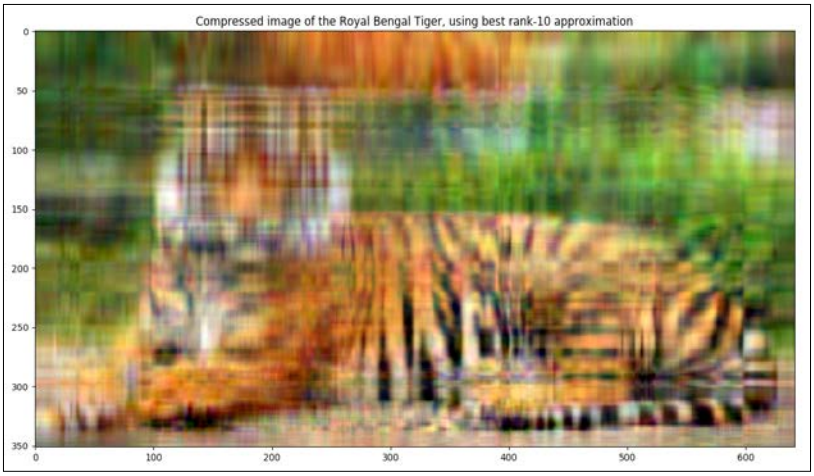


Figure 23: The compressed one after best rank 50 approximations has only 3048 KB in size with better resolution

Using the best rank 200 approximations, the size of the compressed one is:

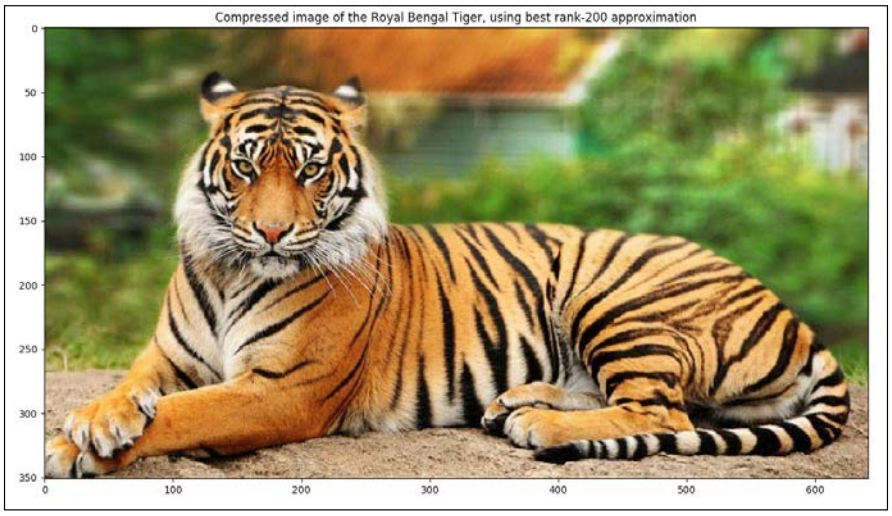


Figure 24: The compressed one after best rank 200 approximations with full resolution

Now the preceding images of the tiger can be generated using SVD. Just execute Python3 `SVD_Demo.py`.

Predictive analytics tools in Python

We will see throughout this book that Python is a great tool for developing predictive models that can be used for predictive analytics. There are many other tools and frameworks have been developed around it such as TensorFlow, H20, Caffe, Theano, PyTorch, and so on.

TensorFlow is mathematical software and an open-source software library for machine intelligence, developed in 2011, by the Google brain team. The initial target of TensorFlow was to conduct research in machine learning and in deep neural networks. However, the system is general enough to be applicable in a wide variety of other domains such as numerical computation using data flow graphs that enables machine learning practitioners to do more data-intensive computing. It provides some robust implementation of the widely used implementation of deep learning algorithms. TensorFlow offers you a very flexible architecture that enables you to deploy computation to one or more CPUs or GPUs in a desktop, server or mobile device with a single API.

Theano is probably the most widespread library. Written in Python, one of the languages most used in the field of machine learning (also in TensorFlow), allows the calculation using the GPU, which has 24x performance, even better than the CPU. It allows you to define, optimize, and efficiently evaluate complex mathematical expressions such as multidimensional arrays.

Other predictive analytics tools include Matlab, Torch, Weka, KNIME, SAS, SPSS, R, Mahout, Minitab, SAM, StatSoft, and so on.

Throughout this book, we will be using TensorFlow only. A more detail discussion is beyond the scope of this book. However, interested readers can read about and explore other tools and frameworks.

Summary

Linear algebra plays an important role in predictive analytics especially in machine learning and also in broader mathematics. In this chapter, we have tried to provide a very basic introduction to predictive analytics. We have seen where and why to use this. Then we have seen how linear algebra helps in learning predictive modeling. We have seen how to perform SVD and PCA operations using Python modules. Finally, we have had a quick look at the widely used predictive analytics tools in Python.

In the next chapter, we will cover some statistical concepts before getting started with predictive analytics formally. For example, random sampling, hypothesis testing, chi-square test, correlation, expectation, variance, covariance and Bayes' rule, and so on. In the second part of this chapter, we will discuss probability and information theory for the predictive analytics.

Information theory that deals with the quantification, storage, and communication of information will be discussed too. Probability theory, which is a branch of mathematics concerned with probability, and the analysis of random phenomena will be discussed in the last section to help the data scientist gain more insight while performing predictive analytics.

2
Statistics, Probability, and Information Theory for Predictive Modeling

This chapter covers some statistical, probabilistic, and information theory concepts before getting started with predictive analytics. Some examples are random sampling, hypothesis testing, chi-square testing, correlation, expectation, variance, covariance and Bayes' rule, and so on. In the second part of this chapter, we will discuss probability and information theory for predictive analytics. Information theory studies the quantification, storage, and communication of information. Probability theory is the branch of mathematics concerned with probability, the analysis of random phenomena.

The central objects of probability theory are random variables, stochastic processes, and events. Considering these topics, in this chapter, we will cover the required background in probability and information theory that you, as a data scientist, should be aware of, in order to get more insight while developing predictive models.

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

- Statistics for predictive modeling
- Basic probability for predictive modeling
- Information theory for predictive modeling

Using statistics in predictive modeling

In this section, we will discuss some widely used statistical concepts required in predictive analytics, followed by some basic understanding of predictive modeling, such as random sampling, central limit theorem, hypothesis testing using chi-square tests, correlation, expectation, variance and covariance, and so on. We will see an example of each of them.

Predictive modeling uses statistics to predict outcomes. In many cases, the trained model predicts the probability of an outcome based on a set of inputs. Statistics are used to determine the variable importance and their correlation.

In *Chapter 1*, *Basic Python and Linear Algebra for Predictive Analytics*, we argued that feature vectors are everywhere: they are used in both learning and inferencing stages in predictive analytics. Consequently, nearly any regression, classification, or clustering model can be used for prediction purposes. We now need to know how to establish the relationship between data attributes to generate feature vectors to be used to train the statistical models.

Once we have our predictive models trained with the training data, we need to deploy them commercially for predictive analytics. For example, if we want to develop and deploy the animal/bird classifier described in *Chapter 1*, *Basic Python and Linear Algebra for Predictive Analytics*, commercially, we can follow the predictive analytics value chain depicted in Figure 1:

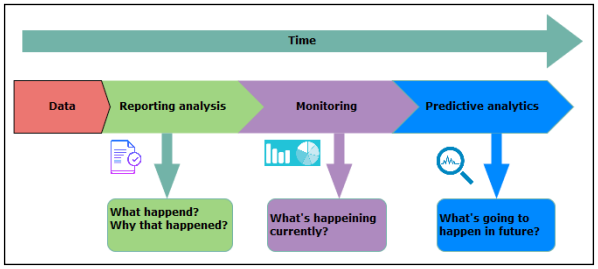


Figure 1: Predictive analytics value chain

Unfortunately, the preceding value chain is abstract until we construct a statistical model.

Statistical models

A statistical model (SM) is at the core of predictive modeling. A statistical model describes the statistical characterization of numerical data, extrapolation or interpolation of data based on some best-fit, estimating the probabilistic future behavior of a system based on past behavior, and error estimates of observations. Apart from these, an SM can also be used for spectral analysis of data or to model generated output.

More formally, we can also think of a statistical model as an empirical model that relates a set of inputs (predictors, *X*s) to one or more outcomes (responses, *Y*s) by separating the response variation into signal and noise. Formally, a statistical predictive model can be defined as follows:

$$Y = f(X) + E$$

In the preceding equation:

- *Y* is one or more continuous or categorical response outcomes
- *X* is one or more continuous or categorical predictors
- *f(X)* describes predictable variation in *Y* (signal)
- *E* describes a non-predictable variation in *Y* (noise)

We can also train machine learning algorithms that assume a type of model of a specific form will describe the relationship and find the parameters to fit the model to the data. In these types of machine learning models, statistical models are applied as notions of a fit; overfitting and underfitting happen when the model is too specific or not specific enough in its ability to generalize beyond observed data.

Parametric versus nonparametric model

Simpler models are easier to understand and use than more complex models. As such, it is a good idea to start with the simplest models for a problem and increase complexity as you need. For example, assume a linear form for your model before considering a nonlinear form, or a parametric before a nonparametric model. Broadly speaking, there are two classes of predictive models: parametric and nonparametric.

Parametric predictive models

Numerical methods are used to optimize parameters, for example, gradient descent, competitive learning, and so on. Parametric models have a finite, fixed number of parameters θ, regardless of the size of the dataset. Given θ, the predictions are independent of the data D that can be defined as follows:

$$p(x, \boldsymbol{\theta} \mid \boldsymbol{D}) = p(x \mid \boldsymbol{\theta})\, p(\boldsymbol{\theta} \mid D)$$

The parameters are a finite summary of the data. We can also call this model-based learning (for example, a mixture of k Gaussians). We will see an example of these types of models and algorithms in the upcoming chapters.

Nonparametric predictive models

Nonparametric models have a decision tree-like form that is typically optimized using an exhaustive search. Nonparametric models allow the number of "parameters" to grow with the dataset size, or alternatively, we can think of the predictions as depending on the data, and possibly a usually small number of parameters a:

$$p(x \mid \boldsymbol{D}, \alpha)$$

This type of model is often called memory-based learning (for example, kernel density estimation). Now, in order to develop either a parametric or nonparametric model, some statistical concepts are essential.

Population and sample

Population refers to the whole set of values or individuals we are interested in. The sample is a subset of the population and is the set of values we actually use in our estimation while developing a predictive model:

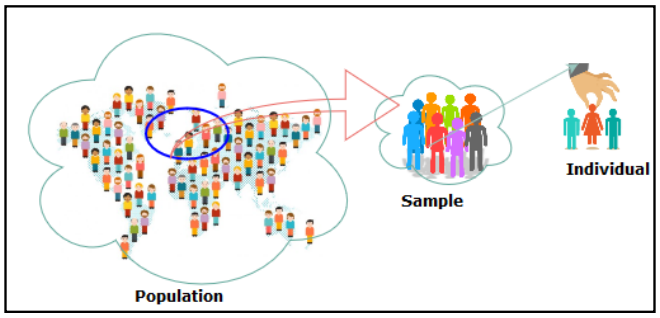


Figure 2: Population versus sample

For example, if we want to know the average height of the people in Bangladesh, that is your population: the population of Bangladesh.

Obviously, it would be a quite large number, for example, 160 million numbers, but we say we won't be able to get the height for everyone there. So in a statistical model, we draw a sample to get some observations by only collecting the height of some of the people in Bangladesh, and then we can make some inferences based on that.

Random sampling

Random sampling refers to a subset of a population, where each individual of the subset has an equal probability of being selected from the population. A simple example would be randomly taking 10 people from Bangladesh that has a population of 100 people. We are selecting a subset (10 people) of the population (100 people), where the probability of being selected is equal for each and every person while computing the average height:

```
>>>
import random
l = list(range(0, 10))
print(l)
>>>
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

r = random.sample(l, 5)
print(r)
>>>
[2, 1, 8, 4, 3]
```

Expectation

The expectation of a discrete random variable is the weighted average of all values, where each value is weighted by its probability of being selected. The expected value of a random variable X is given by:

$$E(X) = \sum x_i p_i$$

Where x_i is the value of the ith outcome and p_i is the probability of the ith outcome. For example, if we consider rolling dice, the probability of getting *1*, P(1) is ⅙, similarly, the probability of getting any other number is ⅙. So the expected value of a rolling dice is the weighted average of all the numbers, that is:

E(X) = 1(⅙) + 2(⅙) + 3(⅙) + 4(⅙) + 5(⅙) + 6(⅙) = 3.5

Now, using NumPy, we can make our life easier as follows:

```
>>>
import numpy as np
numbers = [1, 2, 3, 4, 5, 6]
num_array = np.array(numbers)
exp = np.mean(num_array)
print(exp)

>>>
3.5
```

Central limit theorem

The **central limit theorem** (**CLT**) is one of the fundamental theorems in statistics. It states that the average of all samples of a sufficiently large sample size with independent and identically distributed variables is approximately equal to the mean of the sample space, regardless of the size of the distribution with a finite variance level.

A simple example would be to calculate the average height of people in Dhaka city. It will be a lot of work if we take the height of each and everyone in the city and then average it. Instead, the central limit theorem states that taking a sufficiently large sample and averaging the height of that sample will give an approximately equal result to if we did it for all people in the city.

So, if we randomly select 1,000 people, then their average should be almost same as the arithmetic mean of people in the whole city. Now, if you're given enough samples, and if you plot the results on a histogram, it will approach what is known as a normal bell curve:

1. Take a random sample from your population.
2. Take the mean of your sample.
3. Plot your sample on a histogram:

```
from numpy import random
from matplotlib import pyplot as plt
X = random.random_integers(10, size = 1000)
plt.hist(X)
plt.title("Frequency distribution")
plt.xlabel("Integers")
plt.ylabel("Value")
plt.show()
>>>
```

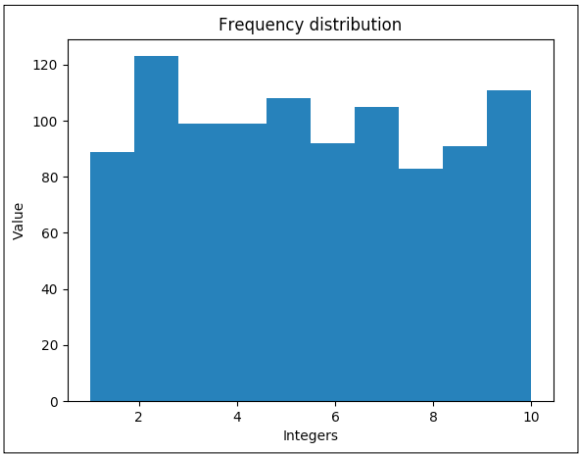


Figure 3: Frequency distribution (normal)

Now, repeat this 1000 times and you will have what looks like a normal distribution bell curve:

```
>>>
Y = random.normal(size=1000)
print(Y)
plt.hist(Y)
plt.title("Frequency distribution")
plt.xlabel("Integers")
plt.ylabel("Value")
plt.show()
>>>
```

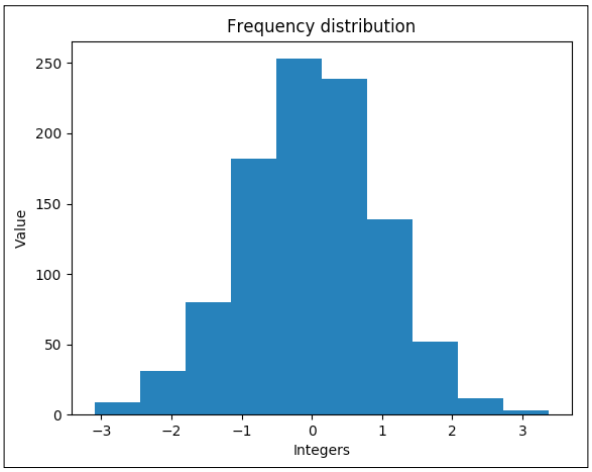


Figure 4: Frequency distribution (bell-shaped)

Of course, under certain circumstances, the CLT tells me no matter what my population distribution looks like, if I take enough means of sample sets, my sample distribution will approach a normal bell curve. It is also suggested that the sample size should be at least 30, but there is no upper limit for the sample size. The bigger the sample size, the more accurate results we will get.

Skewness and data distribution

In general, if the distribution is normal (a bell curve), then 30-40 samples are enough to get the average of the whole population; however, if the distribution is skewed, then larger samples are required to get a better approximation.

Negative skew means the distribution of the dataset is skewed to the left, that is, it has a tail on the left. Positive skew means the distribution of the dataset is skewed to the right, with a long tail. Whereas, a normal distribution is not skewed, as shown in the following figure:

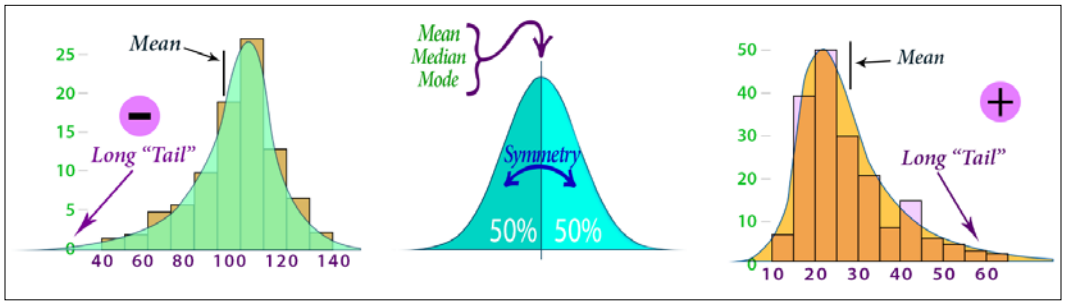


Figure 5: Skewness and the data distribution

Now, the mean, median, and mode can be calculated using either Numpy or SciPy packages, as follows:

```
import numpy as np
from scipy import stats
data = np.array([3, 5, 9, 2, 7, 3, 6, 9, 3])
# Mean
dt_mean = np.mean(data);
print("Mean :", round(dt_mean, 2))
# Median
dt_median = np.median(data);
print("Median :", dt_median)
# Mode
dt_mode = stats.mode(data);
print("Mode :", dt_mode[0][0])
>>>
```

```
Mean : 5.22
Median : 5.0
Mode: 3
```

Standard deviation and variance

Suppose you have some numbers, for example, age, weight, and height. Now, the **Standard Deviation** (**SD**) is a measure of how to spread out these numbers. It is often expressed with the σ, that is, the Greek letter sigma. Less technically, it is the square root of the variance.

Now, the term variance can be defined as the average of the squared differences from the mean of these numbers.) In other words, the variance is the expectation of squared deviation from the mean.

Often, the variance is used as the measure of the spread of the probability distribution (don't worry, we will see what probability distribution is in the next section) and it is given by the following equation:

$$\sigma^2 = \frac{\sum (X - \mu)^2}{N}$$

Here, σ^2 is the variance and μ is the arithmetic mean. X is an individual data point and N is the total number of data points. Now, the standard deviation, σ, can be calculated as follows:

$$\sigma^2 = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

Now, from the preceding equation, it is clear that to compute the SD, we need the values of the mean and variance precomputed.

Variance can be used as one of the key parameters in investment and asset allocation. Along with correlation (we will see this term's definition shortly), the variance of asset returns helps the investors to develop an optimal portfolio by optimizing return volatility trade-off. As a result, risk or volatility is often expressed as an SD rather than variance because of its expressiveness and easy interpretation.

Let's look at a concrete example. Suppose the returns for a stock are *20%* in year one, *10%* in year two, *-15%* in year three and *5%* in year four. The mean μ:

= (20% + 10% + (-15%) + 5%) / 4 = 20%/4 = 5%

Now, we need to calculate the difference from the mean to get the variance:

Variance σ^2 = (15%) 2 + (5%) 2 + (-20%) 2 + (0%) 2 / 4 = 162.5%

Now, taking the square root of the variance yields the SD of 12.748% for the returns. Now, using the Scipy or NumPy, we can easily compute the SD as follows:

```
>>>
import numpy as np
returns = [20, 10, -15, 5]
ret_arr = np.array(returns)
mean = np.mean(ret_arr)
variance = np.var(ret_arr)
sd = np.std(ret_arr)
print("Mean:", mean)
print("Variance:", variance)
print("Standard deviation:", sd)

>>>
Mean:  5.0
Variance: 162.5
Standard deviation: 12.747548784
```

Therefore, these Python packages make our lives easier. Now, let's see some other concepts that are also often needed when developing predictive models. One thing that is important to know is the difference between the population SD and sample SD that is,

Our population has only four values, but if the data is a sample taken from a bigger population, then the calculation changes! For the population, we divide by *N* when calculating variance, but for the sample, we divide by *N-1* when calculating variance. This means, for the sample, we have a slightly different formula for the sample SD:

$$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}\left(x_i - \overline{x}\right)^2}$$

Covariance and correlation

Covariance is a measure showing the extent to which two random variables change in tandem. Correlation, on the other hand, is also a statistical measure that indicates how strongly two variables are related.

Statically saying covariance lies between -∞ and +∞ but is the measure of correlation. Correlation is the scaled version of covariance but lies between *-1* and *+1*.

Thus, both the covariance and correlation are measures of the relationship between two variables, but covariance also gives a measure of the degree of their relationship:

```
>>>
import numpy as np
a = np.array([2.1, 2.5, 4.0, 3.6])
b = np.array([8,12,14,10])
np.cov(a,b)
```

This will return a 2 x 2 matrix with the following values:

```
cov(a,a)   cov(a,b)
cov(b,a)          cov(b,b)
```

Since both `a` and `b` are one-dimensional sequences, `np.cov(a,b)[0][1]` is equivalent to your `cov(a,b)` and gives you the correct covariance, which is as follows:

```
cv = np.cov(a,b)[0][1]
print(cv)
>>>
1.53333333333
```

When two datasets are strongly linked together, we say they have a high correlation. Correlation is positive when two values increase together but is negative when one value decreases as the other increases:

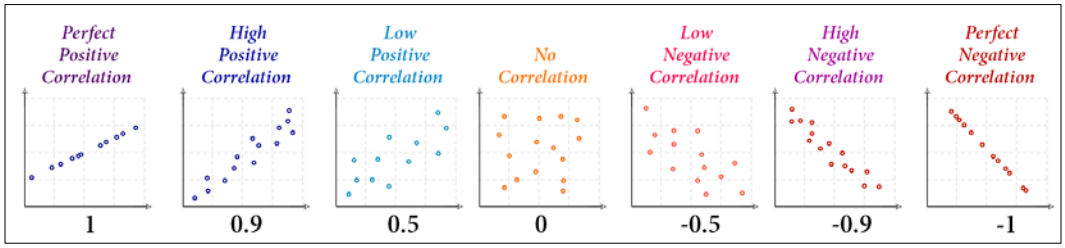


Figure 6: Negative versus positive correlation

In the preceding figure, we can see that correlation can be either perfect positive (**1**), no correlation (**0**), or perfect negative correlation (**-1**). Now, in order to calculate the correlation between two list or array elements, we can either use NumPy or Scipy. Both of them are pretty simple and straightforward:

```
>>>
import numpy as np
a = np.array([2.1, 2.5, 4.0, 3.6])
b = np.array([8, 12, 14, 10])
```

Now, using `np.corrcoef(a, b)` gives you a 2x2 matrix containing the following code:

```
[[1          correlation(a, b)]
[correlation(b, a)          1]]
```

However, `corrcoef(a, b)[0, 1]` is equivalent to your `np.corrcoef(a, b)` and gives you the correct correlation coefficient:

```
>>>
corr = np.corrcoef(a, b)[0, 1]
print(corr)
>>>
0.662573882203
```

Now lets use the stats method of SciPy to compute the correlation coefficient:

```
scipy.stats.pearsonr(x, y)
```

This gives you a slightly different result with NumPy.

Interquartile, range, and quartiles

The **interquartile range** (**IQR**) is a measure of variability, based on dividing a dataset into quartiles. The quartiles divide a rank-ordered dataset into four equal parts. The values that separate parts are called the first, second, and third quartiles. These three are commonly denoted by Q1, Q2, and Q3, respectively.

Now, to show an example of each, let's define data points having the following numbers:

[35, 56, 43, 59, 63, 79, 35, 41, 64, 43, 93, 60, 77, 24, 82]

Now, for these data points, we can calculate the range as follows:

```
import numpy as np
from statistics import variance, stdev
data_points = np.array([35, 56, 43, 59, 63, 79, 35, 41, 64, 43, 93,
60, 77, 24, 82])
dt_rng = np.max(data_points, axis=0) - np.min(data_points, axis=0)
print ("Range:", dt_rng)
>>>
Range: 69
```

Then, for the same data points, we can further calculate the percentiles as follows:

```
print("Quantiles:")
for val in [20, 80, 100]:
    qntls = np.percentile(data_points,val)
print(str(val)+"%", qntls)
>>>
Quantiles:
20% 39.8
80% 77.4
100% 93.0
```

Finally, the IQR can be computed as follows:

```
q75, q25 = np.percentile(data_points, [75, 25])
print("Inter quartile range:", q75-q25)
>>>
Inter quartile range: 28.5
```

Hypothesis testing

A hypothesis test is based on the assumption and tries to determine whether the assumption is true for the entire population. However, in most of the cases, it is not possible to examine the entire population. In that case, a sample is taken from the population and a hypothesis test is carried out. If the hypothesis is true for the sample data, then it is accepted, or else it is rejected.

Hypothesis tests consider two hypotheses, null hypothesis and alternative hypothesis:

- The null hypotheses are generally accepted facts, such as the probability of getting heads or tails of a coin flip is 0.5.
- An alternative hypothesis is a hypothesis we want to prove. For example, we want to find out if a coin is biased or not.

Usually, we test a null hypothesis, and if the null hypothesis is false, then we accept the alternative hypothesis; if the null hypothesis is true then we reject alternative hypothesis. For the coin flip example we saw earlier, if a coin is flipped 100 times and the result is heads 75 times and tails 25 times, then we can safely say that null hypothesis is not true, that is, the probability of getting heads is 0.75, not 0.5. Hence, our alternative hypothesis that the coin is biased towards heads is true.

Hypothesis tests are important in statistics because they give better insights into data. Using this test, we can determine the statistical significance of data. It is used in drug tests to find out effects or if some information is present in some data, and so on. Now, to test this hypothesis, we can use a chi-square test, which is discussed in the next section.

Chi-square tests

A Chi-square test is the measure of the relationship between categorical and numerical data. There are two types of statistical variables, numerical and categorical variables. Consider an example where we have the following data table:

Variable	**Beach**	**Cruise**
Men	209	280
Women	225	248

Which holiday do you prefer? Does the gender affect the preferred holiday? Now, if the gender (man or woman) does affect the preferred holiday, we say they are dependent. But how to determine the dependency? Well, we come up with a "p" value, which is the probability the variables are independent.

Now, to calculate the p-value, we use the Chi-square test. However, this test only works for categorical data, such as Gender {Men, Women}, not numerical data, such as height or weight. Also, the numbers must be large enough so that each entry must be >=5. In our example, we have values such as `209`, `280`, `225`, and `248`, which is good:

```
from scipy.stats import chisquare
x = [209, 280, 225, 248]
chi_statistic, p_value = chisquare(x)
print(chi_statistic)
print(p_value)
>>>
11.846153846153847
0.00792918984728097
```

Now, $p < 0.05$ is the usual test for dependence. In this case, p is less than 0.05, so we believe the variables are dependent (that is, linked together). In other words, men and women probably have the same preference for beach holidays or cruises.

Chi-square independence test

Suppose we have some survey dataset that tries to check if the smoking habit has any dependency with the sex. Now we would like to test H0, to check if the two attributes **Smoke** and **Sex** are independent, versus H1, that the two attributes **Smoke** and **Sex** are dependent. For this type of problem, we can use the chi-square independence test:

Sex	Wr.Hnd	NW.Hnd	W.Hnd	Fold	Pulse	Clap	Exer	Smoke	Height	M.I	Age
Female	18.5	18	Right	R on L	92	Left	Some	Never	173	Metric	18.25
Male	19.5	20.5	Left	R on L	104	Left	None	Regul	177.8	Imperial	17.583
Male	18	13.3	Right	L on R	87	Neither	None	Occas	NA	NA	16.917
Male	18.8	18.9	Right	R on L	NA	Neither	None	Never	160	Metric	20.333
Male	20	20	Right	Neither	35	Right	Some	Never	165	Metric	23.667
Female	18	17.7	Right	L on R	64	Right	Some	Never	172.72	Imperial	21

Figure 7: A snapshot of the survey dataset on smoking habit of men versus women

Now, to do this, we can use the Scipy package. At first, import the required packages as follows:

```
import pandas as pd
from scipy import stats
```

Then, load the dataset from `https://github.com/jupyter/docker-demo-images/blob/master/datasets/MASS/survey.csv` as follows:

```
survey = pd.read_csv("survey.csv")
```

Now, it's time to tabulate, the two variables with row and column variables respectively:

```
survey_tab = pd.crosstab(survey.Smoke, survey.Exer, margins = True)
```

Then, let's create the observed table for analysis as follows:

```
observed = survey_tab.ix[0:4,0:3]
contg = stats.chi2_contingency(observed= observed)
p_value = round(contg[1],3)
print ("P-value is: ",p_value)

>>>
P-value is:  0.483
```

This means `P-value` is very large (that is, > 0.05), which indicates weak evidence against the null hypothesis, so we fail to reject the null hypothesis, that is, the two attributes `Smoke` and `Sex` are independent.

Basic probability for predictive modeling

Probability can define how likely it is that an event is about to happen. When a coin is tossed, there are two possible outcomes - heads (H) or tails (T). In short, the probability of an *event to happen = Number of ways it can happen / Total number of outcomes*.

Thus, we can say that the probability of the coin landing H is ½. On the other hand, the probability of the coin landing T is ½.

Now that using this kind of simple probability measure is not enough in real-life predictive modeling rather other probabilistic concepts are also necessary to know. In this section, we will discuss some of them, with suitable examples.

Probability and the random variables

Random variables make working with probabilities much neater and easier. A random variable in probability is most commonly denoted by a capital X, and the small letter x is then used to ascribe a value to the random variable.

Random variables are possible outcomes of random phenomena. For example, the outcome of a coin flip is either heads or tails, but it is completely random whether it will be heads or tails for unbiased coin.

Random variables are of two types, discrete and continuous. Discrete random variables consider the countable discrete events, such as coin flips. We can flip coins one, two, or three times, but we can't flip a coin 1.3 times or 1.5 times. Continuous random variables consider continuous events such as the height and weight of people. Values could be real numbers. The continuous random variable is defined as an interval in a sample space.

Generating random numbers and setting the seed

When you need to generate random real numbers in a range with equal probability, you can draw numbers from a uniform distribution. We have already seen how to use the random package from Python to generate random numbers:

```
import random
print([random.uniform(0, 10) for x in range(3)])
>>>
[3.757692173482, 1.9534374975799496, 9.926908574009598]
```

However, the result of a random process can differ from one run to the next. Let's rerun the preceding script once again:

```
print([random.uniform(0, 10) for x in range(3)])
>>>
[9.85369119203387, 8.132435135099835, 1.7607189174594984]
```

This type of randomness that varies the result in each run is not desirable, especially when the reproducibility is a concern. Thus, you can run into problems when you use randomization. To resolve this, set the random seed with `random.seed()`:

```
random.seed(12345)
print([random.uniform(0, 10) for x in range(3)])
>>>
[4.166198725453412, 0.10169169457068361, 8.252065092537432]
```

Now, if you execute the following line again, you should expect the same result:

```
random.seed(12345)
print([random.uniform(0, 10) for x in range(3)])

>>>
[4.166198725453412, 0.10169169457068361, 8.252065092537432]
```

Probability distributions

Probability distributions are mathematical functions that represent the probability of different events in a sample space. For a given random variable, probability distribution provides all the possible outcomes of that variable in an experiment. For example, consider a basket with three black and three white balls. If a random variable *X* represents the outcome of taking a ball from the basket, then the probability distribution of *X* is given by the probability of *X* being a white ball as *3/6=0.5* and probability of *X* being a black ball as *3/6=0.5*. The sum of the probability distribution is always *1*. In our example, the total probability of black ball and white ball is *0.5 + 0.5 = 1*.

Probability distributions are of two types, discrete probability distributions and continuous probability distributions:

- The outcome of discrete probability distributions is discrete, such as the preceding example and a coin flip. It is also referred to as **probability mass functions** (**PMF**).
- The outcome of continuous probability distributions is real-valued. It is also referred to as **probability density functions** (**PDF**).

Probability functions PMF and PDF are central to most statistical models. They give a general idea over a complete population by taking a subset of the population. For example, if we want to measure the average height of people of a country, instead of measuring the height of each one of them, we can randomly select some people and take an average of the findings.

This will give us almost the same result as if we measured the height of each individual. Probability distribution tries to create distributions, as accurately as possible, of a subset of a sample space, instead of the whole sample space—because most of the time we don't have data of the whole sample space available.

Marginal probability

Marginal probability is the probability of an event without considering other events. In other words, the marginal probability is the unconditional probability, for example, the probability of getting heads in a coin flip event.

This is unconditional probability because we are interested in the probability of getting heads irrespective of any other events, that is, we did not consider any other events while measuring the probability of the result of a coin flip:

	Head	**Tail**
Head	0.5	0.5
Tail	0.5	0.5

The preceding table is the joint probability of heads and tails in the coin flip event. Marginal probability in the table is a row or column.

Conditional probability

Conditional probability is the opposite of marginal probability. It is the probability of an event under the condition of another event *B*, written as *P (A | B)*. For example, if there are five white balls in a basket and five black balls in the same basket, then there is a 50% probability that a ball is picked randomly and that it happens to be either black or white. If it is white, then there are four white balls in the basket and five black.

Now, if we pick another ball randomly, then the probability of that ball being white is 5/9 and the probability of it being black is 4/9. We can see that the probability of the second event depends on the first event. If the events are independent, then the conditional probability of $P(A \mid B)$ is the same as $P(A)$, that is $P(A \mid B) = P(A)$.

The chain rule of conditional probability

The chain rule is the generalization of the product rule. The product rule states that the joint probability of two events $P(A, B)$ is equal to the conditional probability of one event given another $P(A \mid B)$, and the probability of the conditioned event $P(B)$ and it is given by:

$$P(A, B) = P(A \mid B) * P(B) = P(B \mid A) * P(A)$$

For three events A, B, and C, the conditional probability is given by:

$$P(A,B,C) = P(A \mid B,C)\, P(B,C) = P(A \mid B,C)\, P(B \mid C)\, P(C)$$

For n events, this can be generalized as follows:

$$P(A_1, A_2, ..., A_n) = P(A_1 \mid A_2, ..., A_n)\, P(A_2 \mid A_3, ..., A_n)\, P(A_{n-1} \mid A_n)\, P(A_n)$$

This is referred to as the chain rule. The chain rule is important when we know the probability of each event and we want to calculate the joint probability of these events. This can be better understood with the following Venn diagram. Suppose we know the probability of events A, B is $P(A)$, $P(B)$ respectively, and we are interested in the probability of the intersection of A and B, $P(A, B)$:

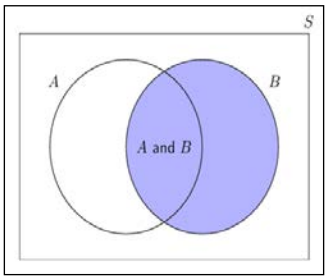


Figure 8: Chain rule conditional probability

This is given by $P(A, B) = P(A \mid B)\, P(B) = P(B \mid A)\, P(A)$.

Independence and conditional independence

Independence and conditional independence are two different types of probability distributions, where the first one means the probability of an event *A* does not depend on another event *B* and is given as follows:

$P(A \mid B) = P(A)$

$P(A \cap B) = P(A).P(B)$

And conditional independence means the probability of two events *A* and *B* are conditionally independent given another event *C*, that is, given that event *C* occurred, the probability of event *A* does not depend on event *B*; it is given as follows:

$P(A \mid B, C) = P(A \mid C)$

For example, if we toss two different coins *A* and *B* one after another, then the result of *A* does not affect the result of *B*, that is, $P(B \mid A) = P(B)$. That is, the probability of *B* is still the same, irrespective of whether the result of event *A* was heads or tails. However, if we toss the same coin twice and the result is *A* and *B*, and another event *C* tells us that the coin is biased toward heads, then the probability of event *A* being heads is more likely than tails, and we can say this because we know from the event *C* that the coin is biased toward heads.

Similarly, the probability of event *B* being heads is also dependent on event *C*. On the other hand, although both events *A* and *B* are from the same coin, they are conditionally independent given the event *C*, that is $P(A \mid B, C) = P(A \mid C)$.

Bayes' rule

Bayes' rule states that the probability of an event *A* happening given the probability of a known event *B* is given as follows:

$$P(A \mid B) = \frac{P(A)P(B \mid A)}{P(B)}$$

In other words, Bayes' rule describes how to update probabilities of an event when the probability of another event is known. For example, consider a cancer test which can correctly identify whether a person has cancer 90% of the time and it can correctly identify if a person does not have any cancer 95% of the time.

Now, if 2% people have cancer and the test gives a positive result, then the probability of that person really having cancer can be calculated with the Bayes' rule as follows:

$$P(A|B) = P(A)\,P(B|A) / P(B)$$

$$= P(A|B) = P(A)\,P(B|A) / (P(B|A)\,P(A) + P(B|\sim A)\,P(\sim A))$$

$$= (0.9 * 0.02) / (0.9 * 0.02 + 0.05 * 0.98)$$

$$= 26.86\%$$

Let's see another example. As you probably know, a well-known Harvard study shows that only 10% of happy people are rich. However, you might think that this statistic is very compelling but you might be somewhat interested in knowing the percentage of rich people that are also really happy. Bayes' theorem helps you out on how to calculate this reserving statistic using two additional clues:

1. The overall percentage of people who are happy, that is, *P(A)*.
2. The overall percentage of people who are rich, that is, *P(B)*.

The key idea behind Bayes' theorem is reversing the statistic considering the overall rates. Suppose that the following pieces of information are available as a prior:

1. 40% of people are happy, that is, *P(A)*
2. 5% of people are rich, that is, *P(B)*

Now, let's consider that the Harvard study is correct, that is, *P (B|A) = 10%*. Now, the fraction of rich people who are happy, that is, *P (A | B)*, can be calculated as follows:

$$P(A|B) = \{P(A)^* P(B|A)\}/ P(B) = (40\%*10\%)/5\% = 80\%$$

Consequently, a majority of the people are also happy! Nice. To make it clearer, let's assume the population of the whole world is 1,000 for simplicity. Then, according to our calculation, there are two facts that exist:

1. **Fact 1**: This tells us 400 people are happy, and the Harvard study tells us that 40 of these happy people are also rich.
2. **Fact 2**: There are 50 rich people altogether, and so the fraction that is happy is *40/50 = 80%*.

Using information theory in predictive modeling

Information theory is a branch of mathematics, but it is commonly used in other fields, such as communication engineering, medical science, and psychology. It was originally proposed by Claude E. Shannon in 1948 to find fundamental limits on signal processing and communication operations such as data compression, in a landmark paper entitled *A Mathematical Theory of Communication*. In that paper, he states:

> "*The fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point*"
>
> *–Claude Shannon, 1948*

Thus, information theory deals with data transmission, storage, and retrieval. However, it can have significant contribution in mathematical or statistical machine learning models. Consequently, it can be used while developing predictive modeling, it can be used extensively too.

Before we dive into information theory, some background knowledge is necessary. Fortunately, we already have the knowledge about the central limit theory, basic probability, random sampling, mean/median/mode and expected value, and so on. We also know about the random variables and random sampling. By random variable, we mean finite discrete random variables, that is, random variables which take only a finite number of values.

Now, mathematically defining entropy, we can say that it is a key measure in information theory that quantifies the amount of uncertainty involved in the value of a random variable or the outcome of a random process; knowing "information" is a prior requirement. Now, let's assume X, Y, and Z are always discrete random variables; p, q, and r are finite discrete distributions; and x, y, and z are values taken by the variables X, Y, and Z respectively. Now we can define self-information also simply called the information, and mutual information.

Self-information

Now, if X follows distribution p, the self-information, of x can be defined as follows:

$$I(x) = \log_2 \frac{1}{p(x)} = -\log_2 p(x)$$

The preceding formula signifies that values with high probability have low self-information. On the other hand, values with low probability have high self-information. For the simplicity, let's call it just information instead of the amount of information.

Mutual information

The mutual information of X relative to Y can be defined as follows:

$$\boldsymbol{I}(\boldsymbol{X};\boldsymbol{Y}) = \mathbb{E}_{\boldsymbol{X},\boldsymbol{Y}}\left[\boldsymbol{SI}(\boldsymbol{x},\boldsymbol{y})\right] = \sum_{\boldsymbol{x},\boldsymbol{y}} \boldsymbol{p}(\boldsymbol{x},\boldsymbol{y}) \log \frac{\boldsymbol{p}(\boldsymbol{x},\boldsymbol{y})}{\boldsymbol{p}(\boldsymbol{x})\,\boldsymbol{p}(\boldsymbol{y})}$$

Mutual information which is used in digital and analog communication is the measure of the amount of information that can be obtained with respect to a random variable by observing another.

Entropy

Entropy is a function that assigns single, real numbers to finite discrete distributions. The entropy is the expected amount of information contained in a random variable:

$$H(X) = \mathbb{E}_X\left[I(X)\right] = -\sum_x p(x) \log_2 p(x)$$

A variable may assume values $x_1, x_2 \ldots, x_n$ and each value may have different self-information, so it makes sense to summarize the information contained in all the values with the mean. To give a concrete example, identifying the outcome of a fair coin flip provides less information (that is, lower entropy) than specifying the outcome from a roll of a dice. Now that we know what entropy is, it's time to see other types of entropies.

Shannon entropy

The Shannon entropy H of an information source is expressed in units of bits and is given as follows:

$$\boldsymbol{H} = -\sum_i \boldsymbol{p}_i \log_2(\boldsymbol{p}_i)$$

$H = -\sum i\, p_i\, log_2\, (p_i)$ {\displaystyle H=-\sum _{i}p_{i}\log _{2}(p_{i})}, Here, p_i is the probability of occurrence of the *ith* possible value of the source symbol. The preceding formula is based on the probability mass function (see the previous section to know more about this).

Joint entropy

The joint entropy of two discrete random variables *X* and *Y* is merely the entropy of their pairing: (*X*, *Y*), which implies that if *X* and *Y* are independent, then their joint entropy is the sum of their individual entropies. This can be defined mathematically as follows:

$$\boldsymbol{H}\left(\boldsymbol{X},\boldsymbol{Y}\right)=\mathbb{E}_{X,Y}\left[-\log \boldsymbol{p}\left(\boldsymbol{x},\boldsymbol{y}\right)\right]=-\sum_{\boldsymbol{x},\boldsymbol{y}}\boldsymbol{p}\left(\boldsymbol{x},\boldsymbol{y}\right)\log \boldsymbol{p}\left(\boldsymbol{x},\boldsymbol{y}\right)$$

For example, if (*X*, *Y*) represents the position of a chess piece where *X* is the row and *Y* is the column, then the joint entropy of the row of the piece and the column of the piece will be the entropy of the position of the piece.

Conditional entropy

The conditional entropy, that is the conditional uncertainty of *X* given random variable *Y* is the average conditional entropy over *Y*, which can be defined as follows:

$$\boldsymbol{H}\left(\boldsymbol{X}\mid\boldsymbol{Y}\right)=\mathbb{E}_{\boldsymbol{Y}}\left[\boldsymbol{H}\left(\boldsymbol{X}\mid\boldsymbol{y}\right)\right]=-\sum_{\boldsymbol{y}\in\boldsymbol{Y}}\boldsymbol{p}\left(\boldsymbol{y}\right)\sum_{\boldsymbol{x}\in\boldsymbol{Y}}\boldsymbol{p}\left(\boldsymbol{x}\mid\boldsymbol{y}\right)\log \boldsymbol{p}\left(\boldsymbol{x}\mid\boldsymbol{y}\right)=-\sum_{x,y}p\left(x,y\right)\log p\left(x\mid y\right)$$

Information gain

Information divergence, or information gain, is a way of comparing two distributions probability say true probability distribution, that is *p(X)*, and an arbitrary probability distribution *q(X)*.

Now, we compress data so that the *q(X)* is the distribution underlying some data, when, in reality, *p(X)* is the correct distribution. In this type of situation, using Kullback–Leibler divergence gives an average additional bit per datum necessary for compression. This way, we can gain some information that can be defined mathematically as follows:

$$D_{KL}\left(p(\boldsymbol{X})\|q(\boldsymbol{X})\right)=\sum_{x\in X}-p(x)\log q(x)-\sum_{x\in X}-p(x)\log q(x)=\sum_{x\in X}-p(x)\log\frac{p(x)}{q(x)}$$

Not only in communication, this is widely used in machine learning ensembles and some other techniques. In the next subsection, we will see some examples of using information theory.

Using information theory

Here are some of the use cases where it is already in use or can be applied:

- In statistics, probability theory deals with uncertain events and information theory is the measure of the uncertainty of probability distribution.
- A direct application of an information theory measure to ML is through cross entropy, which is used as a loss function in the deep neural network-based predictive analytics and machine learning algorithms.
- Then, in decision trees and any rule-based learning algorithm, entropy or information gain can be used to decide the best split to apply or the best rule to use at each level. The idea is to maximize separation, which is the same as minimizing entropy.
- An important topic called information gain is used in tree ensemble technique. For example, in the random forest, we can set the impurity criterion used for information gain calculation. To be more specific, gini or entropy can be set while building the tree-based classifier.
- The minimum description length principle that states that we should describe data in the most succinct possible way is sometimes used to determine the ML model's complexity and overfitting. If the model is too complex for the amount of data we have, then we may be overfitting; this means we are not using the model that minimizes description length.

Nowadays, information-theoretic based concepts have been used widely to characterize processes in dynamic social networks and social media. For instance, the information-theoretic approach is used to classify user activity on Twitter as shown here:

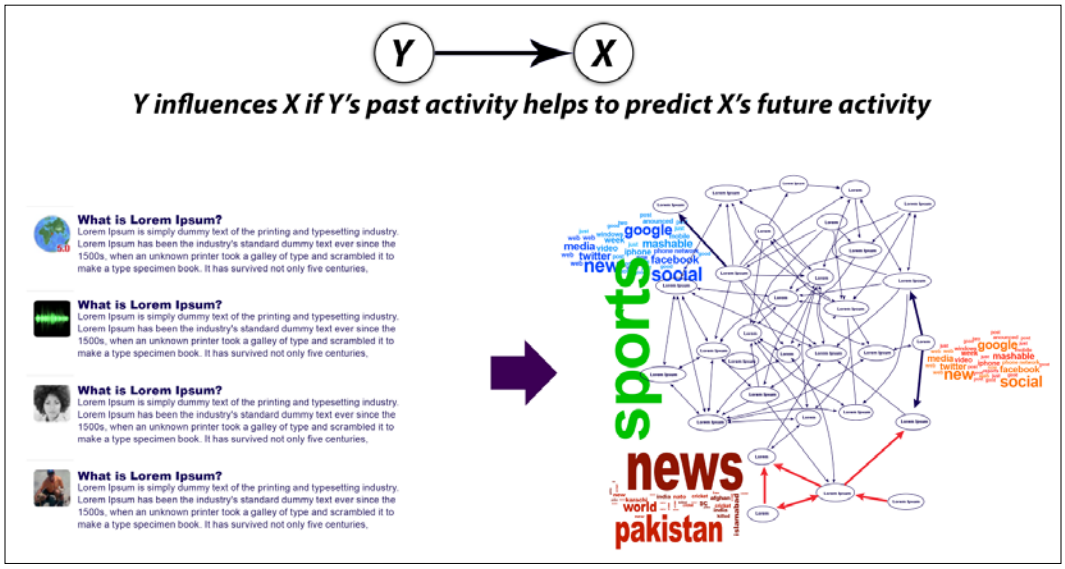


Figure 9: Using information theory to predict someone else's future activities based on past activities

In the preceding figure, the past activities that have been reported in tweets help to predict someone else's future activities. There are other use cases that we will see in the upcoming chapters. Now, we will see how to use information theory and its related features in Python.

Using information theory in Python

Python provides a module called the `dit` module for information theory. It can be downloaded using Python `pip` as follows:

```
$ sudo pip3 install dit
```

However, the following packages have to be installed before using this package:

- Python 2.7, 3.3, 3.4, 3.5, or 3.6 Numpy
- iterutils
- six
- contextlib2

- prettytable
- networkx
- cython
- cvxopt
- numdifftools
- scipy

Now, let's see some example, of using this module. At first, let's import `dit` as follows:

```
import dit
```

Now, suppose we have a really thick coin, one so thick that there is a reasonable chance of it landing on its edge. Here is how we might represent the coin in `dit`:

```
d = dit.Distribution(['H', 'T', 'E'], [.4, .4, .2])
print(d)
>>>
Class:          Distribution
Alphabet:       ('E', 'H', 'T') for all rvs
Base:           linear
Outcome Class:  str
Outcome Length: 1
RV Names:       None
x   p(x)
E   0.2
H   0.4
T   0.4
```

Let's calculate the probability of `H`:

```
print(d['H'])
>>>
0.4
```

We can calculate the probability of the combination of *H* or *T*:

```
print(d.event_probability(['H','T']))
>>>
0.8
```

Calculate the Shannon entropy of the joint distribution:

```
entropy = dit.shannon.entropy(d)
print(entropy)
>>>
1.52192809489
```

Calculate `extropy` of the joint distribution:

```
extropy = dit.other.extropy(d)
print(extropy)
>>>
1.14190118891
```

Now, let's create a distribution, where `Z = xor(X, Y)`:

```
import dit.example_dists
e = dit.example_dists.Xor()
e.set_rv_names(['X', 'Y', 'Z'])
print(e)

>>>
Class:          Distribution
Alphabet:       ('0', '1') for all rvs
Base:           linear
Outcome Class:  str
Outcome Length: 3
RV Names:       ('X', 'Y', 'Z')
x     p(x)
000   0.25
011   0.25
101   0.25
110   0.25
```

Now, calculate the Shannon mutual information `I[X:Z]`:

```
xz = dit.shannon.mutual_information(e, [,X'], [,Z'])
print(xz)
>>>
0.0
```

Let's calculate the Shannon mutual information `I[Y:Z]`:

```
yz = dit.shannon.mutual_information(e, ['Y'], ['Z'])
print(yz)
>>>
0.0
```

Let's calculate the Shannon mutual information `I[X,Y:Z]`:

```
xyz = dit.shannon.mutual_information(e, ['X', 'Y'], ['Z'])
print(xyz)
>>>
1.0
```

For the time being, this is enough I guess. We will keep learning in the upcoming chapters.

Summary

In this chapter, we covered some statistical concepts before getting started with predictive analytics. Some examples are random sampling, hypothesis testing, the chi-square test, correlation, expectation, variance, covariance and Bayes' rule, and so on. In the second part of this chapter, we discussed probability and information theory for predictive analytics. The central objects of probability theory are random variables, stochastic processes, and events, which are also discussed in this chapter.

We have provided some theoretical aspects. However, predictive models are models of the relation between the specific performance of a unit in a sample and one or more known attributes and features of the unit. The objective of the model is to assess the likelihood that a similar unit in a different sample will exhibit the specific performance.

The next chapter describes the main TensorFlow capabilities, motivated by a real-life Titanic example. The second part of the chapter will cover some introductory aspects of TensorFlow. We will go through the installation of TensorFlow for both CPUs and GPUs. Also, we will explain the main computational concepts behind TensorFlow and how to get you on track by implementing linear regression and logistic regression.

3
From Data to Decisions – Getting Started with TensorFlow

Despite the huge availability of data and significant investments, many business organizations still go on gut feel because they neither make the proper use of the data nor do they take appropriate and effective business decisions. TensorFlow, on the other hand, can be used to help take the business decision from this huge collection of data. TensorFlow is mathematical software and an open source software library for Machine Intelligence, developed in 2011 by the Google Brain Team and it can be used to help us analyze data to predict the effective business outcome. Although the initial target of TensorFlow was to conduct research in machine learning and in deep neural networks, however, the system is general enough to be applicable in a wide variety of other domains as well.

Keeping in mind your needs and based on all the latest and exciting features of TensorFlow 1.x, in this chapter, we will give a description of the main TensorFlow capabilities that are mostly motivated by a real-life example using the data.

In summary, the following topics will be discussed in this chapter:

- From data to decision: Titanic example
- General overview of TensorFlow
- Installing and configuring TensorFlow
- TensorFlow computational graph
- TensorFlow programming model

- TensorFlow data model
- Visualizing through TensorBoard
- Getting started with TensorFlow: Linear regression and beyond

Taking decisions based on data - Titanic example

The growing demand for data is a key challenge. Decision support teams such as institutional research and business intelligence often cannot take the right decisions on how to expand their business and research outcomes from a huge collection of data. Although data plays an important role in driving the decision, however, in reality, taking the right decision at right time is the goal.

In other words, the goal is the decision support, not the data support. This can be achieved through an advanced use of data management and analytics.

Data value chain for making decisions

The following diagram in figure 1 (source: *H. Gilbert Miller and Peter Mork, From Data to Decisions: A Value Chain for Big Data, Proc. Of IT Professional, Volume: 15, Issue: 1, Jan.-Feb. 2013, DOI: 10.1109/MITP.2013.11*) shows the data chain towards taking actual decisions–that is, the goal. The value chains start through the data discovery stage consisting of several steps such as data collection and annotating data preparation, and then organizing them in a logical order having the desired flow. Then comes the data integration for establishing a common data representation of the data. Since the target is to take the right decision, for future reference having the appropriate provenance of the data–that is, where it comes from, is important:

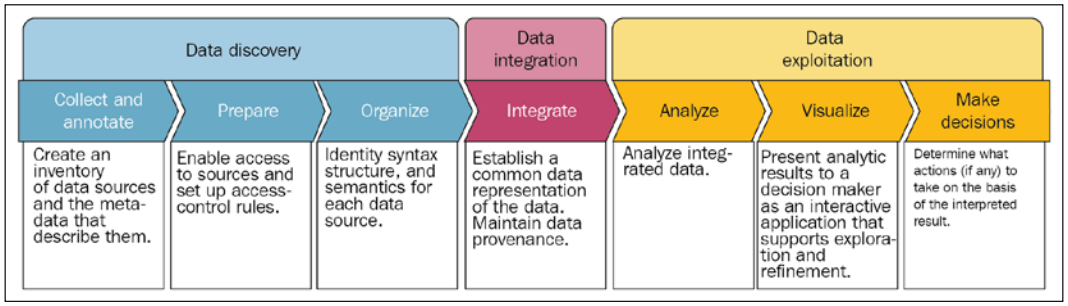


Figure 1: From Data to Decisions: A Value Chain for Big Data

Well, now your data is somehow integrated into a presentable format, it's time for the data exploration stage, which consists of several steps such as analyzing the integrated data and visualization before taking the actions to take on the basis of the interpreted results.

However, is this enough before taking the right decision? Probably not! The reason is that it lacks enough analytics, which eventually helps to take the decision with an actionable insight. Predictive analytics comes in here to fill the gap between. Now let's see an example of how in the following section.

From disaster to decision – Titanic survival example

Here is the challenge, Titanic–Machine Learning from Disaster from Kaggle (`https://www.kaggle.com/c/titanic`):

> *"The sinking of the RMS Titanic is one of the most infamous shipwrecks in history. On April 15, 1912, during her maiden voyage, the Titanic sank after colliding with an iceberg, killing 1502 out of 2224 passengers and crew. This sensational tragedy shocked the international community and led to better safety regulations for ships. One of the reasons that the shipwreck led to such loss of life was that there were not enough lifeboats for the passengers and crew. Although there was some element of luck involved in surviving the sinking, some groups of people were more likely to survive than others, such as women, children, and the upper-class. In this challenge, we ask you to complete the analysis of what sorts of people were likely to survive. In particular, we ask you to apply the tools of machine learning to predict which passengers survived the tragedy"*

But going into this deeper, we need to know about the data of passengers travelling in the Titanic during the disaster so that we can develop a predictive model that can be used for survival analysis. The dataset can be downloaded from the preceding URL. Table 1 here shows the metadata about the Titanic survival dataset:

Variable	Definition
survival	Two labels: 0 = No 1 = Yes
pclass	This is a proxy for the **Socioeconomic Status** (**SES**) of a passenger that is categorized as upper, middle and lower. In particular, 1 = 1st, 2 = 2nd, 3 = 3rd
sex	Male or female

Variable	Definition
Age	Age in years
sibsp	This signifies the family relation as follows: Sibling = brother, sister, stepbrother, stepsister Spouse = husband, wife (mistresses and fiancés were ignored)
parch	In the dataset, family relations are defined as follows: Parent = mother, father Child = daughter, son, stepdaughter, stepson Some children travelled only with a nanny, therefore parch=0 for them
ticket	Ticket number
fare	Passenger ticket fare
cabin	Cabin number
embarked	Three ports: C = Cherbourg Q = Queenstown S = Southampton

A snapshot of the dataset can be seen as follows:

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
1	0	3	Braund, Mr. Owen Harris	male	22	1	0	A/5 21171	7.25		S
2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Thayer)	female	38	1	0	PC 17599	71.2833	C85	C
3	1	3	Heikkinen, Miss. Laina	female	26	0	0	STON/O2. 3101282	7.925		S
4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35	1	0	113803	53.1	C123	S
5	0	3	Allen, Mr. William Henry	male	35	0	0	373450	8.05		S
6	0	3	Moran, Mr. James	male		0	0	330877	8.4583		Q
7	0	1	McCarthy, Mr. Timothy J	male	54	0	0	17463	51.8625	E46	S
8	0	3	Palsson, Master. Gosta Leonard	male	2	3	1	349909	21.075		S
9	1	3	Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)	female	27	0	2	347742	11.1333		S
10	1	2	Nasser, Mrs. Nicholas (Adele Achem)	female	14	1	0	237736	30.0708		C
11	1	3	Sandstrom, Miss. Marguerite Rut	female	4	1	1	PP 9549	16.7	G6	S
12	1	1	Bonnell, Miss. Elizabeth	female	58	0	0	113783	26.55	C103	S
13	0	3	Saundercock, Mr. William Henry	male	20	0	0	A/5. 2151	8.05		S
14	0	3	Andersson, Mr. Anders Johan	male	39	1	5	347082	31.275		S
15	0	3	Vestrom, Miss. Hulda Amanda Adolfina	female	14	0	0	350406	7.8542		S

Figure 2: A snapshot of the Titanic survival dataset

The ultimate target of using this dataset is to predict what kind of people survived the Titanic disaster. However, a bit of exploratory analysis of the dataset is a mandate. At first, we need to import necessary packages and libraries:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
```

Now read the dataset and create a panda's DataFrame:

```
df = pd.read_csv('/home/asif/titanic_data.csv')
```

Before drawing the distribution of the dataset, let's specify the parameters for the graph:

```
fig = plt.figure(figsize=(18,6), dpi=1600)
alpha=alpha_scatterplot = 0.2
alpha_bar_chart = 0.55
fig = plt.figure()
ax = fig.add_subplot(111)
```

Draw a bar diagram for showing who survived versus who did not:

```
ax1 = plt.subplot2grid((2,3),(0,0))
ax1.set_xlim(-1, 2)
df.Survived.value_counts().plot(kind='bar', alpha=alpha_bar_chart)
plt.title("Survival distribution: 1 = survived")
```

Plot a graph showing survival by `Age`:

```
plt.subplot2grid((2,3),(0,1))
plt.scatter(df.Survived, df.Age, alpha=alpha_scatterplot)
plt.ylabel("Age")
plt.grid(b=True, which='major', axis='y')
plt.title("Survival by Age: 1 = survived")
```

Plot a graph showing distribution of the `passengers` classes:

```
ax3 = plt.subplot2grid((2,3),(0,2))
df.Pclass.value_counts().plot(kind="barh", alpha=alpha_bar_chart)
ax3.set_ylim(-1, len(df.Pclass.value_counts()))
plt.title("Class dist. of the passengers")
```

Plot a kernel density estimate of the subset of the 1st class passengers' age:

```
plt.subplot2grid((2,3),(1,0), colspan=2)
df.Age[df.Pclass == 1].plot(kind='kde')
df.Age[df.Pclass == 2].plot(kind='kde')
df.Age[df.Pclass == 3].plot(kind='kde')
plt.xlabel("Age")
plt.title("Age dist. within class")
plt.legend(('1st Class', '2nd Class','3rd Class'),loc='best')
```

Plot a graph showing `passengers per boarding location`:

```
ax5 = plt.subplot2grid((2,3),(1,2))
df.Embarked.value_counts().plot(kind='bar', alpha=alpha_bar_chart)
ax5.set_xlim(-1, len(df.Embarked.value_counts()))
plt.title("Passengers per boarding location")
```

Finally, we show all the subplots together:

```
plt.show()
>>>
```

The figure shows the survival distribution, survival by age, age distribution, and the passengers per boarding location:

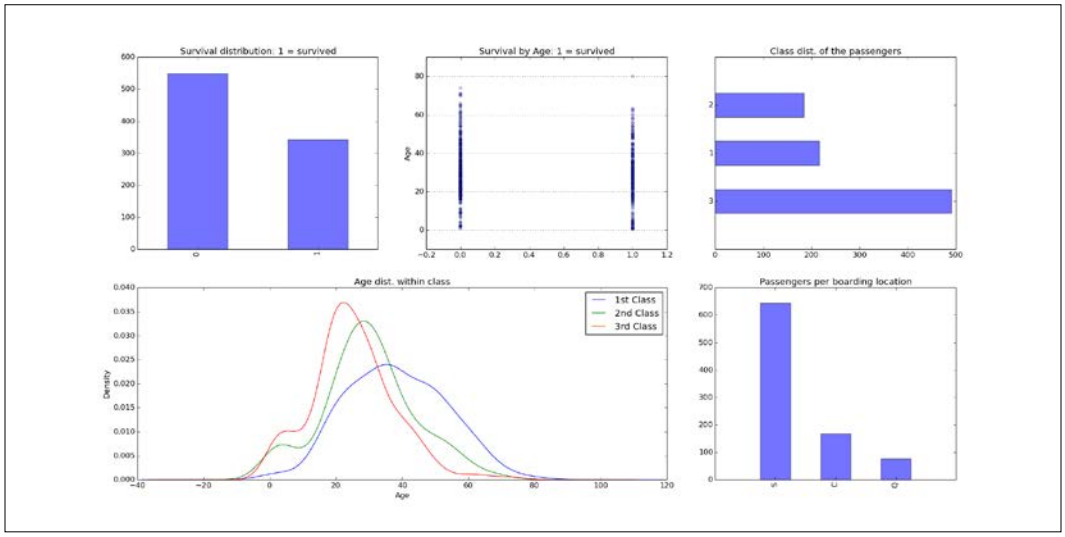


Figure 3: Titanic survival data distribution across age, class, and age within classes and boarding location.

However, to execute the preceding code, you need to install several packages such as matplotlib, pandas, and scipy. They are listed as follows:

- **Installing pandas**: Pandas is a Python package for data manipulation. It can be installed as follows:

```
$ sudo pip3 install pandas
#For Python 2.7, use the following:
$ sudo pip install pandas
```

- **Installing matplotlib**: In the preceding code, matplotlib is a plotting library for mathematical objects. It can be installed as follows:

```
$ sudo apt-get install python-matplotlib   # for Python 2.7
$ sudo apt-get install python3-matplotlib # for Python 3.x
```

- **Installing scipy**: Scipy is a Python package for scientific computing. Installing `blas` and `lapack` and `gfortran` are a prerequisite for this one. Now just execute the following command on your terminal:

```
$ sudo apt-get install libblas-dev liblapack-dev
$ sudo apt-get install gfortran
$ sudo pip3 install scipy # for Python 3.x

$ sudo pip install scipy # for Python 2.7
```

For Mac, use the following command to install the above modules:

```
$ sudo easy_install pip
$ sudo pip install matplotlib
$ sudo pip install libblas-dev liblapack-dev
$ sudo pip install gfortran
$ sudo pip install scipy
```

For windows, I am assuming that Python 2.7 is already installed at C:\Python27. Then open the command prompt and type the following command:

```
C:\Users\admin-karim>cd C:/Python27
C:\Python27> python -m pip install <package_name> # provide package name
accordingly.
```

For Python3, issue the following commands:

```
C:\Users\admin-karim>cd C:\Users\admin-karim\AppData\Local\Programs\
Python\Python35\Scripts
C:\Users\admin-karim\AppData\Local\Programs\Python\Python35\
Scripts>python3 -m pip install <package_name>
```

Well, we have seen the data. Now it's your turn to do some analytics on top of the data. Say predicting what kinds of people survived from that disaster. Don't you agree that we have enough information about the passengers, but how could we do the predictive modeling so that we can draw some fairly straightforward conclusions from this data?

For example, say being a woman, being in 1st class, and being a child were all factors that could boost passenger chances of survival during this disaster.

In a brute-force approach–for example, using if/else statements with some sort of weighted scoring system, you could write a program to predict whether a given passenger would survive the disaster. However, does writing such a program in Python make much sense? Naturally, it would be very tedious to write, difficult to generalize, and would require extensive fine tuning for each variable and samples (that is, passenger).

This is where predictive analytics with machine learning algorithms and emerging tools comes in so that you could build a program that learns from the sample data to predict whether a given passenger would survive. In such cases, we will see throughout this book that TensorFlow could be a perfect solution to achieve outstanding accuracies across your predictive models. We will start describing the general overview of the TensorFlow framework. Then we will show how to install and configure TensorFlow on Linux, Mac OS and Windows.

General overview of TensorFlow

TensorFlow is an open source framework from Google for scientific and numerical computation based on dataflow graphs that stand for the TensorFlow's execution model. The dataflow graphs used in TensorFlow help the machine learning experts to perform more advanced and intensive training on the data for developing deep learning and predictive analytics models. In 2015, Google open sourced the TensorFlow and all of its reference implementation and made all the source code available on GitHub under the Apache 2.0 license. Since then, TensorFlow has achieved wide adoption from academia and research to the industry, and following that recently the most stable version 1.x has been released with a unified API.

As the name TensorFlow implies, operations are performed by neural networks on multidimensional data arrays (aka flow of tensors). This way, TensorFlow provides some widely used and robust implementation linear models and deep learning algorithms.

Deploying a predictive or general purpose model using TensorFlow is pretty straightforward. The thing is that once you have constructed your neural networks model after necessary feature engineering, you can simply perform the training interactively using plotting or TensorBoard (we will see more on it in upcoming sections). Finally, you deploy it eventually after evaluating it by feeding it some test data.

Since we are talking about the dataflow graphs, nodes in a flow graph correspond to the mathematical operations, such as addition, multiplication, matrix factorization, and so on, whereas, edges correspond to tensors that ensure communication between edges and nodes, that is dataflow and controlflow.

You can perform the numerical computation on a CPU. Nevertheless, using TensorFlow, it is also possible to distribute the training across multiple devices on the same system and train on them, especially if you have more than one GPU on your system so that these can share the computational load. But the precondition is if TensorFlow can access these devices, it will automatically distribute the computations to the multiple devices via a greedy process. But TensorFlow also allows the program, to specify which operations will be on which devices via name scope placement.

The APIs in TensorFlow 1.x have changed in ways that are not all backward compatible. That is, TensorFlow programs that worked on TensorFlow 0.x won't necessarily work on TensorFlow 1.x.

The main features offered by the latest release of TensorFlow are:

- **Faster computing**: The latest release of TensorFlow is incredibly faster. For example, it is 7.3 times faster on 8 GPUs for Inception v3 and 58 times speedup for distributed inception (v3 training on 64 GPUs).
- **Flexibility**: TensorFlow is not just a deep learning library, but it comes with almost everything you need for powerful mathematical operations through functions for solving the most difficult problems. TensorFlow 1.x introduces some high-level APIs for high-dimensional arrays or tensors, with `tf.layers`, `tf.metrics`, `tf.losses`, and `tf.keras` modules. These have made TensorFlow very suitable for high-level neural networks computing.
- **Portability**: TensorFlow runs on Windows, Linux, and Mac machines and on mobile computing platforms (that is, Android).
- **Easy debugging**: TensorFlow provides the TensorBoard tool for the analysis of the developed models.
- **Unified API**: TensorFlow offers you a very flexible architecture that enables you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API.

- **Transparent use of GPU computing**: Automating management and optimization of the same memory and the data used. You can now use your machine for large-scale and data-intensive GPU computing with NVIDIA cuDNN and CUDA toolkits.
- **Easy use**: TensorFlow is for everyone, it's for students, researchers, deep learning practitioners, and also for readers of this book.
- **Production ready at scale**: Recently it has evolved as the neural network for machine translation, at production scale. TensorFlow 1.x promises Python API stability, making it easier to choose new features without worrying too much about breaking your existing code.
- **Extensibility**: TensorFlow is relatively newer technology and it's still under active development. However, it is extensible because it was released with source code available on GitHub (`https://github.com/tensorflow/tensorflow`).
- **Supported**: There is a large community of developers and users working together to make TensorFlow a better product, both by providing feedback and by actively contributing to the source code.
- **Wide adoption**: Numerous tech giants are using TensorFlow for increasing their business intelligence. For example, ARM, Google, Intel, eBay, Qualcomm, SAM, Drobox, DeepMind, Airbnb, Twitter, and so on.

Throughout the next chapters, we will see how to achieve these features for predictive analytics.

Installing and configuring TensorFlow

You can install and use TensorFlow on a number of platforms such as Linux, Mac OS, and Windows. Moreover, you can also build and install TensorFlow from the latest GitHub source of TensorFlow. Furthermore, if you have a Windows machine, you can install TensorFlow via native pip or Anacondas. It is to be noted that TensorFlow supports Python 3.5.x and 3.6.x on Windows.

Also, Python 3 comes with the pip3 package manager, which is the program you'll use to install TensorFlow. So you don't need to install pip if you're using this Python version. For simplicity, in this section, I will show you how to install TensorFlow using native pip. Now to install TensorFlow, start a terminal. Then issue the appropriate `pip3 install` command in that terminal.

To install the CPU-only version of TensorFlow, enter the following command:

```
C:\> pip3 install --upgrade tensorflow
```

To install the GPU version of TensorFlow, enter the following command:

```
C:\> pip3 install --upgrade tensorflow-gpu
```

When it comes to Linux, the TensorFlow Python API supports Python 2.7 and Python 3.3+, so you need to install Python to start the TensorFlow installation. You must install Cuda Toolkit 7.5 and cuDNN v5.1+ to get the GPU support. In this section, we will show you how to install and get started with TensorFlow. More details on installing TensorFlow on Linux will be shown.



Installing on Mac OS is more or less similar to Linux. Please refer to the `https://www.tensorflow.org/install/install_mac` for more details. On the other hand, Windows users should refer to `https://www.tensorflow.org/install/install_windows`.

Note that for this and the rest of the chapters, we will provide most of the source codes with Python 3.x compatible.

Installing TensorFlow on Linux

In this section, we will show you how to install TensorFlow on Ubuntu 14.04 or higher. The instructions presented here also might be applicable for other Linux distributions with minimal adjustments.

However, before proceeding with formal steps, we need to determine which TensorFlow to install on your platform. TensorFlow has been developed such that you can run data intensive tensor applications on a GPU as well as a CPU. Thus, you should choose one of the following types of TensorFlow to install on your platform:

- **TensorFlow with CPU support only**: If there is no GPU such as NVIDIA® installed on your machine, you must install and start computing using this version. This is very easy and you can do it in just 5 to 10 minutes.
- **TensorFlow with GPU support**: As you might know, a deep learning application requires typically very high intensive computing resources. Thus TensorFlow is no exception, but can typically speed up the data computation and analytics significantly faster on a GPU rather than on a CPU. Therefore, if there's NVIDIA® GPU hardware on your machine, you should ultimately install and use this version.

From our experience, even if you have NVIDIA GPU hardware integrated on your machine, it would be worth installing and trying the CPU-only version first and if you don't experience good performance you should switch for GPU support then.

The GPU-enabled version of TensorFlow has several requirements such as 64-bit Linux, Python 2.7 (or 3.3+ for Python 3), NVIDIA CUDA® 7.5 or higher (CUDA 8.0 required for Pascal GPUs), and NVIDIA cuDNN v4.0 (minimum) or v5.1 (recommended). More specifically, the current development of TensorFlow supports only GPU computing using NVIDIA toolkits and software. Therefore, the following software must have to be installed on your Linux machine to get the GPU support on your predictive analytics applications:

- Python
- NVIDIA Driver
- CUDA with *compute capability >= 3.0*
- CudNN
- TensorFlow

Installing Python and nVidia driver

We have already seen how to install Python on a different platform, so we can skip this one. Also, I'm assuming that your machine already has a NVIDIA GPU installed.

To find out if your GPU is really installed properly and working, issue the following command on the terminal:

```
$ lspci -nnk | grep -i nvidia

# Expected output (of course may vary for your case): 4b:00.0 VGA
compatible controller [0300]: NVIDIA Corporation Device [10de:1b80] (rev
a1)
4b:00.1 Audio device [0403]: NVIDIA Corporation Device [10de:10f0] (rev
a1)
```

Since predictive analytics largely depend on machine learning and deep learning algorithms, make sure you check that some essential packages are installed on your machine such as GCC and some of the scientific Python packages.

Simply issue the following command for doing so on the terminal:

```
$ sudo apt-get update

$ sudo apt-get install libglu1-mesa libxi-dev libxmu-dev -y
$ sudo apt-get — yes install build-essential
$ sudo apt-get install python-pip python-dev -y
$ sudo apt-get install python-numpy python-scipy –y
```

Now download the NVIDIA driver (don't forget to choose the right version for your machine) via `wget` and run the script in silent mode:

```
$ wget http://us.download.nvidia.com/XFree86/Linux-x86_64/367.44/NVIDIA-Linux-x86_64-367.44.run
$ sudo chmod +x NVIDIA-Linux-x86_64-367.35.run
$ ./NVIDIA-Linux-x86_64-367.35.run --silent
```



Some GPU cards such as NVidia GTX 1080 comes with the built in–driver. Thus, if your machine has a different GPU other than the GTX 1080, you have to download the driver for that GPU.

To make sure if the driver was installed correctly, issue the following command on the terminal:

```
$ nvidia-smi
```

The outcome of the command should be as follows:

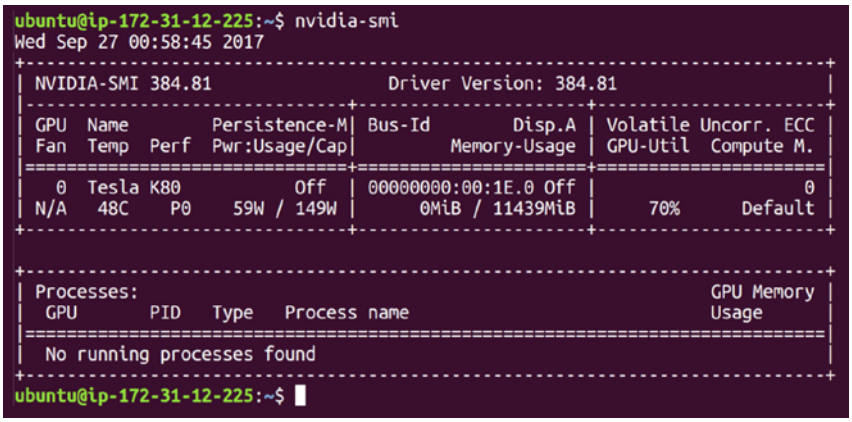


Figure 4: Outcome of the nvidia-smi command

Installing NVIDIA CUDA

To use TensorFlow with NVIDIA GPUs, CUDA® Toolkit 8.0, and associated NVIDIA drivers with CUDA toolkit 8+ are required to be installed. The CUDA toolkit includes:

- GPU-accelerated libraries such as cuFFT for **Fast Fourier Transforms** (**FFT**)
- cuBLAS for **Basic Linear Algebra Subroutines** (**BLAS**)

- cuSPARSE for sparse matrix routines
- cuSOLVER for dense and sparse direct solvers
- cuRAND for random number generation, NPP for image, and video processing primitives
- **nvGRAPH** for **NVIDIA Graph Analytics Library**
- Thrust for template parallel algorithms and data structures and a dedicated CUDA math library

For Linux, download and install required packages:

`https://developer.nvidia.com/cuda-downloads` using `wget` command on Ubuntu as follows:

```
$ wget https://developer.nvidia.com/compute/cuda/8.0/Prod2/local_
installers/cuda_8.0.61_375.26_linux-run

$ sudo chmod +x cuda_8.0.61_375.26_linux.run

$ ./ cuda_8.0.61_375.26_linux.run --driver --silent
$ ./ cuda_8.0.61_375.26_linux.run --toolkit --silent
$ ./ cuda_8.0.61_375.26_linux.run --samples –silent
```

Also, ensure that you have added the CUDA installation path to the `LD_LIBRARY_PATH` environment variable as follows:

```
$ echo 'export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/usr/local/cuda/lib64:/
usr/local/cuda/extras/CUPTI/lib64"' >> ~/.bashrc

$ echo 'export CUDA_HOME=/usr/local/cuda' >> ~/.bashrc
$ source ~/.bashrc
```

Installing NVIDIA cuDNN v5.1+

Once the CUDA Toolkit is installed, you should download the cuDNN v5.1 library from `https://developer.nvidia.com/cudnn` for Linux and once downloaded, uncompress the files and copy them into the CUDA Toolkit directory (assumed here to be in `/usr/local/cuda/`):

```
$ cd /usr/local

$sudo mkdir cuda

$ cd ~/Downloads/

$ wget http://developer2.download.nvidia.com/compute/machine-learning/
cudnn/secure/v6/prod/8.0_20170427/cudnn-8.0-linux-x64-v6.0.tgz

$ sudo tar –xvzf cudnn-8.0-linux-x64-v6.0.tgz

$ cp cuda/lib64/* /usr/local/cuda/lib64/

$ cp cuda/include/cudnn.h /usr/local/cuda/include/
```

Note that to install the cuDNN v5.1 library, you must need to register for the Accelerated Computing Developer Program at `https://developer.nvidia.com/accelerated-computing-developer`. Now when you have installed the cuDNN v5.1 library, ensure that you create the `CUDA_HOME` environment variable.

Installing the libcupti-dev library

Lastly, you need to have the libcupti-dev library installed on your machine. This is the NVIDIA CUDA that provides advanced profiling support. To install this library, issue the following command:

```
$ sudo apt-get install libcupti-dev
```

Installing TensorFlow

Refer to the following section for more step-by-step guidelines on how to install the latest version of TensorFlow for the CPU only and GPU supports with NVIDIA cuDNN and CUDA computing capability. You can install TensorFlow on your machine in a number of ways, such as using virtualenv, pip, Docker, and Anaconda. However, using Docker and Anaconda is a bit advanced and this is why we have decided to use pip and virtualenv instead.



Interested readers can try using Docker and Anaconda from `https://www.tensorflow.org/install/`.

Installing TensorFlow with native pip

If steps 1 to 6 are completed, install TensorFlow by invoking one of the following commands. For Python 2.7 and, of course, with only CPU support:

```
$ pip install tensorflow
# For Python 3.x and of course with only CPU support:
$ pip3 install tensorflow
# For Python 2.7 and of course with GPU support:
$ pip install tensorflow-gpu
# For Python 3.x and of course with GPU support:
$ pip3 install tensorflow-gpu
```

If step 3 failed somehow, install the latest version of TensorFlow by issuing a command manually:

```
$ sudo pip install --upgrade TF_PYTHON_URL
#For Python 3.x, use the following command:
$ sudo pip3 install --upgrade TF_PYTHON_URL
```

For both cases, `TF_PYTHON_URL` signifies the URL of the TensorFlow Python package presented at `https://www.tensorflow.org/install/install_linux#the_url_of_the_tensorflow_python_package`.

For example, to install the latest version with CPU-only support (at the time of writing v1.1.0), use the following command:

```
$ sudo pip3 install --upgrade https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-1.1.0-cp34-cp34m-linux_x86_64.whl
```

Installing with virtualenv

We assume that you already have Python 2+ (or 3+) and pip (or pip3) installed on your system. If so, follow these steps to install TensorFlow:

1. Create a virtualenv environment as follows:

   ```
   $ virtualenv --system-site-packages targetDirectory
   ```

   The `targetDirectory` signifies the root of the virtualenv tree. By default, it is `~/tensorflow` (however, you may choose any directory).

2. Activate virtualenv environment as follows:

   ```
   $ source ~/tensorflow/bin/activate # bash, sh, ksh, or zsh
      $ source ~/tensorflow/bin/activate.csh  # csh or tcsh
   ```

   If the command succeeds in step 2, then you should see the following on your terminal:

   ```
   (tensorflow)$
   ```

3. Installing TensorFlow.

   Follow one of the following commands to install TensorFlow in the active virtualenv environment. For Python 2.7 with CPU-only support, use the following command:

   ```
   (tensorflow)$ pip install --upgrade tensorflow
   #For Python 3.x with CPU support, use the following command:
   (tensorflow)$ pip3 install --upgrade tensorflow
   #For Python 2.7 with GPU support, use the following command:
   (tensorflow)$ pip install --upgrade tensorflow-gpu
   #For Python 3.x with GPU support, use the following command:
    (tensorflow)$ pip3 install --upgrade tensorflow-gpu
   ```

If the preceding command succeeds, skip step 5. If the preceding command fails, perform step 5. Moreover, if step 3 failed somehow, try to install TensorFlow in the active virtualenv environment by issuing a command of the following format:

```
#For python 2.7 (select appropriate URL with CPU or GPU support):
(tensorflow)$ pip install --upgrade TF_PYTHON_URL
#For python 3.x (select appropriate URL with CPU or GPU support):
  (tensorflow)$ pip3 install --upgrade TF_PYTHON_URL
```

4. Validate the installation.

   To validate the installation in step 3, you must activate the virtual environment. If the virtualenv environment is not currently active, issue one of the following commands:

   ```
   $ source ~/tensorflow/bin/activate  # bash, sh, ksh, or zsh
   $ source ~/tensorflow/bin/activate.csh  # csh or tcsh
   ```

5. Uninstalling TensorFlow

   To uninstall TensorFlow, simply remove the tree you created. For example:

   ```
   $ rm -r targetDirectory
   ```

   Finally, if you want to control which devices are visible to TensorFlow manually, you should set the `CUDA_VISIBLE_DEVICES`. For example, the following command can be used to force the use of only GPU 0:

   ```
   $ CUDA_VISIBLE_DEVICES=0 python
   ```

Installing TensorFlow from source

The pip installation can cause problems using TensorBoard (this will be discussed later in this chapter). For this reason, I suggest you build TensorFlow directly from the source. The steps are described as follows.



Follow the instructions and guidelines on how to install Bazel on your platform at `http://bazel.io/docs/install.html`

At first, clone the entire TensorFlow repository as follows:

```
$git clone --recurse-submodules https://github.com/tensorflow/tensorflow
```

Then it's time to install Bazel, which is a tool that automates software builds and tests. Also, for building TensorFlow from source, Bazel build system must be installed on your machine. For this, issue the following command:

```
$ sudo apt-get install software-properties-common swig
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt-get install oracle-java8-installer
$ echo "deb http://storage.googleapis.com/bazel-apt stable jdk1.8" | sudo
tee /etc/apt/sources.list.d/bazel.list
$ curl https://storage.googleapis.com/bazel-apt/doc/apt-key.pub.gpg |
sudo apt-key add -
$ sudo apt-get update
$ sudo apt-get install bazel
```

Then run the Bazel installer by issuing the following command:

```
$ chmod +x bazel-version-installer-os.sh
$ ./bazel-version-installer-os.sh --user
```

Moreover, you might need some Python dependencies such as `python-numpy`, `swig`, and `python-dev`. Now, issue the following command for doing so:

```
$ sudo apt-get install python-numpy swig python-dev
```

Now it's time to configure the installation (GPU or CPU). Let's do it by executing the following command:

```
$ ./configure
```

Then create your TensorFlow package using `bazel`:

```
$ bazel build -c opt //tensorflow/tools/pip_package:
$ build_pip_package
```

However, to build with the GPU support, issue the following command:

```
$ bazel build -c opt --config=cuda //tensorflow/tools/pip_package:build_
pip_package
```

Finally, install TensorFlow. Here I have listed, as per the Python version:

- For Python 2.7:

  ```
  $ sudo pip install --upgrade /tmp/tensorflow_pkg/
  tensorflow-1.1.0-*.whl
  ```

- For Python 3.4:

  ```
  $ sudo pip3 install --upgrade /tmp/tensorflow_pkg/
  tensorflow-1.1.0-*.whl
  ```

Testing your TensorFlow installation

We start with the popular TensorFlow alias `tf`. Open a Python terminal (just type `python` or `python3` on terminal) and issue the following lines of code:

```
>>> import tensorflow as tf
```

If your favourite Python interpreter doesn't complain, then you're ready to start using TensorFlow!

```
>>> hello = tf.constant("Hello, TensorFlow!")
>>> sess=tf.Session()
```

Now to verify your installation just type the following:

```
>>> print sess.run(hello)
```

If the installation is OK, you'll see the following output:

```
Hello, TensorFlow!
```

TensorFlow computational graph

When thinking of execution of a TensorFlow program we should be familiar with a graph creation and a session execution. Basically the first one is for building the model and the second one is for feeding the data in and getting the results. An interesting thing is that TensorFlow does each and everything on the C++ engine, which means even a little multiplication or addition is not executed on Python but Python is just a wrapper. Fundamentally, TensorFlow C++ engine consists of following two things:

- Efficient implementations for operations like convolution, max pool, sigmoid, and so on.
- Derivatives of forwarding mode operation.

When we/you're performing a little complex operation with TensorFlow, for example training a linear regression, TensorFlow internally represents its computation using a dataflow graph. The graph is called a computational graph, which is a directed graph consisting of the following:

- A set of nodes, each one representing an operation
- A set of directed arcs, each one representing the data on which the operations are performed.

TensorFlow has two types of edges:

- **Normal**: They carry the data structures between the nodes. The output of one operation from one node, becomes input for another operation. The edge connecting two nodes carries the values.
- **Special**: This edge doesn't carry values, but only represents a control dependency between two nodes, say X and Y. It means that the node Y will be executed only if the operation in X is executed already, but before the relationship between operations on the data.

The TensorFlow implementation defines control dependencies to enforce orderings between otherwise independent operations as a way of controlling the peak memory usage.

A computational graph is basically like a dataflow graph. Figure 5 shows a computational graph for a simple computation like $z=d\times c=(a+b)\times c$:

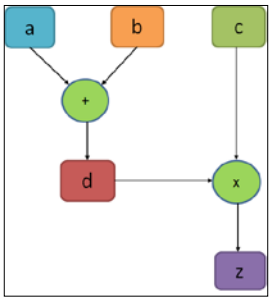


Figure 5: A very simple execution graph that computes a simple equation

In the preceding figure, the circles in the graph indicate the operations, while rectangles indicate a data computational graph. As stated earlier, a TensorFlow graph contains the following:

- **A set of tf.Operation objects**: This is used to represent units of computation to be performed
- **A tf.Tensor object**: This is used to represent units of data that control the dataflow between operations

Using TensorFlow, it is also possible to perform a deferred execution. To give an idea, once you have composed a highly compositional expression during the building phase of the computational graph, you can still evaluate them in the running session phase. Technically saying TensorFlow schedules the job and executes on time in an efficient manner. For example, parallel execution of independent parts of the code using the GPU is shown in figure 6.

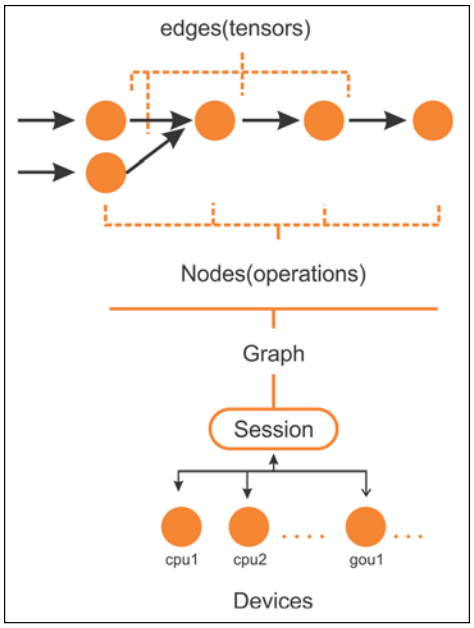


Figure 6: Edges and nodes in TensorFlow graph to be executed under a session on devices such as CPUs or GPUs

After a computational graph is created, TensorFlow needs to have an active session to be executed by multiple CPUs (and GPUs if available) in a distributed way. In general, you really don't need to specify whether to use a CPU or a GPU explicitly, since TensorFlow can choose and use which one is to be used. By default, a GPU will be picked for as many operations as possible; otherwise, a CPU will be used. So in a broad view, here are the main components of TensorFlow:

1. **Variables**: Used to contain values for the weights and bias between TensorFlow sessions.
2. **Tensors**: A set of values that pass in between nodes.
3. Placeholders: Is used to send data between the program and the TensorFlow graph.
4. Session: When a session is started, TensorFlow automatically calculates gradients for all the operations in the graph and use them in a chain rule. In fact, a session is invoked when the graph is to be executed.

Don't worry much, each of the preceding components will be discussed in later sections. Technically saying, the program you will be writing can be considered as a client. The client is then used to create the execution graph in C/C++ or Python symbolically, and then your code can ask TensorFlow to execute this graph. See details in the following figure:

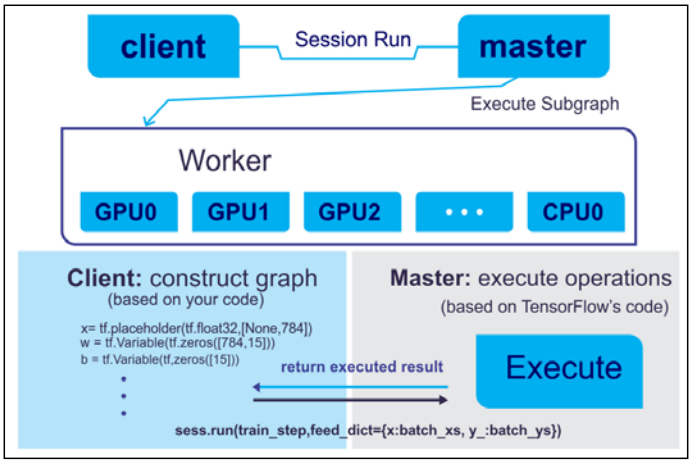


Figure 7: using a client–master architecture for executing TensorFlow graph

A computational graph helps to distribute the work load across multiple computing nodes having a CPU or a GPU. This way, a neural network can further be equaled to a composite function where each layer (input, hidden or output layer) can be represented as a function. Now to understand the operations performed on the tensors, knowing a good workaround about TensorFlow programming model is a mandate. The next section explains the role of the computational graph to implement a neural network.

TensorFlow programming model

The TensorFlow programming model signifies how to structure your predictive models. A TensorFlow program is generally divided into four phases once you have imported TensorFlow library for associated resources:

1. Construction of the computational graph that involves some operations on tensors (we will see what is a tensor soon)
2. Create a session
3. Running a session, that is performed for the operations defined in the graph
4. Computation for data collection and analysis

These main steps define the programming model in TensorFlow. Consider the following example, in which we want to multiply two numbers:

```
import tensorflow as tf
x = tf.constant(8)
y = tf.constant(9)
z = tf.multiply(x, y)
sess = tf.Session()
out_z = sess.run(z)
Finally, close the TensorFlow session when you're done:
sess.close()
print('The multiplicaiton of x and y: %d' % out_z)
```

The preceding code segment can be represented by the following figure:

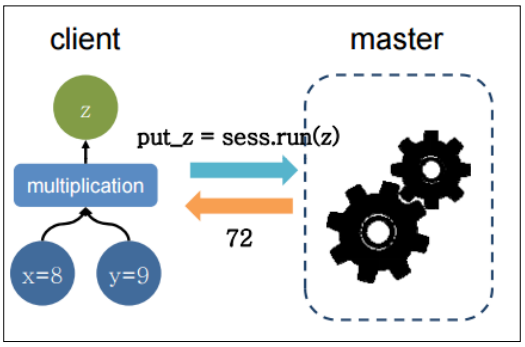


Figure 8: A simple multiplication executed and returned the product on client-master architecture

To make the preceding program more efficient, TensorFlow also allows you to exchange data in your graph variables through placeholders (to be discussed later). Now imagine the following code segment that does the same but in a more efficient way:

```
# Import tensorflow
import tensorflow as tf
# Build a graph and create session passing the graph:
with tf.Session() as sess:
 x = tf.placeholder(tf.float32, name="x")
 y = tf.placeholder(tf.float32, name="y")
 z = tf.multiply(x,y)
# Put the values 8,9 on the placeholders x,y and execute the graph
z_output = sess.run(z,feed_dict={x: 8, y:9})
# Finally, close the TensorFlow session when you're done:
sess.close()
print(z_output)
```

TensorFlow is not necessary to multiply two numbers; also the number of lines of the code for this simple operation is so many. However, the example wants to clarify how to structure any code, from the simplest as in this instance, to the most complex. Furthermore, the example also contains some basic instructions that we will find in all the other examples given in the course of this book.



We will demonstrate most of the examples in this book with Python 3.x compatible. However, a few examples will be given using Python 2.7.x too.

This single import in the first line helps to import the TensorFlow for your command that can be instantiated with `tf as stated earlier. Then the` TensorFlow operator will then be expressed by `tf` and the dot `'.'` and by the name of the operator to use. In the next line, we construct the object `session`, by means of the instruction `tf.Session()`:

```
with tf.Session() as sess:
```



The session object (that is, sess) encapsulates the environment for the TensorFlow so that all the operation objects are executed, and Tensor objects are evaluated. We will see them in upcoming sections.

This object contains the computation graph, which as we said earlier, are the calculations to be carried out.

The following two lines define variables x and y, using the notion of placeholder. Through a placeholder you may define both an input (such as the variable x of our example) and an output variable (such as the variable y):

```
x = tf.placeholder(tf.float32, name='x')
y = tf.placeholder(tf.float32, name='y')
```

Placeholder provides an interface between the elements of the graph and the computational data of the problem, it allows us to create our operations and build our computation graph, without needing the data, but only a reference to it.

To define a data or tensor (soon I will introduce you to the concept of tensor) via the placeholder function, three arguments are required:

- **Data type**: Is the type of element in the tensor to be fed.
- **Shape**: Of the placeholder–that is, shape of the tensor to be fed (optional). If the shape is not specified, you can feed a tensor of any shape.

- **Name**: Very useful for debugging and code analysis purposes, but it is optional.



For more, refer to`https://www.tensorflow.org/api_docs/python/tf/Tensor`.

So, we may introduce the model that we want to compute with two arguments, the placeholder and the constant that are previously defined. Next, we define the computational model.

The following statement, inside the session, builds the data structures of the `x` product with `y`, and the subsequent assignment of the result of the operation to the placeholder `z`. Then it goes as follows:

```
z = tf.multiply(x, y)
```

Now since the result is already held by the placeholder `z`, we execute the graph, through the `sess.run` statement. Here we feed two values to patch a tensor into a graph node. It temporarily replaces the output of an operation with a tensor value (more in upcoming sections):

```
z_output = sess.run(z,feed_dict={x: 8, y:9})
```

Then we close the TensorFlow session when we're done:

```
sess.close()
```

In the final instruction, we print out the result:

```
print(z_output)
```

This essentially prints output 72.0.

Data model in TensorFlow

The data model in TensorFlow is represented by **tensors**. Without using complex mathematical definitions, we can say that a tensor (in TensorFlow) identifies a multidimensional numerical array. But we will see more details on tensor in the next sub-section.

Tensors

Let's see a formal definition of tensors from Wikipedia (`https://en.wikipedia.org/wiki/Tensor`) as follows:

> *"Tensors are geometric objects that describe linear relations between geometric vectors, scalars, and other tensors. Elementary examples of such relations include the dot product, the cross product, and linear maps. Geometric vectors, often used in physics and engineering applications, and scalars themselves are also tensors."*

This data structure is characterized by three parameters: Rank, Shape, and Type, as shown in the following figure:

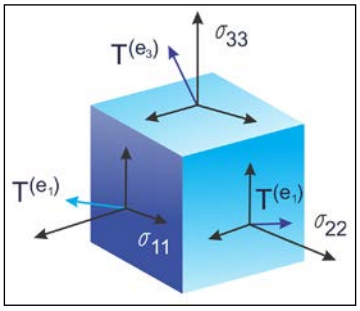


Figure 9: Tensors are nothing but geometrics objects having shape, rank, and type used to hold multidimensional arrays

A tensor thus can be thought of as a generalization of a matrix that specifies an element by an arbitrary number of indices. While practically used, the syntax for tensors is even more or less like nested vectors.



Tensors just define the type of this value and the means by which this value should be calculated during the session. Therefore, essentially, they do not represent or hold any value produced by an operation.

A few people love to compare NumPy versus TensorFlow comparison; however, in reality, TensorFlow and NumPy are quite similar in a sense that both are N-d array libraries!

Well, it's true that NumPy has the n–dimensional array support, but it doesn't offer methods to create tensor functions and automatically compute derivatives (+ no GPU support). The following table can be seen as a short and one-to-one comparison that could make some sense of such comparisons:

Numpy	TensorFlow
`a = np.zeros((2,2)); b = np.ones((2,2))`	`a = tf.zeros((2,2)), b = tf.ones((2,2))`
`np.sum(b, axis=1)`	`tf.reduce_sum(a,reduction_indices=[1])`
`a.shape`	`a.get_shape()`
`np.reshape(a, (1,4))`	`tf.reshape(a, (1,4))`
`b * 5 + 1`	`b * 5 + 1`
`np.dot(a,b)`	`tf.matmul(a, b)`
`a[0,0], a[:,0], a[0,:]`	`a[0,0], a[:,0], a[0,:]`

Figure 10: NumPy vs TensorFlow

Now let's see an alternative way of creating tensors before they could be fed (we will see other feeding mechanisms later on) by the TensorFlow graph:

```
>>> X = [[2.0, 4.0],
         [6.0, 8.0]]
>>> Y = np.array([[2.0, 4.0],
                  [6.0, 6.0]], dtype=np.float32)
>>> Z = tf.constant([[2.0, 4.0],
                     [6.0, 8.0]])
```

Here `X` is a list, `Y` is an n-dimensional array from the NumPy library, and `Z` is itself the TensorFlow's Tensor object. Now let's see their types:

```
>>> print(type(X))
>>> print(type(Y))
>>> print(type(Z))
#Output
<class 'list'>
<class 'numpy.ndarray'>
<class 'tensorflow.python.framework.ops.Tensor'>
```

Well, their types are printed correctly. However, a more convenient function that we're formally dealing with tensors, as opposed to the other types is `tf.convert_to_tensor()` function as follows:

```
t1 = tf.convert_to_tensor(X, dtype=tf.float32)
t2 = tf.convert_to_tensor(Z, dtype=tf.float32)
t3 = tf.convert_to_tensor(Z, dtype=tf.float32)
```

Now let's see their type using the following lines:

```
>>> print(type(t1))
>>> print(type(t2))
>>> print(type(t3))
#Output:
<class 'tensorflow.python.framework.ops.Tensor'>
<class 'tensorflow.python.framework.ops.Tensor'>
<class 'tensorflow.python.framework.ops.Tensor'>
```

Fantastic! I think up to now it's enough discussion already carried out on tensors, so now we can think about the structure that is characterized by the term **rank**.

Rank

Each tensor is described by a unit of dimensionality called rank. It identifies the number of dimensions of the tensor, for this reason, a rank is known as order or n–dimensions of a tensor. A rank zero tensor is a scalar, a rank one tensor id a vector, while a rank two tensor is a matrix. The following code defines a TensorFlow scalar, a `vector`, a `matrix`, and a `cube_matrix`, in the next example we will show how the rank works:

```
import tensorflow as tf
scalar = tf.constant(100)
vector = tf.constant([1,2,3,4,5])
matrix = tf.constant([[1,2,3],[4,5,6]])
cube_matrix = tf.constant([[[1],[2],[3]],[[4],[5],[6]],[[7],[8],[9]]])
print(scalar.get_shape())
print(vector.get_shape())
print(matrix.get_shape())
print(cube_matrix.get_shape())
```

The results are printed here:

```
>>>
()
(5,)
(2, 3)
(3, 3, 1)
>>>
```

Shape

The shape of a tensor is the number of rows and columns it has. Now we will see how to relate the shape to a rank of a tensor:

```
>>scalar1.get_shape()
TensorShape([])
>>vector1.get_shape()
TensorShape([Dimension(5)])
>>matrix1.get_shape()
TensorShape([Dimension(2), Dimension(3)])
>>cube1.get_shape()
TensorShape([Dimension(3), Dimension(3), Dimension(1)])
```

Data type

In addition to rank and shape, tensors have a data type. The following is the list of the data types:

Data type	Python type	Description
DT_FLOAT	tf.float32	32 bits floating point.
DT_DOUBLE	tf.float64	64 bits floating point.
DT_INT8	tf.int8	8 bits signed integer.
DT_INT16	tf.int16	16 bits signed integer.
DT_INT32	tf.int32	32 bits signed integer.
DT_INT64	tf.int64	64 bits signed integer.
DT_UINT8	tf.uint8	8 bits unsigned integer.
DT_STRING	tf.string	Variable length byte arrays. Each element of a tensor is a byte array.
DT_BOOL	tf.bool	Boolean.
DT_COMPLEX64	tf.complex64	Complex number made of two 32 bits floating points: real and imaginary parts.
DT_COMPLEX128	tf.complex128	Complex number made of two 64 bits floating points: real and imaginary parts.
DT_QINT8	tf.qint8	8 bits signed integer used in quantized Ops.
DT_QINT32	tf.qint32	32 bits signed integer used in quantized Ops.
DT_QUINT8	tf.quint8	8 bits unsigned integer used in quantized Ops.

We believe the preceding table is self-explanatory hence we did not provide detailed discussion on the preceding data types. Now the TensorFlow APIs are implemented to manage data *to* and *from* NumPy arrays. Thus, to build a tensor with a constant value, pass a NumPy array to the `tf.constant()` operator, and the result will be a TensorFlow tensor with that value:

```
import tensorflow as tf
import numpy as np
tensor_1d = np.array([1,2,3,4,5,6,7,8,9,10])
tensor_1d = tf.constant(tensor_1d)
with tf.Session() as sess:
    print (tensor_1d.get_shape())
    print sess.run(tensor_1d)
# Finally, close the TensorFlow session when you're done
sess.close()
```

Running the example, we obtain:

```
>>>
 (10,)
 [ 1  2  3  4  5  6  7  8  9 10]
```

To build a tensor, with variable values, use a `NumPy` array and pass it to the `tf.Variable` constructor, the result will be a TensorFlow variable tensor with that initial value:

```
import tensorflow as tf
import numpy as np
tensor_2d = np.array([(1,2,3),(4,5,6),(7,8,9)])
tensor_2d = tf.Variable(tensor_2d)
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    print (tensor_2d.get_shape())
    print sess.run(tensor_2d)

# Finally, close the TensorFlow session when you're done
sess.close()
```

The result is:

```
>>>
 (3, 3)
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
```

For ease of use in interactive Python environments, we can use the `InteractiveSession` class, and then use that session for all `Tensor.eval()` and `Operation.run()` calls:

```
import tensorflow as tf
import numpy as np

interactive_session = tf.InteractiveSession()
tensor = np.array([1,2,3,4,5])
tensor = tf.constant(tensor)
print(tensor.eval())

interactive_session.close()
```



`tf.InteractiveSession()` is just a convenient syntactic sugar for keeping a default session open in Ipython.

The result is:

```
>>>
   [1 2 3 4 5]
```

This can be easier in an interactive setting, such as the shell or an IPython notebook, when it's tedious to pass around a session object everywhere.



The IPython Notebook is now known as the Jupyter Notebook. It is an interactive computational environment, in which you can combine code execution, rich text, mathematics, plots and rich media. For more information, interested readers should refer to the web page at `https://ipython.org/notebook.html`.

Another way to define a tensor is using the TensorFlow statement `tf.convert_to_tensor`:

```
import tensorflow as tf
import numpy as np
tensor_3d = np.array([[[0, 1, 2], [3, 4, 5], [6, 7, 8]],
                      [[9, 10, 11], [12, 13, 14], [15, 16, 17]],
                      [[18, 19, 20], [21, 22, 23], [24, 25, 26]]])
tensor_3d = tf.convert_to_tensor(tensor_3d, dtype=tf.float64)
with tf.Session() as sess:
    print(tensor_3d.get_shape())
    print(sess.run(tensor_3d))
```

```
# Finally, close the TensorFlow session when you're done
sess.close()
>>>
(3, 3, 3)
[[[  0.   1.   2.]
  [  3.   4.   5.]
  [  6.   7.   8.]]
 [[  9.  10.  11.]
  [ 12.  13.  14.]
  [ 15.  16.  17.]]
 [[ 18.  19.  20.]
  [ 21.  22.  23.]
  [ 24.  25.  26.]]]
```

Variables

Variables are TensorFlow objects to hold and update parameters. A variable must be initialized; also you can save and restore it to analyze your code. Variables are created by using the `tf.Variable()` statement. In the following example, we want to count the numbers from 1 to 10, but let's import TensorFlow first:

```
import tensorflow as tf
```

We created a variable that will be initialized to the scalar value `0`:

```
value = tf.Variable(0, name="value")
```

The `assign()` and `add()` operators are just nodes of the computation graph, so they do not execute the assignment until the run of the session:

```
one = tf.constant(1)
new_value = tf.add(value, one)
update_value = tf.assign(value, new_value)
initialize_var = tf.global_variables_initializer()
```

We can instantiate the computation graph:

```
with tf.Session() as sess:
    sess.run(initialize_var)
    print(sess.run(value))
    for _ in range(5):
        sess.run(update_value)
        print(sess.run(value))
# Finally, close the TensorFlow session when you're done:
sess.close()
```

Let's recall that a tensor object is a symbolic handle to the result of an operation, but it does not actually hold the values of the operation's output:

```
>>>
0
1
2
3
4
5
```

Fetches

To fetch the outputs of operations, execute the graph by calling `run()` on the session object and pass in the tensors to retrieve. Except fetching the single tensor node, you can also fetch multiple tensors. In the following example, the sum and multiply tensors are fetched together, using the `run()` call:

```
import tensorflow as tf

constant_A = tf.constant([100.0])
constant_B = tf.constant([300.0])
constant_C = tf.constant([3.0])

sum_ = tf.add(constant_A,constant_B)
mul_ = tf.multiply(constant_A,constant_C)

with tf.Session() as sess:
    result = sess.run([sum_,mul_])
    print(result)

# Finally, close the TensorFlow session when you're done:
sess.close()
```

The output is as follows:

```
>>>
[array(400.],dtype=float32),array([ 300.],dtype=float32)]
```

All the ops needed to produce the values of the requested tensors are run once (not once per requested tensor).

Feeds and placeholders

There are four methods of getting data into a TensorFlow program (see details at `https://www.tensorflow.org/api_guides/python/reading_data`):

- **The Dataset API**: This enables you to build complex input pipelines from simple and reusable pieces from distributed file systems and perform complex operations. Using the Dataset API is recommended while dealing with large amounts of data in different data formats. The Dataset API introduces two new abstractions to TensorFlow for creating feedable dataset using either `tf.contrib.data.Dataset` (by creating a source or applying a transformation operations) or using a `tf.contrib.data.Iterator`.
- **Feeding**: Allows us to inject data into any Tensor in a computation graph.
- **Reading from files**: We can develop an input pipeline using Python's built-in mechanism for reading data from data files at the beginning of a TensorFlow graph.
- **Preloaded data**: For small datasets, we can use either constants or variables in the TensorFlow graph for holding all the data.

In this section, we will see an example of the feeding mechanism only. For the other methods, we will see them in upcoming chapters. TensorFlow provides the feed mechanism that allows us inject data into any tensor in a computation graph. You can provide the feed data through the `feed_dict` argument to a `run()` or `eval()` invoke that initiates the computation.



Feeding using the `feed_dict` argument is the least efficient way to feed data into a TensorFlow execution graph and should only be used for small experiments needing small datasets. It can also be used for debugging.

We can also replace any tensor with feed data (that is variables and constants), the best practice is to use a TensorFlow placeholder node using `tf.placeholder()` invocation. A placeholder exists exclusively to serve as the target of feeds. An empty placeholder is not initialized so it does not contain any data. Therefore, it will always generate an error if it is executed without a feed, so you won't forget to feed it.

The following example shows how to feed data to build a random 2×3 matrix:

```
import tensorflow as tf
import numpy as np

a = 3
b = 2
```

```
x = tf.placeholder(tf.float32,shape=(a,b))
y = tf.add(x,x)

data = np.random.rand(a,b)
sess = tf.Session()
print sess.run(y,feed_dict={x:data})

# Finally, close the TensorFlow session when you're done:
sess.close()
```

The output is:

```
>>>
[[ 1.78602004  1.64606333]
 [ 1.03966308  0.99269408]
 [ 0.98822606  1.50157797]]
>>>
```

TensorBoard

TensorFlow includes functions to debug and optimize programs in a visualization tool called **TensorBoard**. Using TensorBoard, you can observe different types of statistics concerning the parameters and details of any part of the graph computing graphically.

Moreover, while doing predictive modeling using the complex deep neural network, the graph can be complex and confusing. Thus to make it easier to understand, debug, and optimize TensorFlow programs, you can use TensorBoard to visualize your TensorFlow graph, plot quantitative metrics about the execution of your graph, and show additional data such as images that pass through it.

Therefore, the TensorBoard can be thought of as a framework designed for analysis and debugging of predictive models. TensorBoard uses the so-called summaries to view the parameters of the model: once a TensorFlow code is executed, we can call TensorBoard to view summaries in a GUI.

How does TensorBoard work?

As explained previously, TensorFlow uses the computation graph to execute an application, where each node represents an operation and the arcs are the data between operations.

The main idea in TensorBoard is to associate the so-called summary with nodes (operations) of the graph. Upon running the code, the summary operations will serialize the data of the node that is associated with it and output the data into a file that can be read by TensorBoard. Then TensorBoard can be run and visualize the summarized operations. The workflow when using TensorBoard is:

- Build your computational graph/code
- Attach summary ops to the nodes you are interested in examining
- Start running your graph as you normally would
- Additionally, run the summary ops
- When the code is done running, run TensorBoard to visualize the summary outputs

If you type `$ which tensorboard` in your terminal, it should exist if you installed with `pip`:

```
asif@ubuntu:~$ which tensorboard
/usr/local/bin/tensorboard
```

You need to give it a log directory, so you are in the directory where you ran your graph; you can launch it from your terminal with something like:

```
tensorboard --logdir .
```

Then open your favorite web browser and type in `localhost:6006` to connect. When TensorBoard is fully configured, this can be accessed by issuing the following command:

```
$ tensorboard –logdir=<trace_file_name>
```

Now you simply need to access the local port 6006 from the browser `http://localhost:6006/`. Then it should look like this:



Figure 11: Using TensorBoard on browser

Is this already too much? Don't worry, in the last section, we'll combine all the ideas previously explained to build a single input neuron model and to analyze it with TensorBoard.

Getting started with TensorFlow – linear regression and beyond

In this example, we will take a closer look at TensorFlow's and TensorBoard's main concepts and try to do some basic operations to get you started. The model we want to implement simulates the linear regression.

In the statistics and machine learning realm, linear regression is a technique frequently used to measure the relationship between variables. This is also a quite simple but effective algorithm that can be used in predictive modeling too. Linear regression models the relationship between a dependent variable y_i, an interdependent variable x_i, and a random term b. This can be seen as follows:

$$y = W * \mathrm{x} + \mathrm{b}$$

Now to conceptualize the preceding equation, I am going to write a simple Python program for creating data into a 2D space. Then I will use TensorFlow to look for the line that best fits in the data points:

```
# Import libraries (Numpy, matplotlib)

import numpy as np
import matplotlib.pyplot as plt

# Create 1000 points following a function y=0.1 * x + 0.4 (i.e. y \= W
* x + b) with some normal random distribution:

num_points = 1000
vectors_set = []
for i in range(num_points):
    W = 0.1 # W
    b = 0.4 # b
    x1 = np.random.normal(0.0, 1.0)
    nd = np.random.normal(0.0, 0.05)
    y1 = W * x1 + b

 # Add some impurity with some normal distribution -i.e. nd:
    y1 = y1+nd

 # Append them and create a combined vector set:
    vectors_set.append([x1, y1])

# Separate the data point across axises:
x_data = [v[0] for v in vectors_set]
y_data = [v[1] for v in vectors_set]

# Plot and show the data points in a 2D space
plt.plot(x_data, y_data, 'r*', label='Original data')
plt.legend()
plt.show()
```

If your compiler does not make any complaints, you should observe the following graph:

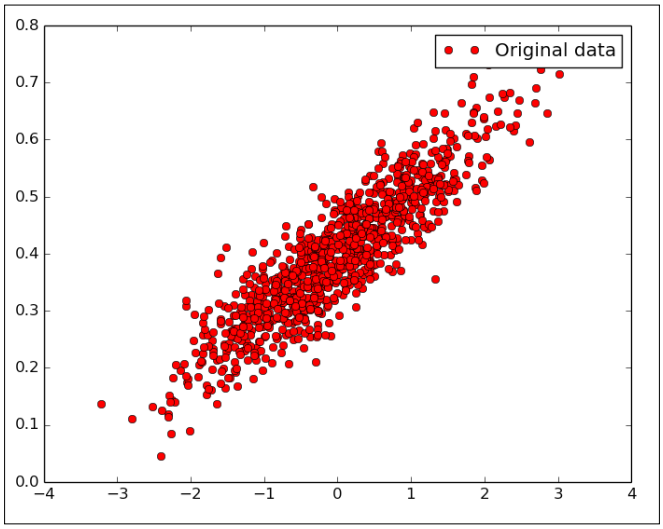


Figure 12: randomly generated (but original) data

Well, so far we have just created a few data points without any associated model that could be executed through TensorFlow. So the next step is to create a linear regression model to be able to obtain the output values `y` that is estimated from the input data points–that is, `x_data`. In this context, we have only two associated parameters–that is, `W` and `b`. Now the objective is to create a graph that allows finding the values for these two parameters based on the input data `x_data` by adjusting them to `y_data`–that is, optimization problem.

So the target function in our case would be as follows:

$$y_\text{data} = W * \text{x_data} + \text{b}$$

If you recall, we defined $W = 0.1$ and $b = 0.4$ while creating the data points in the 2D space. Now TensorFlow has to optimize these two values so that `W` tends to 0.1 and `b` to 0.4, but without knowing any optimization function, TensorFlow does not even know anything.

A standard way to solve such optimization problems is to iterate through each value of the data points and adjust the value of `W` and `b` in order to get a more precise answer on each iteration. Now to realize if the values are really improving, we need to define a cost function that measures how good a certain line is.

In our case, the cost function is the mean squared error that helps find the average of the errors based on the distance function between the real data points and the estimated ones on each iteration. We start by importing the TensorFlow library:

```
import tensorflow as tf
W = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
b = tf.Variable(tf.zeros([1]))
y = W * x_data + b
```

In the preceding code segment, we are generating a random point using a different strategy and storing in variable W. Now let's define a loss function *loss=mean [(y−y_data)* ^{2}*]* and this returns a scalar value with the mean of all distances between our data and the model prediction. In terms of TensorFlow convention, the loss function can be expressed as follows:

```
loss = tf.reduce_mean(tf.square(y - y_data))
```

Without going into further detail, we can use some widely used optimization algorithms such as gradient descent. At a minimal level, the gradient descent is an algorithm that works on a set of given parameters that we already have. It starts with an initial set of parameter values and iteratively moves toward a set of values that minimize the function by taking another parameter called learning rate. This iterative minimization is achieved by taking steps in the negative direction of the function called gradient.

```
optimizer = tf.train.GradientDescentOptimizer(0.6)
train = optimizer.minimize(loss)
```

Before running this optimization function, we need to initialize all the variables that we have so far. Let's do it using TensorFlow convention as follows:

```
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)
```

Since we have created a TensorFlow session, we are ready for the iterative process that helps us find the optimal values of `W` and `b`:

```
for i in range(16):
  sess.run(train)
  print(i, sess.run(W), sess.run(b), sess.run(loss))
```

You should observe the following output:

```
>>>
0 [ 0.18418592] [ 0.47198644] 0.0152888
1 [ 0.08373772] [ 0.38146532] 0.00311204
2 [ 0.10470386] [ 0.39876288] 0.00262051
```

```
3 [ 0.10031486] [ 0.39547175] 0.00260051
4 [ 0.10123629] [ 0.39609471] 0.00259969
5 [ 0.1010423] [ 0.39597753] 0.00259966
6 [ 0.10108326] [ 0.3959994] 0.00259966
7 [ 0.10107458] [ 0.39599535] 0.00259966
```

Thus you can see the algorithm starts with the initial values of $W = 0.18418592$ and $b = 0.47198644$ where the loss is pretty high. Then the algorithm iteratively adjusted the values by minimizing the cost function. In the eighth iteration, all the values tend to our desired values.

Now what if we could plot them? Let's do it by adding the plotting line under the `for` loop as follows:

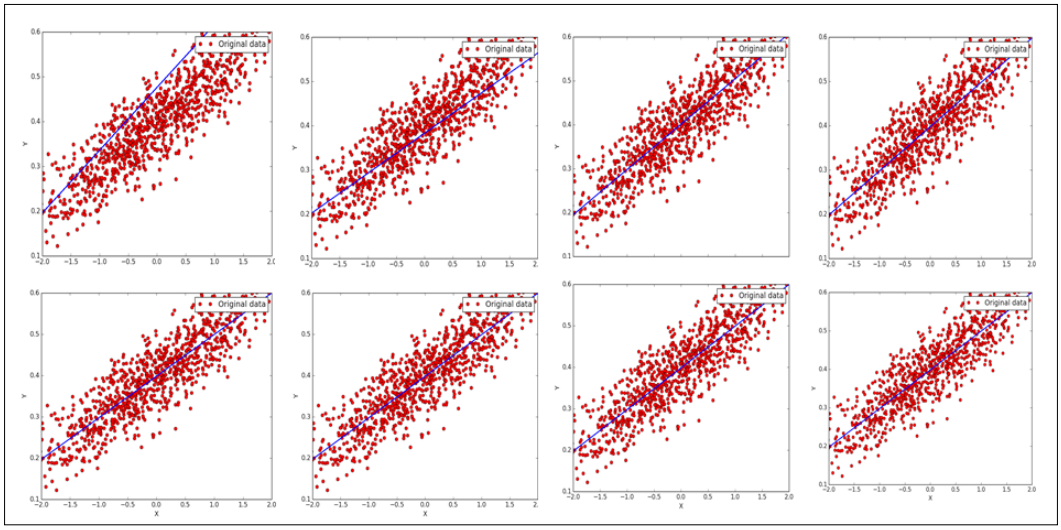


Figure 13: Linear regression after 8th iteration that optimizes the loss function

Now let's iterate the same up to the 16th iteration:

```
>>>
0 [ 0.23306453] [ 0.47967502] 0.0259004
1 [ 0.08183448] [ 0.38200468] 0.00311023
2 [ 0.10253634] [ 0.40177572] 0.00254209
3 [ 0.09969243] [ 0.39778906] 0.0025257
4 [ 0.10008509] [ 0.39859086] 0.00252516
5 [ 0.10003048] [ 0.39842987] 0.00252514
6 [ 0.10003816] [ 0.39846218] 0.00252514
7 [ 0.10003706] [ 0.39845571] 0.00252514
8 [ 0.10003722] [ 0.39845699] 0.00252514
```

```
9 [ 0.10003719] [ 0.39845672] 0.00252514
10 [ 0.1000372] [ 0.39845678] 0.00252514
11 [ 0.1000372] [ 0.39845678] 0.00252514
12 [ 0.1000372] [ 0.39845678] 0.00252514
13 [ 0.1000372] [ 0.39845678] 0.00252514
14 [ 0.1000372] [ 0.39845678] 0.00252514
15 [ 0.1000372] [ 0.39845678] 0.00252514
```

Much better and we're closer to the optimized values, right? Now, what if we further improve our visual analytics through TensorFlow that help visualize what is happening in these graphs. TensorBoard provides a web page for debugging your graph as well as inspecting the used variables, node, edges, and their corresponding connections.

However, to get the facility of the preceding regression analysis, you need to annotate the preceding graphs with the variables such as loss function, `W`, `b`, `y_data`, `x_data`, and so on. Then you need to generate all the summaries by invoking the function `tf.summary.merge_all()`.

Now, we need to make the following changes to the preceding code. However, it is a good practice to group related nodes on the graph using the `tf.name_scope()` function. Thus, we can use `tf.name_scope()` to organize things on the TensorBoard graph view, but let's give it a better name:

```
with tf.name_scope("LinearRegression") as scope:
   W = tf.Variable(tf.random_uniform([1], -1.0, 1.0), name="Weights")
   b = tf.Variable(tf.zeros([1]))
   y = W * x_data + b
```

Then let's annotate the loss function in a similar way, but by giving a suitable name such as `LossFunction`:

```
with tf.name_scope("LossFunction") as scope:
  loss = tf.reduce_mean(tf.square(y - y_data))
```

Let's annotate the loss, weights, and bias that are needed for the TensorBoard:

```
loss_summary = tf.summary.scalar("loss", loss)
w_ = tf.summary.histogram("W", W)
b_ = tf.summary.histogram("b", b)
```

Well, once you annotate the graph, it's time to configure the summary by merging them:

```
merged_op = tf.summary.merge_all()
```

Now before running the training (after the initialization), write the summary using the `tf.summary.FileWriter()` API as follows:

```
writer_tensorboard = tf.summary.FileWriter('/home/asif/LR/', sess.
graph_def)
```

Then start the TensorBoard as follows:

```
$ tensorboard –logdir=<trace_file_name>
```

In our case, it could be something like the following:

```
$ tensorboard --logdir=/home/asif/LR/
```

Now let's move to `http://localhost:6006` and on clicking on the **GRAPHS** tab, you should see the following graph:

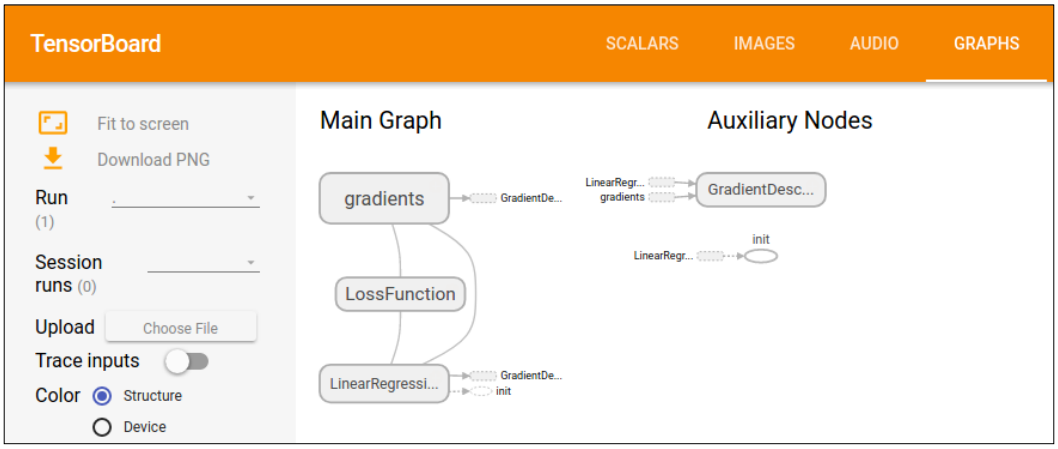


Figure 14: main graph and auxiliary nodes on TensorBoard

Source code for the linear regression

We reported for the entire source code for the example previously described:

```
# Import libraries (Numpy, Tensorflow, matplotlib)
import numpy as np
import matplotlib.pyplot as plot

# Create 1000 points following a function y=0.1 * x + 0.4 (i.e. y = W
* x + b) with some normal random distribution:
num_points = 1000
vectors_set = []
for i in range(num_points):
    W = 0.1  # W
```

```
    b = 0.4  # b
    x1 = np.random.normal(0.0, 1.0)
    nd = np.random.normal(0.0, 0.05)
    y1 = W * x1 + b

# Add some impurity with some normal distribution -i.e. nd:
    y1 = y1 + nd

# Append them and create a combined vector set:
    vectors_set.append([x1, y1])

# Separate the data point across axises
x_data = [v[0] for v in vectors_set]
y_data = [v[1] for v in vectors_set]

# Plot and show the data points in a 2D space
plot.plot(x_data, y_data, 'ro', label='Original data')
plot.legend()
plot.show()

import tensorflow as tf

#tf.name_scope organize things on the tensorboard graph view
with tf.name_scope("LinearRegression") as scope:
   W = tf.Variable(tf.random_uniform([1], -1.0, 1.0), name="Weights")
   b = tf.Variable(tf.zeros([1]))
   y = W * x_data + b

# Define a loss function that takes into account the distance between
the prediction and our dataset
with tf.name_scope("LossFunction") as scope:
   loss = tf.reduce_mean(tf.square(y - y_data))

optimizer = tf.train.GradientDescentOptimizer(0.6)
train = optimizer.minimize(loss)


# Annotate loss, weights, and bias (Needed for tensorboard)
loss_summary = tf.summary.scalar("loss", loss)
w_ = tf.summary.histogram("W", W)
b_ = tf.summary.histogram("b", b)

# Merge all the summaries
merged_op = tf.summary.merge_all()

init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)
```

```
# Writer for TensorBoard  (replace with our preferred location
writer_tensorboard = tf.summary.FileWriter('/ LR/', sess.graph_def)

for i in range(16):
   sess.run(train)
   print(i, sess.run(W), sess.run(b), sess.run(loss))
   plot.plot(x_data, y_data, 'ro', label='Original data')
   plot.plot(x_data, sess.run(W)*x_data + sess.run(b))
   plot.xlabel('X')
   plot.xlim(-2, 2)
   plot.ylim(0.1, 0.6)
   plot.ylabel('Y')
   plot.legend()
   plot.show()
# Finally, close the TensorFlow session when you're done
sess.close()
```

Ubuntu may ask you to install the python-tk package. You can do it by executing the following command on Ubuntu:

```
$ sudo apt-get install python-tk
# For Python 3.x, use the following
$ sudo apt-get install python3-tk
```

Summary

TensorFlow is designed to make the predictive analytics through the machine and deep learning easy for everyone, but using it does require understanding some general principles and algorithms. Furthermore, the latest release of TensorFlow comes with lots of exciting features. Thus I also tried to cover them so that you can use them with ease. I have shown how to install TensorFlow on different platforms including Linux, Windows, and Mac OS. In summary, here is a brief recap of the key concepts of TensorFlow explained in this chapter:

- **Graph**: each TensorFlow computation can be represented as a set of dataflow graphs where each graph is built as a set of operation objects. There are three core graph data structures:
    1. `tf.Graph`
    2. `tf.Operation`
    3. `tf.Tensor`

- **Operation**: A graph node takes tensors as input and also produces a tensor as output. A node can be represented by an operation object for performing units of computations such as addition, multiplication, division, subtraction or more complex operation.
- **Tensor**: Tensors are like high-dimensional array objects. In other words, they can be represented as edges of a dataflow graph but still they don't hold any value produced out of an operations.
- **Session**: A session object is an entity that encapsulates the environment in which operation objects are executed for running calculations on the dataflow graph. As a result, the tensors objects are evaluated inside the `run()` or `eval()` invocation.

In a later section of the chapter, we introduced TensorBoard, which is a powerful tool for analyzing and debugging neural network models, the chapter ended with an example that shows how to implement a simple neuron model and how to analyze its learning phase with TensorBoard.

Predictive models often perform calculations during live transactions, for example, to evaluate the risk or opportunity of a given customer or transaction, in order to guide a decision. With advancements in computing speed, individual agent modeling systems have become capable of simulating human behavior or reactions to given stimuli or scenarios. In the next chapter, we will cover linear models for regression, classification, and clustering and dimensionality reduction and will also give some insights about some performance measures.

4
Putting Data in Place - Supervised Learning for Predictive Analytics

In this chapter, we will discuss supervised learning from the theoretical and practical perspective. In particular, we will revisit the linear regression model for regression analysis discussed in *Chapter 3*, *From Data to Decisions - Getting Started with TensorFlow*, using a real dataset. Then we will see how to develop Titanic survival predictive models using **Logistic Regression** (**LR**), **Random Forests**, and **Support Vector Machines** (**SVMs**).

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

- Supervised learning for predictive analytics
- Linear regression for predictive analytics: revisited
- Logistic regression for predictive analytics
- Random forests for predictive analytics
- SVMs for predictive analytics
- A comparative analysis

Supervised learning for predictive analytics

Depending on the nature of the learning feedback available, the machine learning process is typically classified into three broad categories: supervised learning, unsupervised learning, and reinforcement learning – see figure 1. A predictive model based on supervised learning algorithms can make predictions based on a labelled dataset that map inputs to outputs aligning with the real world.

For example, a dataset for spam filtering usually contains spam messages as well as not-spam messages. Therefore, we could know which messages in the training set are spam and which are ham. Nevertheless, we might have the opportunity to use this information to train our model in order to classify new unseen messages:

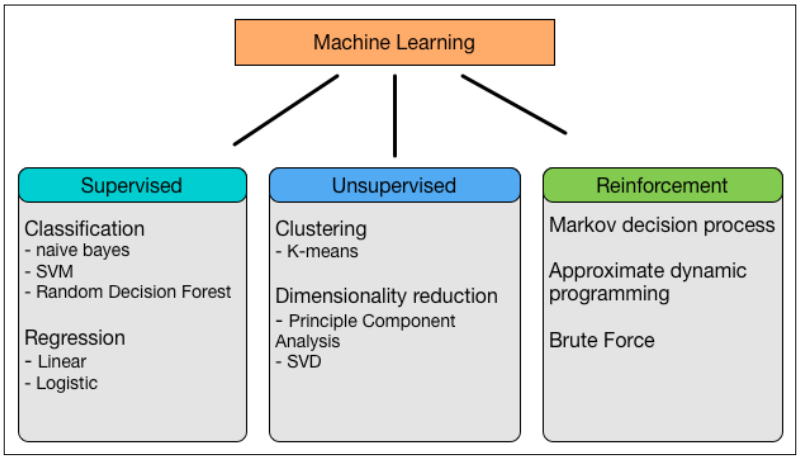


Figure 1: Machine learning tasks (containing a few algorithms only)

The following figure shows the schematic diagram of supervised learning. After the algorithm has found the required patterns, those patterns can be used to make predictions for unlabeled test data:

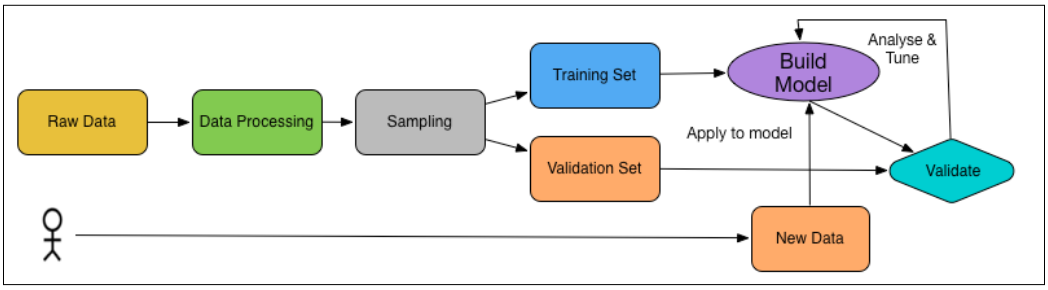


Figure 2: Supervised learning in action

Examples include classification and regression for solving supervised learning problems so that predictive models can be built for predictive analytics based on them. We will provide several examples of supervised learning like linear regression, logistic regression, random forest, decision trees, Naive Bayes, multilayer perceptron, and so on.

In this chapter, we will mainly focus on the supervised learning algorithms for predictive analytics. Let's start from the very simple linear regression algorithm.

Linear regression - revisited

In *Chapter 3*, *From Data to Decisions - Getting Started with TensorFlow* we have seen an example of linear regression. We have observed how to work TensorFlow on the randomly generated dataset, that is, fake data. We have seen that the regression is a type of supervised machine learning for predicting the continuous-valued output. However, running a linear regression on fake data is just like buying a new car we talked about in *Chapter 1*, *Basic Python and Linear Algebra for Predictive Analytics*, and never driving it. This awesome machinery begs to manifest itself in the real world!

Fortunately, many datasets are available online to test your new-found knowledge of regression:

- The University of Massachusetts Amherst supplies small datasets of various types: `http://www.umass.edu/statdata/statdata/`
- Kaggle contains all types of large-scale data for machine learning competitions: `https://www.kaggle.com/datasets`
- Data.gov is an open data initiative by the US government, which contains many interesting and practical datasets: `https://catalog.data.gov`

Therefore, in this section, by defining a set of models, we will see how to reduce the search space of possible functions. Moreover, TensorFlow takes advantage of the differentiable property of the functions by running its efficient gradient descent optimizers to learn the parameters. To avoid overfitting our data, we regularize the cost function by penalizing larger valued parameters.

The linear regression is shown in *Chapter 3*, *From Data to Decision - Getting Started with TensorFlow*, shows some tensors that just contained a single scalar value, but you can, of course, perform computations on arrays of any shape. In TensorFlow, operations such as addition and multiplication take two inputs and produce an output. In contrast, constants and variables do not take any input. We will also see an example of how TensorFlow can manipulate 2D arrays to perform linear regression like operations.

Problem statement

Online movie ratings and recommendations have become a serious business around the world. For example, Hollywood generates about $10 billion at the U.S. box office each year. Websites like Rotten Tomatoes aggregates movie reviews into one overall rating and also reports poor opening weekends. Although a single movie critic or a single negative review can't make or break a film, thousands of reviews and critics do.

Rotten Tomatoes, Metacritic, and IMDb have their own way of aggregating film reviews and distinct rating systems. On the other hand, Fandango, a NBCUniversal subsidiary uses a five-star rating system in which most of the movies get at least three stars, according to a FiveThirtyEight analysis.

An exploratory analysis of the dataset used by Fandango shows that out of 510 films, 437 films got at least one review where, hilariously, 98% had a 3-star rating or higher and 75 percent had a 4-star rating or higher. This implies, that using Fandango's standards it's almost impossible for a movie to be a flop at the box office. Therefore, Fandango's rating is biased and skewed:

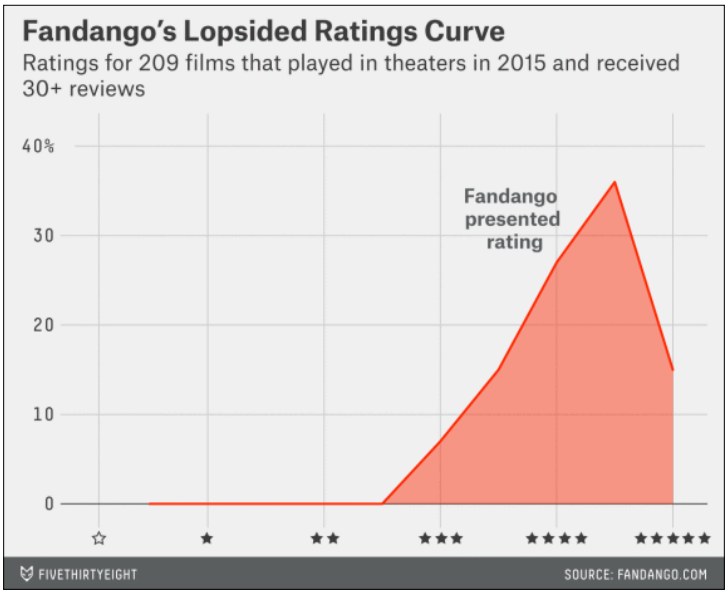


Figure 3: Fandango's lopsided ratings curve

(Source: `https://fivethirtyeight.com/features/fandango-movies-ratings/`)

Since the ratings from Fandango are unreliable, we will instead predict our own ratings based on IMDb ratings. More specifically, this is a multivariate regression problem, since our predictive model will use multiple features to make the rating prediction having many predictors.

Fortunately, the data is small enough to fit in memory, so plain batch learning should do just fine. Considering these factors and need, we will see that linear regression will meet our requirements. However, for more robust regression, you can still use deep neural network based regression techniques such as deep belief networks Regressor in *Chapter 7*, *Using Deep Neural Networks for Predictive Analytics*.

Using linear regression for movie rating prediction

Now, the first task is downloading the Fandango's rating dataset from GitHub at `https://github.com/fivethirtyeight/data/tree/master/fandango`. It contains every film that has a Rotten Tomatoes rating, an RT user rating, a Metacritic score, a Metacritic user score, IMDb score, and at least 30 fan reviews on Fandango.

The dataset has 22 columns that can be described as follows:

Column	Definition
`FILM`	Name of the film.
`RottenTomatoes`	Corresponding Tomatometer score for the film by Rotten Tomatoes.
`RottenTomatoes_User`	Rotten Tomatoes user score for the film.
`Metacritic`	Metacritic critic score for the film.
`Metacritic_User`	Metacritic user score for the film.
`IMDB`	IMDb user score for the film.
`Fandango_Stars`	A number of stars the film had on its Fandango movie page.
`Fandango_Ratingvalue`	The Fandango rating value for the film, as pulled from the HTML of each page. This is the actual average score the movie obtained.
`RT_norm`	Tomatometer score for the film. It is normalized to a 0 to 5 point system.
`RT_user_norm`	Rotten Tomatoes user score for the film. It is normalized to a 0 to 5 point system.
`Metacritic_norm`	The Metacritic critic scores for the film. It is normalized to a 0 to 5 point system.
`Metacritic_user_nom`	Metacritic user score for the film, normalized to a 0 to 5 point system.
`IMDB_norm`	IMDb user score for the film which is normalized to a 0 to 5 point system.
`RT_norm_round`	Rotten Tomatoes Tomatometer score for the film which is normalized to a 0 to 5 point system and rounded to the nearest half-star.
`RT_user_norm_round`	Rotten Tomatoes user score for the film, normalized to a 0 to 5 point system and rounded to the nearest half-star.
`Metacritic_norm_round`	Metacritic critic score for the film, normalized to a 0 to 5 point system and rounded to the nearest half-star.
`Metacritic_user_norm_round`	Metacritic user score for the film, normalized to a 0 to 5 point system and rounded to the nearest half-star.
`IMDB_norm_round`	IMDb user score for the film, normalized to a 0 to 5 point system and rounded to the nearest half-star.
`Metacritic_user_vote_count`	A number of user votes the film had on Metacritic.
`IMDB_user_vote_count`	A number of user votes the film had on IMDb.

`Fandango_votes`	A number of user votes the film had on Fandango.
`Fandango_ Difference`	Difference between the presented Fandango_Stars and the actual Fandango_Ratingvalue.

Table 1: description of the columns in the `fandango_score_comparison.csv`

We have already seen that a typical linear regression problem using TensorFlow has the following workflow that updates the parameters to minimize the given cost function of Fandango's lopsided rating curve:

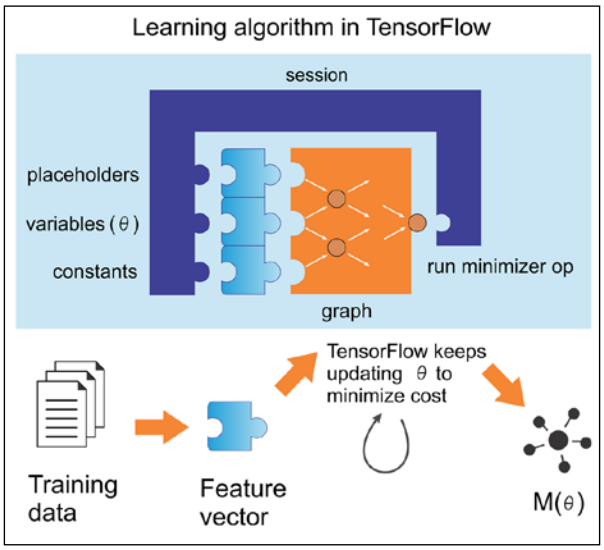


Figure 4: The learning algorithm using linear regression in TensorFlow

Now, let's try to follow the preceding figure and reproduce the same for the linear regression:

1. Import the required libraries:

```
import numpy as np
import pandas as pd
from scipy import stats
import sklearn
from sklearn.model_selection import train_test_split
import tensorflow as tf
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
```

2. Read the dataset and create a Panda DataFrame:

```
df = pd.read_csv('fandango_score_comparison.csv')
print(df.head())
```

The output is as follows:

Fandango_Ratingvalue	RT_norm	RT_user_norm	Metacritic_norm	Metacritic_user_nom	IMDB_norm	RT_norm_round
4.5	3.7	4.3	3.3	3.55	3.9	3.5
4.5	4.25	4	3.35	3.75	3.55	4.5
4.5	4	4.5	3.2	4.05	3.9	4
4.5	0.9	4.2	1.1	2.35	2.7	1
3	0.7	1.4	1.45	1.7	2.55	0.5
4	3.15	3.1	2.5	3.4	3.6	3
3.5	2.1	2.65	2.65	3.8	3.45	2
3.5	4.3	3.2	4.05	3.4	3.25	4.5
4	4.95	4.1	4.05	4.4	3.7	5
4	4.45	4.35	4	4.25	3.9	4.5

Figure 5: A snap of the dataset showing a typo in the `Metacritic_user_nom`

So, if you look at the preceding DataFrame carefully, there is a typo that could cause a disaster. From our intuition, it is clear that `Metacritic_user_nom` should have actually been `Metacritic_user_norm`. Let's rename it to avoid further confusion:

```
df.rename(columns={'Metacritic_user_nom':'Metacritic_user_norm'},
inplace=True)
```

Moreover, according to a statistical analysis at `https://fivethirtyeight.com/features/fandango-movies-ratings/`, all the variables don't contribute equally; the following columns have more importance in ranking the movies:

```
'Fandango_Stars',
'RT_user_norm',
'RT_norm',
'IMDB_norm',
'Metacritic_user_norm',
'Metacritic_norm'
```

Now we can check the correlation coefficients between variables before build the LR model. First, let's create a ranking list for that:

```
rankings_lst = ['Fandango_Stars',
                'RT_user_norm',
                'RT_norm',
                'IMDB_norm',
                'Metacritic_user_norm',
                'Metacritic_norm']
```

The following function computes the `Pearson` correlation coefficients and builds a full correlation matrix:

```
def my_heatmap(df):
    import seaborn as sns
    fig, axes = plt.subplots()
    sns.heatmap(df, annot=True)
    plt.show()
    plt.close()
```

Let's call the preceding method to plot the matrix as follows:

```
my_heatmap(df[rankings_lst].corr(method='pearson'))
```



Pearson correlation coefficients: A measure of the strength of the linear relationship between two variables. If the relationship between the variables is not linear, then the correlation coefficient cannot accurately and adequately represent the strength of the relationship between those two variables. It is often represented as "ρ" when measured on population and "r" when measured on a sample. Statistically, the range is -1 to 1, where -1 indicates a perfect negative linear relationship, an r of 0 indicates no linear relationship, and an r of 1 indicates a perfect positive linear relationship between variables.

The following correlation matrix shows correlation between considered features using the Pearson correlation coefficients:

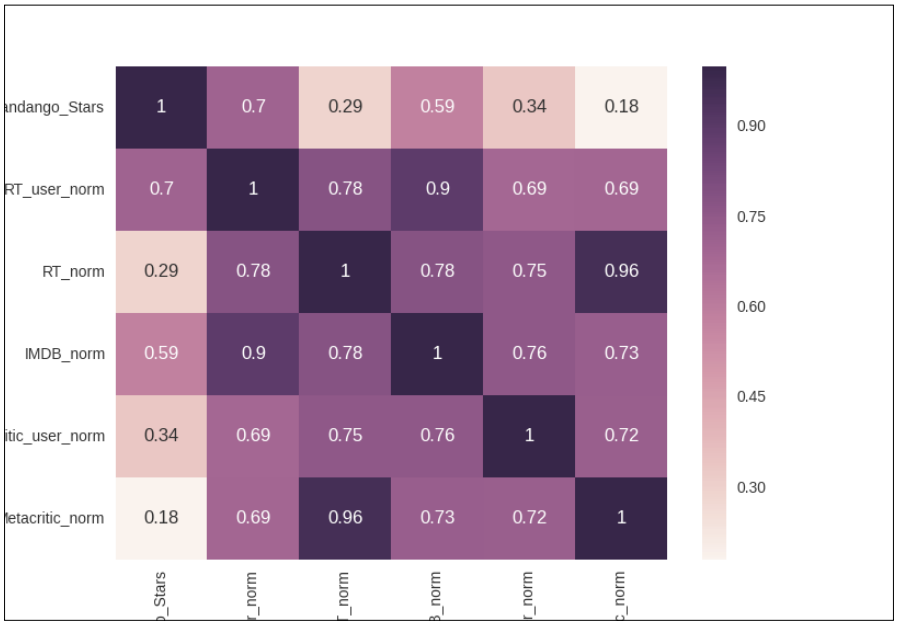


Figure 6: The correlation matrix on the ranking list movies

So, the correlation between Fandango and Metacritic is still positive. Now, let's do another study by considering only the movies for which RT has provided at least a 4-star rating:

```
RT_lst = df['RT_norm'] >= 4.
my_heatmap(df[RT_lst][rankings_lst].corr(method='pearson'))
>>>
```

The output is the correlation matrix on the ranked movies and RT movies having ratings of at least 4 showing a correlation between considered features using the Pearson correlation coefficients:

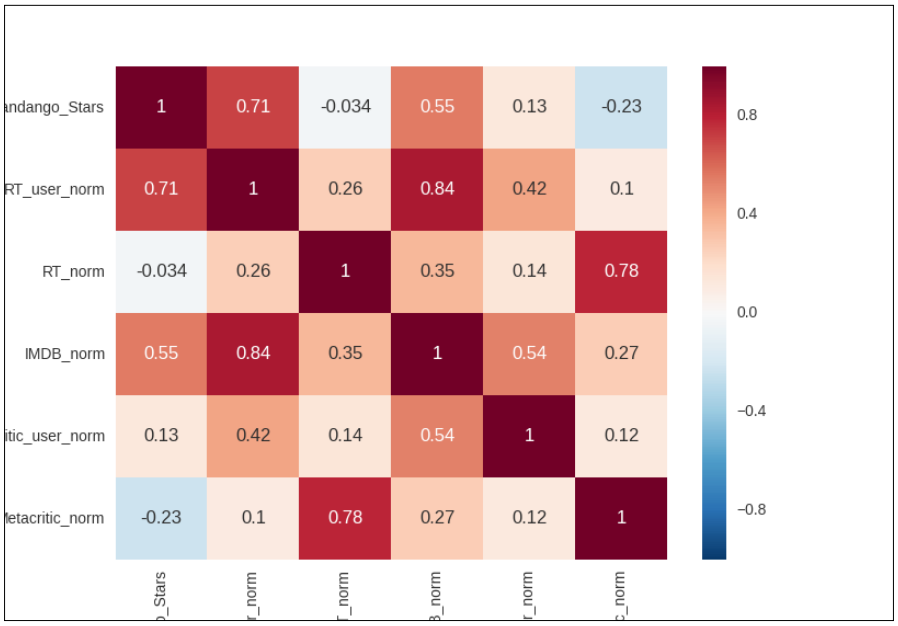


Figure 7: The correlation matrix on the ranked movies and RT movies having ratings at least 4

This time, we have obtained anticorrelation (that is, negative correlation) between Fandango and Metacritic, with the correlation coefficient-0.23. This means that the correlation of Metacritic in terms of Fandango is significantly biased toward high ratings.

Therefore, we can train our model without considering Fandango's rating, but before that let's build the LR model using this first. Later on, we will decide which option would produce a better result eventually.

3. Preparing the training and test sets.

   Let's create a feature matrix `X` by selecting two DataFrame columns:

```
feature_cols = ['Fandango_Stars', 'RT_user_norm', 'RT_norm',
'Metacritic_user_norm', 'Metacritic_norm']
X = df.loc[:, feature_cols]
```

   Here, I have used only the selected column as features and now we need to create a response vector `y`:

```
y = df['IMDB_norm']
```

We are assuming that the IMDB is the most reliable and the baseline source of ratings. Our ultimate target is to predict the rating of each movie and compare the predicted ratings with the response column `IMDB_norm`.

Now that we have the features and the response columns, it's time to split data into training and testing sets:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_
size=0.50, random_state=43)
```

If you want to change the `random_state`, it helps you generate pseudo-random numbers for a random sampling value to obtain different final results.



Random state: As the name sounds can be used for initializing the internal random number generator, which will decide the splitting of data into train and test indices. This also signifies that every time you run it without specifying `random_state`, you will get a different result, this is expected behavior. So, we can have the following three options:

- If `random_state` is None (or `np.random`), a randomly-initialized `RandomState` object is returned
- If `random_state` is an integer, it is used to seed a new `RandomState` object
- If `random_state` is a `RandomState` object, it is passed through

Now, we need to have the dimension of the dataset to be passed through the tensors:

```
dim = len(feature_cols)
```

We need to include an extra dimension for independent coefficient:

```
dim += 1
```

And so we need to create an extra column for the independent coefficient in the training set and test feature set as well:

```
X_train = X_train.assign( independent = pd.Series([1] * len(y_
train), index=X_train.index))
X_test = X_test.assign( independent = pd.Series([1] * len(y_
train), index=X_test.index))
```

So far, we have used and utilized the panda DataFrames but converting it into tensors is troublesome so instead let's convert them into a NumPy array:

```
P_train = X_train.as_matrix(columns=None)
P_test = X_test.as_matrix(columns=None)
```

```
q_train = np.array(y_train.values).reshape(-1,1)
q_test = np.array(y_test.values).reshape(-1,1)
```

4. Creating a place holder for TensorFlow.

   Now that we have all the training and test sets, before initializing these variables, we have to create the place holder for TensorFlow to feed the training sets across the tensors:

   ```
   P = tf.placeholder(tf.float32,[None,dim])
   q = tf.placeholder(tf.float32,[None,1])
   T = tf.Variable(tf.ones([dim,1]))
   ```

   Let's add some bias to differing from the value in the case where both types are quantized as follows:

   ```
   bias = tf.Variable(tf.constant(1.0, shape = [n_dim]))
   q_ = tf.add(tf.matmul(P, T),bias)
   ```

5. Creating an optimizer.

   Let's create an optimizer for the objective function:

   ```
   cost = tf.reduce_mean(tf.square(q_ - q))
   learning_rate = 0.0001
   training_op = tf.train.GradientDescentOptimizer(learning_
   rate=learning_rate).minimize(cost)
   ```

6. Initializing global variables:

   ```
   init_op = tf.global_variables_initializer()
   cost_history = np.empty(shape=[1],dtype=float)
   ```

7. Training the LR model.

   Here we are iterating the training 50,000 times and tracking several parameters, such as means square error that signifies how good the training is; we are keeping the cost history for future visualization, and so on:

   ```
   training_epochs = 50000
   with tf.Session() as sess:
       sess.run(init_op)
       cost_history = np.empty(shape=[1], dtype=float)
       t_history = np.empty(shape=[dim, 1], dtype=float)
       for epoch in range(training_epochs):
           sess.run(training_op, feed_dict={P: P_train, q: q_train})
           cost_history = np.append(cost_history, sess.run(cost,
   feed_dict={P: P_train, q: q_train}))
           t_history = np.append(t_history, sess.run(T, feed_dict={P:
   P_train, q: q_train}), axis=1)
   ```

```
    q_pred = sess.run(q_, feed_dict={P: P_test})[:, 0]
    mse = tf.reduce_mean(tf.square(q_pred - q_test))
    mse_temp = mse.eval()
    sess.close()
```

Finally, we evaluate the `mse` to get the scalar value out of the training evaluation on the test set. Now, let's compute the `mse` and `rmse` values, as follows:

```
print(mse_temp)
RMSE = math.sqrt(mse_temp)
print(RMSE)
>>>
0.425983107542
0.6526738140461913
```

You can also change the feature column, as follows:

```
feature_cols = ['RT_user_norm', 'RT_norm', 'Metacritic_user_norm',
'Metacritic_norm']
```

Now that we are not considering the Fandango's stars, I experienced the following result of `mse` and `rmse` respectively:

```
0.426362842426
0.6529646563375979
```

8. Observing the training cost throughout iterations:

```
fig, axes = plt.subplots()
plt.plot(range(len(cost_history)), cost_history)
axes.set_xlim(xmin=0.95)
axes.set_ylim(ymin=1.e-2)
axes.set_xscale("log", nonposx='clip')
axes.set_yscale("log", nonposy='clip')
axes.set_ylabel('Training cost')
axes.set_xlabel('Iterations')
axes.set_title('Learning rate = ' + str(learning_rate))
plt.show()
plt.close()
>>>
```

The output is as follows:

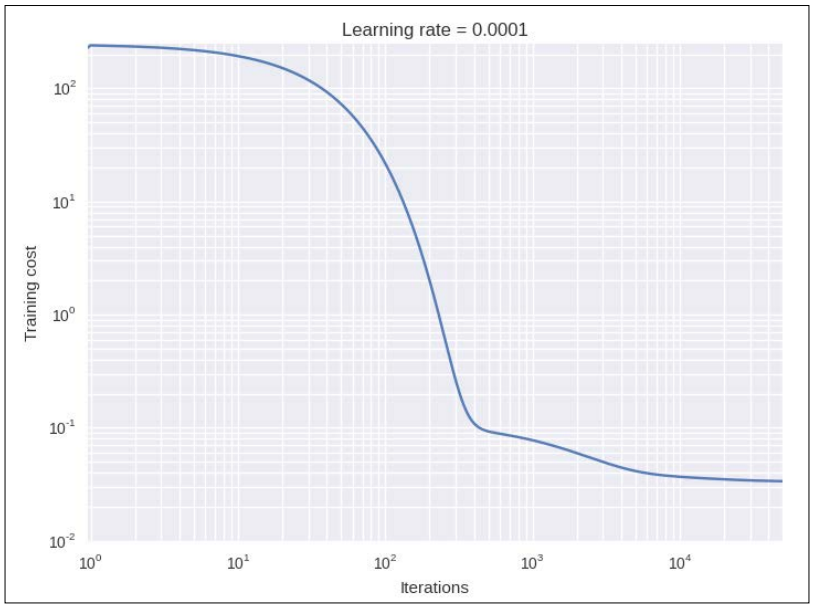


Figure 8: The training and training cost become saturated after 10000 iterations

The preceding graph shows that the training cost becomes saturated after 10,000 iterations. This also means that, even if you iterate the model more than 10,000 times, the cost is not going to experience a significant decrease.

9. Evaluating the model:

```
predictedDF = X_test.copy(deep=True)
predictedDF.insert(loc=0, column='IMDB_norm_predicted', value=pd.
Series(data=q_pred, index=predictedDF.index))
predictedDF.insert(loc=0, column='IMDB_norm_actual', value=q_test)

print('Predicted vs actual rating using LR with TensorFlow')
print(predictedDF[['IMDB_norm_actual', 'IMDB_norm_predicted']].
head())
print(predictedDF[['IMDB_norm_actual', 'IMDB_norm_predicted']].
tail())
>>>
```

The following shows the predicted versus actual rating using LR:

```
          IMDB_norm_actual  IMDB_norm_predicted
45                    3.30             3.232061
50                    3.35             3.381659
```

```
98              3.05            2.869175
119             3.60            3.796200
133             2.15            2.521702
140             4.30            4.033006
143             3.70            3.816177
42              4.10            3.996275
90              3.05            3.226954
40              3.45            3.509809
```

We can see that the prediction is a continuous value. Now it's time to see how well the LR model generalizes and fits to the regression line:

```
How the LR fit with the predicted data points:
plt.scatter(q_test, q_pred, color='blue', alpha=0.5)
plt.plot([q_test.min(), q_test.max()], [q_test.min(), q_test.
max()], '--', lw=1)
plt.title('Predicted vs Actual')
plt.xlabel('Actual')
plt.ylabel('Predicted')
plt.show()
plt.show()

>>>
```

The output is as follows:

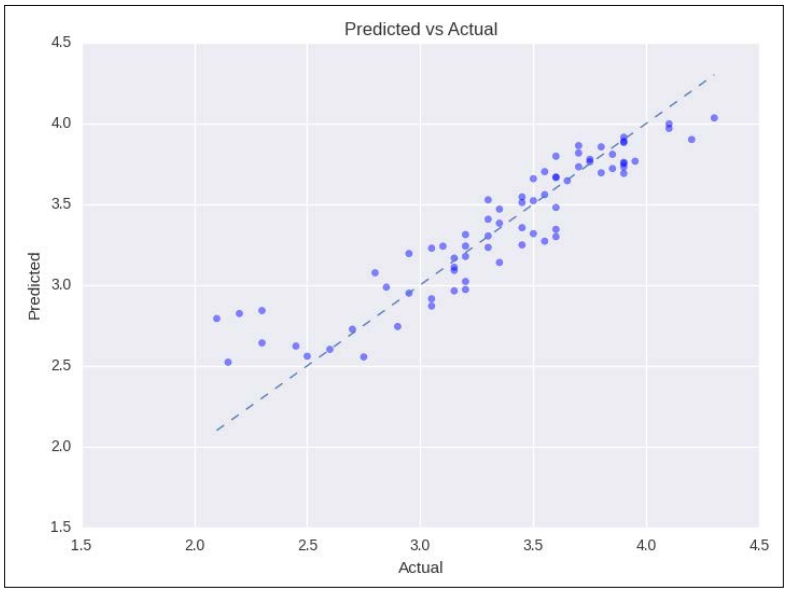


Figure 9: Prediction made by the LR model

The graph does not tell us that the prediction made by the LR model is good or bad. But we can still improve the performance of such models using layer architectures such as deep neural networks.

The next example is about applying other supervised learning algorithms such as logistic regression, support vector machines, and random forest for predictive analytics.

From disaster to decision - Titanic example revisited

In *Chapter 3, From Data to Decisions - Getting Started with TensorFlow*, we have seen a minimal data analysis of the Titanic dataset. Now it's our turn to do some analytics on top of the data. Let's look at what kinds of people survived the disaster.

Since we have enough data, but how could we do the predictive modeling so that we can draw some fairly straightforward conclusions from this data? For example, being a woman, being in first class, and being a child were all factors that could boost a passengers chances of survival during this disaster.

Using the brute-force approach such as if-else statements with some sort of weighted scoring system, you could write a program to predict whether a given passenger would survive the disaster. However, writing such a program in Python does not make much sense. Naturally, it would be very tedious to write, difficult to generalize, and would require extensive fine-tuning for each variable and samples (that is, each passenger):

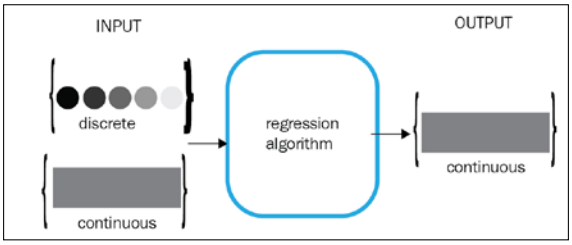


Figure 10: A regression algorithm is meant to produce continuous output

At this point, you might have confusion in your mind about what the basic difference between a classification and a regression problem is. Well, a regression algorithm is meant to produce continuous output. The input is allowed to be either discrete or continuous. In contrast, a classification algorithm is meant to produce discrete output from an input from a set of discrete or continuous values. This distinction is important to know because discrete-valued outputs are handled better by classification, which will be discussed in upcoming sections:

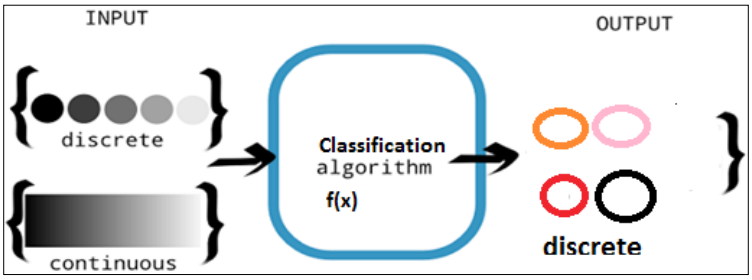


Figure 11: A classification algorithm is meant to produce discrete output

In this section, we will see how we could develop several predictive models for Titanic survival prediction and do some analytics using them. In particular, we will discuss logistic regression, random forest, and linear SVM. We start with logistic regression. Then we go with SVM since the number of features is not that large. Finally, we will see how we could improve the performance using Random Forests. However, before diving in too deeply, a short exploratory analysis of the dataset is required.

An exploratory analysis of the Titanic dataset

We will see how the variables contribute to survival. At first, we need to import the required packages:

```
import os
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
import shutil
```

Now, let's load the data and check what the features available to us are:

```
train = pd.read_csv(os.path.join('input', 'train.csv'))
test = pd.read_csv(os.path.join('input', 'test.csv'))
print("Information about the data")
print(train.info())
>>>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
PassengerId    891 non-null int64
Survived       891 non-null int64
Pclass         891 non-null int64
Name           891 non-null object
Sex            891 non-null object
Age            714 non-null float64
SibSp          891 non-null int64
Parch          891 non-null int64
Ticket         891 non-null object
Fare           891 non-null float64
Cabin          204 non-null object
Embarked       889 non-null object
```

So, the training dataset has `12` columns and `891` rows altogether. Also, the `Age`, `Cabin`, and `Embarked` columns have null or missing values. We will take care of the null values in the feature engineering section, but for the time being, let's see how many have survived:

```
print("How many have survived?")

print(train.Survived.value_counts(normalize=True))

count_plot = sns.countplot(train.Survived)

count_plot.get_figure().savefig("survived_count.png")

>>>
```

How many have survived?

```
0    0.616162
1    0.383838
```

So, approximately 61% died and only 39% of passengers managed to survive as shown in the following figure:

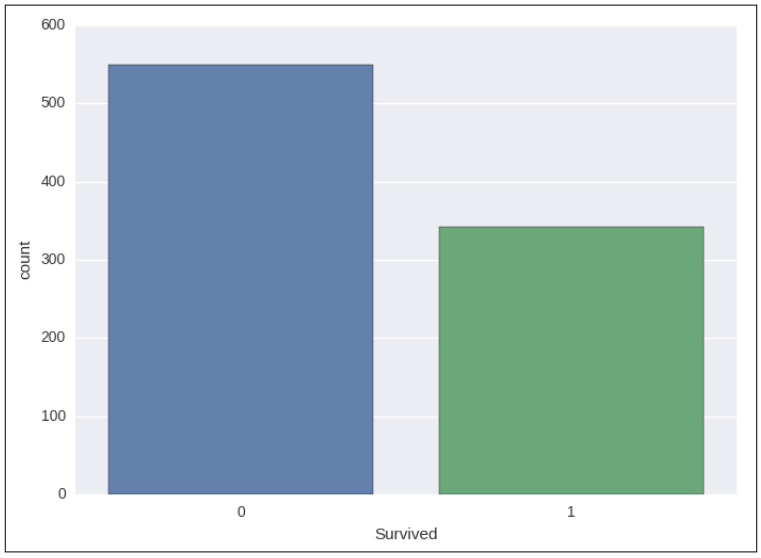


Figure 12: Survived versus dead from the Titanic training set

Now, what is the relationship between the class and the rate of survival? At first we should see the counts for each class:

```
train['Name_Title'] = train['Name'].apply(lambda x: x.split(',')[1]).
apply(lambda x: x.split()[0])
print('Title count')
print(train['Name_Title'].value_counts())
print('Survived by title')
print(train['Survived'].groupby(train['Name_Title']).mean())
>>>
Title      count
Mr.          517
Miss.        182
Mrs.         125
Master.       40
Dr.            7
Rev.           6
Mlle.          2
```

```
Col.           2
Major.         2
Sir.           1
Jonkheer.      1
Lady.          1
Capt.          1
the            1
Don.           1
Ms.            1
Mme.           1
```

As you may remember from the movie (that is, Titanic 1997), people from higher classes had better chances of surviving. So, you may assume that the title could be an important factor in survival, too. Another funny thing is that people with longer names have a higher probability of survival. This happens due to most of the people with longer names being married ladies whose husband or family members probably helped them to survive:

```
train['Name_Len'] = train['Name'].apply(lambda x: len(x))
print('Survived by name length')
print(train['Survived'].groupby(pd.qcut(train['Name_Len'],5)).mean())
>>>
Survived by name length
(11.999, 19.0]    0.220588
(19.0, 23.0]      0.301282
(23.0, 27.0]      0.319797
(27.0, 32.0]      0.442424
(32.0, 82.0]      0.674556
```

Women and children had a higher chance to survive, since they are the first to evacuate the shipwreck:

```
print('Survived by sex')
print(train['Survived'].groupby(train['Sex']).mean())
>>>
Survived by sex
Sex
female    0.742038
male      0.188908
```

Cabin has the most nulls (almost 700), but we can still extract information from it, like the first letter of each cabin. Therefore, we can see that most of the cabin letters are associated with survival rate:

```
train['Cabin_Letter'] = train['Cabin'].apply(lambda x: str(x)[0])
print('Survived by Cabin_Letter')
print(train['Survived'].groupby(train['Cabin_Letter']).mean())
>>>
Survived by Cabin_Letter
A    0.466667
B    0.744681
C    0.593220
D    0.757576
E    0.750000
F    0.615385
G    0.500000
T    0.000000
n    0.299854
```

Finally, it also seems that people who embarked at Cherbourg had a 20% higher survival rate than those embarked at other embarking locations. This is very likely due to the high percentage of upper-class passengers from that location:

```
print('Survived by Embarked')
print(train['Survived'].groupby(train['Embarked']).mean())
count_plot = sns.countplot(train['Embarked'], hue=train['Pclass'])
count_plot.get_figure().savefig("survived_count_by_embarked.png")

>>>
Survived by Embarked
C    0.553571
Q    0.389610
S    0.336957
```

Graphically, the preceding result can be seen as follows:

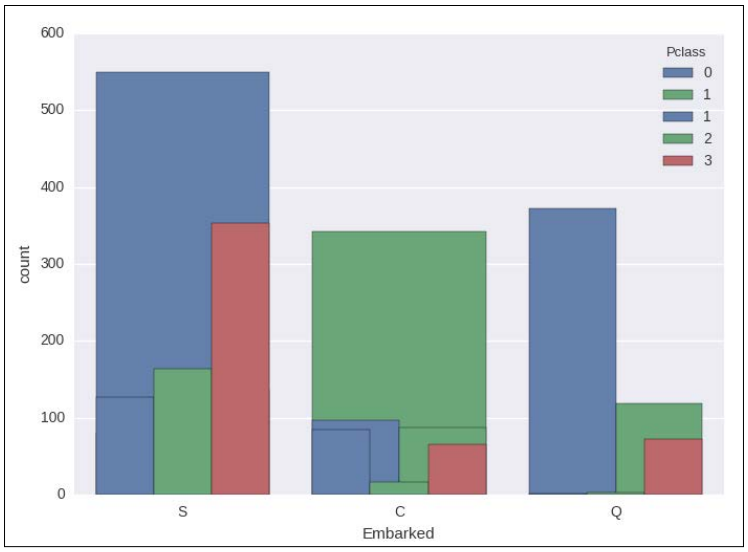


Figure 13: Survived by embarked

Thus, there were several important factors to people's survival. This means we need to consider these facts while developing our predictive models.

We will train several binary classifiers since this is a binary classification problem having two predictors, that is, 0 and 1 using the training set and will use the test set for making survival predictions.

But, before we even do that, let's do some feature engineering since you have seen that there are some missing or null values. We will either impute them or drop the entry from the training and test set. Moreover, we cannot use our datasets directly, but need to prepare them such that they could feed our machine learning models.

Feature engineering

Since we are considering the length of the passenger's name as an important feature, it would be better to remove the name itself and compute the corresponding length and also we extract only the title:

```
def create_name_feat(train, test):
    for i in [train, test]:
        i['Name_Len'] = i['Name'].apply(lambda x: len(x))
        i['Name_Title'] = i['Name'].apply(lambda x: x.split(',')[1]).
apply(lambda x: x.split()[0])
```

```
        del i['Name']
    return train, test
```

As there are 177 null values for Age, and those ones have a 10% lower survival rate than the non-nulls. Therefore, before imputing values for the nulls, we are including an Age_null flag, just to make sure we can account for this characteristic of the data:

```
def age_impute(train, test):
    for i in [train, test]:
        i['Age_Null_Flag'] = i['Age'].apply(lambda x: 1 if pd.isnull(x)
else 0)
        data = train.groupby(['Name_Title', 'Pclass'])['Age']
        i['Age'] = data.transform(lambda x: x.fillna(x.mean()))
    return train, test
```

We are imputing the null age values with the mean of that column. This will add some extra bias in the dataset. But, for the betterment of our predictive model, we will have to sacrifice something.

Then we combine the `SibSp` and `Parch` columns to create get family size and break it into three levels:

```
def fam_size(train, test):
    for i in [train, test]:
        i['Fam_Size'] = np.where((i['SibSp']+i['Parch']) == 0, 'One',
                                 np.where((i['SibSp']+i['Parch']) <= 3,
'Small', 'Big'))
        del i['SibSp']
        del i['Parch']
    return train, test
```

We are using the `Ticket` column to create `Ticket_Letr`, which indicates the first letter of each ticket and `Ticket_Len`, which indicates the length of the Ticket field:

```
def ticket_grouped(train, test):
    for i in [train, test]:
        i['Ticket_Letr'] = i['Ticket'].apply(lambda x: str(x)[0])
        i['Ticket_Letr'] = i['Ticket_Letr'].apply(lambda x: str(x))
        i['Ticket_Letr'] = np.where((i['Ticket_Letr']).isin(['1', '2',
'3', 'S', 'P', 'C', 'A']),
                                    i['Ticket_Letr'],
```

```
                                    np.where((i['Ticket_Letr']).
isin(['W', '4', '7', '6', 'L', '5', '8']),'Low_ticket', 'Other_ticket'))
        i['Ticket_Len'] = i['Ticket'].apply(lambda x: len(x))
        del i['Ticket']
    return train, test
```

We also need to extract the first letter of the `Cabin` column:

```
def cabin(train, test):
    for i in [train, test]:
        i['Cabin_Letter'] = i['Cabin'].apply(lambda x: str(x)[0])
        del i['Cabin']
    return train, test
```

Fill the null values in the `Embarked` column with the most commonly occurring value, which is `'S'`:

```
def embarked_impute(train, test):
    for i in [train, test]:
        i['Embarked'] = i['Embarked'].fillna('S')
    return train, test
```

We now need to convert our categorical columns. So far, we have considered it important for the predictive models that we will be creating to have numerical values for string variables. The `dummies()` function below does a one-hot encoding to the string variables:

```
def dummies(train, test,
            columns = ['Pclass', 'Sex', 'Embarked', 'Ticket_Letr',
'Cabin_Letter', 'Name_Title', 'Fam_Size']):
    for column in columns:
        train[column] = train[column].apply(lambda x: str(x))
        test[column] = test[column].apply(lambda x: str(x))
        good_cols = [column+'_'+i for i in train[column].unique() if i in
test[column].unique()]
        train = pd.concat((train, pd.get_dummies(train[column],
prefix=column)[good_cols]), axis=1)
        test = pd.concat((test, pd.get_dummies(test[column],
prefix=column)[good_cols]), axis=1)
        del train[column]
        del test[column]
    return train, test
```

We have the numerical features, finally, we need to create a separate column for the predicted values or targets:

```
def PrepareTarget(data):
    return np.array(data.Survived, dtype='int8').reshape(-1, 1)
```

We have seen the data and its characteristics and done some feature engineering to construct the best features for the linear models. The next task is to build the predictive models and make a prediction on the test set. Let's start with the logistic regression.

Logistic regression for survival prediction

Logistic regression is one of the most widely used classifiers to predict a binary response. It is a linear machine learning method as described in *Chapter 1*, *Basic Python and Linear Algebra for Predictive Analytics*. The loss function in the formulation given by the logistic loss:

$$L\left(\mathbf{w};\mathbf{x},y\right):=\log\left(1+\exp\left(-y\mathbf{w}^{T}\mathbf{x}\right)\right)$$

For the logistic regression model, the loss function is the logistic loss. For a binary classification problem, the algorithm outputs a binary logistic regression model such that, for a given new data point, denoted by *x*, the model makes predictions by applying the logistic function:

$$\mathrm{f}\left(z\right)=\frac{1}{1+e^{-z}}$$

In the preceding equation, $z=\mathrm{W}^{\mathrm{T}}\mathrm{X}$ *and* if $\mathrm{f}\left(\mathrm{W}^{\mathrm{T}}\mathrm{X}\right)>0.5$, the outcome is positive; otherwise, it is negative. Note that the raw output of the logistic regression model, *f (z)*, has a probabilistic interpretation.

Well, if you now compare logistic regression with its predecessor linear regression, the former provides you with a higher accuracy of the classification result. Moreover, it is a flexible way to regularize a model for custom adjustment and overall the model responses are measures of probability. And, most importantly, whereas linear regression can predict only continuous values, logistic regression can be generalized enough to make it predict discrete values. From now on, we will often be using the TensorFlow contrib API. So let's have a quick look at it.

Using TensorFlow contrib

The contrib is a high level API for learning with TensorFlow. It supports the following Estimators:

- `tf.contrib.learn.BaseEstimator`
- `tf.contrib.learn.Estimator`
- `tf.contrib.learn.Trainable`
- `tf.contrib.learn.Evaluable`
- `tf.contrib.learn.KMeansClustering`
- `tf.contrib.learn.ModeKeys`
- `tf.contrib.learn.ModelFnOps`
- `tf.contrib.learn.MetricSpec`
- `tf.contrib.learn.PredictionKey`
- `tf.contrib.learn.DNNClassifier`
- `tf.contrib.learn.DNNRegressor`
- `tf.contrib.learn.DNNLinearCombinedRegressor`
- `tf.contrib.learn.DNNLinearCombinedClassifier`
- `tf.contrib.learn.LinearClassifier`
- `tf.contrib.learn.LinearRegressor`
- `tf.contrib.learn.LogisticRegressor`

Thus, without developing the logistic regression, from scratch, we will use the estimator from the TensorFlow contrib package. When we are creating our own estimator from scratch, the constructor still accepts two high-level parameters for model configuration, `model_fn` and `params`:

```
nn = tf.contrib.learn.Estimator(
    model_fn=model_fn, params=model_params)
```

To instantiate an Estimator we need to provide two parameters such as `model_fn` and the `model_params` as follows:

```
nn = tf.contrib.learn.Estimator(model_fn=model_fn, params=model_params)
```

It is to be noted that the `model_fn()` function contains all the above mentioned TensorFlow logic to support the training, evaluation, and prediction. Thus, you only need to implement the functionality that could use it efficiently.

Now, upon invoking the `main()` method, `model_params` containing the learning rate, instantiates the Estimator. You can define the `model_params` as follows:

```
model_params = {"learning_rate": LEARNING_RATE}
```



For more information on the TensorFlow contrib, interested readers can refer to this URL at:

`https://www.tensorflow.org/extend/estimators`

Well, so far we have acquired enough background knowledge to create an LR model with TensorFlow with our dataset. It's time to implement it:

1. Import required packages and modules:

```
import os
import shutil
import random
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from feature import *
import tensorflow as tf
from tensorflow.contrib.learn.python.learn.estimators import
estimator
from tensorflow.contrib import learn
```

2. Loading and preparing the dataset.

   At first, we load both the datasets:

```
random.seed(12345) # For the reproducibility
train = pd.read_csv(os.path.join('input', 'train.csv'))
test = pd.read_csv(os.path.join('input', 'test.csv'))
```

   Let's do some feature engineering. We will invoke the function we defined in the feature engineering section, but will be provided as separate Python script with name `feature.py`:

```
train, test = create_name_feat(train, test)
train, test = age_impute(train, test)
train, test = cabin(train, test)
train, test = embarked_impute(train, test)
```

```
train, test = fam_size(train, test)
test['Fare'].fillna(train['Fare'].mean(), inplace=True)
train, test = ticket_grouped(train, test)
```

It is to be noted that the sequence of the above invocation is important to make the training and test set consistent. Now, we also need to create numerical values for categorical variables using the `dummies()` function from sklearn:

```
train, test = dummies(train, test, columns=['Pclass', 'Sex',
'Embarked', 'Ticket_Letr', 'Cabin_Letter', 'Name_Title', 'Fam_
Size'])
```

We need to prepare the training and test set:

```
TEST = True
if TEST:
    train, test = train_test_split(train, test_size=0.25, random_
state=10)
    train = train.sort_values('PassengerId')
    test = test.sort_values('PassengerId')

X_train = train.iloc[:, 1:]
x_test = test.iloc[:, 1:]
```

We then convert the training and test set into a NumPy array since so far we have kept them in Pandas DataFrame format:

```
x_train = np.array(x_train.iloc[:, 1:], dtype='float32')
if TEST:
    x_test = np.array(x_test.iloc[:, 1:], dtype='float32')
else:
    x_test = np.array(x_test, dtype='float32')
```

Let's prepare the target column for prediction:

```
y_train = PrepareTarget(train)
```

We also need to know the feature count to build the LR estimator:

```
feature_count = x_train.shape[1]
```

3. Preparing the LR estimator.

   We build the LR estimator. We will utilize the `LinearClassfier` estimator for it. Since this is a binary classification problem, we provide two classes:

```
def build_lr_estimator(model_dir, feature_count):
    return estimator.SKCompat(learn.LinearClassifier(
```

```
        feature_columns=[tf.contrib.layers.real_valued_column("",
dimension=feature_count)],
        n_classes=2, model_dir=model_dir))
```

4. Training the model.

   Here, we train the above LR estimator for `10,000` iterations. The `fit()` method does the trick and the `predict()` method computes the prediction on the training set containing the feature, that is, `X_train` and the label, that is, `y_train`:

```
print("Training...")
try:
    shutil.rmtree('lr/')
except OSError:
    pass
lr = build_lr_estimator('lr/', feature_count)
lr.fit(x_train, y_train, steps=1000)
lr_pred = lr.predict(x_test)
lr_pred = lr_pred['classes']
```

5. Model evaluation.

   We will evaluate the model seeing several classification performance metrics such as precision, recall, f1 score, and confusion matrix:

```
if TEST:
    target_names = ['Not Survived', 'Survived']
    print("Logistic Regression Report")
    print(classification_report(test['Survived'], lr_pred, target_
names=target_names))
    print("Logistic Regression Confusion Matrix")

>>>
Logistic Regression Report
                  precision    recall  f1-score   support
Not Survived       0.90         0.88      0.89       147
Survived           0.78         0.80      0.79        76
-------------------------------------------------------------
 avg / total       0.86         0.86       0.86       223
```

Since we trained the LR model with NumPy data, we now need to convert it back to a Panda DataFrame for confusion matrix creation:

```
cm = confusion_matrix(test['Survived'], lr_pred)
    df_cm = pd.DataFrame(cm, index=[i for i in ['Not Survived',
'Survived']],
                         columns=[i for i in ['Not Survived',
'Survived']])
    print(df_cm)

>>>
Logistic Regression Confusion Matrix
              Not Survived  Survived
Not Survived           130        17
Survived               15         61
```

Now, let's see the count:

```
print("Predicted Counts")
print(sol.Survived.value_counts())

>>>
Predicted Counts
0    145
1     78
```

Since seeing the count graphically is awesome, let's draw it:

```
sol = pd.DataFrame()
sol['PassengerId'] = test['PassengerId']
sol['Survived'] = pd.Series(lr_pred.reshape(-1)).map({True:1,
False:0}).values
sns.plt.suptitle("Predicted Survived LR")
count_plot = sns.countplot(sol.Survived)
count_plot.get_figure().savefig("survived_count_lr_prd.png")

>>>
```

The output is as follows:

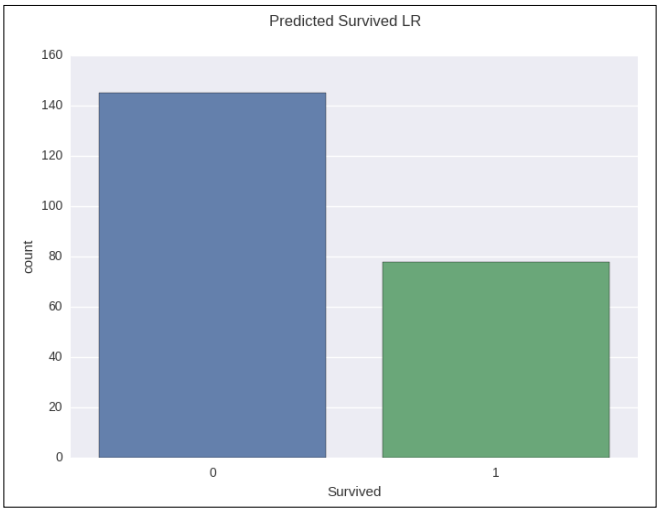


Figure 14: Survival prediction using logistic regression with TensorFlow

So, the accuracy we achieved with the LR model is 86% which is not that bad at all. But it can still be improved with better predictive models. In the next section, we will try to do that using linear **SVM** for survival prediction.

Linear SVM for survival prediction

The linear **SVM** is one of the most widely used and standard methods for large-scale classification tasks. Both the multiclass and binary classification problem can be solved using SVM with the loss function in the formulation given by the hinge loss:

$$L\left(\mathrm{w};\mathrm{x},\mathrm{y}\right) := \max\left\{0, 1-\mathrm{y}\mathrm{w}^{T}\mathrm{x}\right\}$$

Usually, linear SVMs are trained with L2 regularization. Eventually, the linear SVM algorithm outputs an SVM model that can be used to predict the label of unknown data.

Suppose you have an unknown data point, *x*, the SVM model makes predictions based on the value of $\mathrm{W}^{\mathrm{T}}\mathrm{X}$. The outcome can be either positive or negative. More specifically, if $\mathrm{W}^{\mathrm{T}}\mathrm{X} \geq 0$, then the predicted value is positive; otherwise, it is negative.

The current version of the TensorFlow contrib package supports only the linear SVM. TensorFlow uses SDCAOptimizer for the underlying optimization. Now, the thing is that if you want to build an SVM model of your own, you need to consider the performance and convergence tuning issues. Fortunately, you can pass the `num_loss_partitions` parameter to the SDCAOptimizer function. But you need to set the *X* such that it converges to the concurrent train ops per worker.

If you set the `num_loss_partitions` larger than or equal to this value, convergence is guaranteed, but this makes the overall training slower with the increase of `num_loss_partitions`. On the other hand, if you set its value to a smaller one, the optimizer is more aggressive in reducing the global loss, but convergence is not guaranteed.



For more on the implemented contrib packages, interested readers should refer to this URL at:

`https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/learn/python/learn/estimators`.

Well, so far we have acquired enough background knowledge for creating an SVM model, now it's time to implement it:

1. Import the required packages and modules:

```
import os
import shutil
import random
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from feature import *
import tensorflow as tf
from tensorflow.contrib.learn.python.learn.estimators import svm
```

2. Dataset preparation for building SVM model:

   Now, the data preparation for building an SVM model is more or less the same as an LR model, except that we need to convert the `PassengerId` to string which is required for the SVM:

```
train['PassengerId'] = train['PassengerId'].astype(str)
test['PassengerId'] = test['PassengerId'].astype(str)
```

3. Creating a dictionary for SVM for continuous feature column.



To feed the data to the SVM model, we further need to create a dictionary mapping from each continuous feature column name (`k`) to the values of that column stored in a constant Tensor. For more information on this issue, refer to this issue on TensorFlow GitHub repository at:

`https://github.com/tensorflow/tensorflow/issues/9505`.

I have written two functions for both the feature and labels. Let's see what the first one looks like:

```
def train_input_fn():
    continuous_cols = {k: tf.expand_dims(tf.constant(train[k].
values), 1)
                       for k in list(train) if k not in
['Survived', 'PassengerId']}
    id_col = {'PassengerId' : tf.constant(train['PassengerId'].
values)}
    feature_cols = continuous_cols.copy()
    feature_cols.update(id_col)
    label = tf.constant(train["Survived"].values)
    return feature_cols, label
```

The preceding function creates a dictionary mapping from each continuous feature column and then another for the `passengerId` column. Then I merged them into one. Since we want to target the `'Survived'` column as the labels, I converted the label column into constant tensor. Finally, through this function, I returned both the feature column and the label.

Now, the second method does almost the same trick except that it returns only the feature columns as follows:

```
def predict_input_fn():
    continuous_cols = {k: tf.expand_dims(tf.constant(test[k].
values), 1)
                       for k in list(test) if k not in
['Survived', 'PassengerId']}
    id_col = {'PassengerId' : tf.constant(test['PassengerId'].
values)}
    feature_cols = continuous_cols.copy()
    feature_cols.update(id_col)
    return feature_cols
```

4. Training the SVM model.

   Now we will iterate the training 10,000 times over the real valued column only. Finally, it creates a prediction list containing all the prediction values:

```
svm_model = svm.SVM(example_id_column="PassengerId",
                    feature_columns=[tf.contrib.layers.real_
valued_column(k) for k in list(train)
                                     if k not in ['Survived',
'PassengerId']],
                    model_dir="svm/")
svm_model.fit(input_fn=train_input_fn, steps=10000)
svm_pred = list(svm_model.predict_classes(input_fn=predict_input_
fn))
```

5. Evaluation of the model:

```
target_names = ['Not Survived', 'Survived']
print("SVM Report")
print(classification_report(test['Survived'], svm_pred, target_
names=target_names))

>>>
SVM Report
                      precision    recall  f1-score   support
Not Survived       0.94         0.72      0.82        117
Survived           0.63         0.92      0.75         62
-------------------------------------------------------------
 avg / total       0.84          0.79      0.79        179
```

   Thus using SVM, the accuracy is only 79%, which is lower than that of an LR model. Well, similar to an LR model, draw and observe the confusion matrix:

```
print("SVM Confusion Matrix")
cm = confusion_matrix(test['Survived'], svm_pred)
df_cm = pd.DataFrame(cm, index=[i for i in ['Not Survived',
'Survived']],
                         columns=[i for i in ['Not Survived',
'Survived']])
print(df_cm)
>>>
SVM Confusion Matrix
              Not Survived  Survived
Not Survived             84        33
Survived                  5         57
```

Then, let's draw the count plot to see the ratio visually:

```
sol = pd.DataFrame()
sol['PassengerId'] = test['PassengerId']
sol['Survived'] = pd.Series(svm_pred).values
sns.plt.suptitle("Titanic Survival prediction using SVM with
TensorFlow")
count_plot = sns.countplot(sol.Survived)
```

The output is as follows:



Figure 15: Survival prediction using linear SVM with TensorFlow

Now, the count:

```
print("Predicted Counts")
print(sol.Survived.value_counts())

>>>
Predicted Counts
1    90
0    89
```

Ensemble method for survival prediction: random forest

One of the most widely used machine learning techniques is using the ensemble methods, which are learning algorithms that construct a set of classifiers. It can then be used to classify new data points by taking a weighted vote of their predictions. In this section, we will mainly focus on the random forest that can be built by combining 100s of decision trees.

Decision trees (**DTs**) is a technique which is used in supervised learning for solving classification and regression tasks. Where a DT model learns simple decision rules that are inferred from the data features by utilizing a tree-like graph to demonstrate the course of actions. Each branch of a decision tree represents a possible decision, occurrence or reaction in terms of statistical probability:

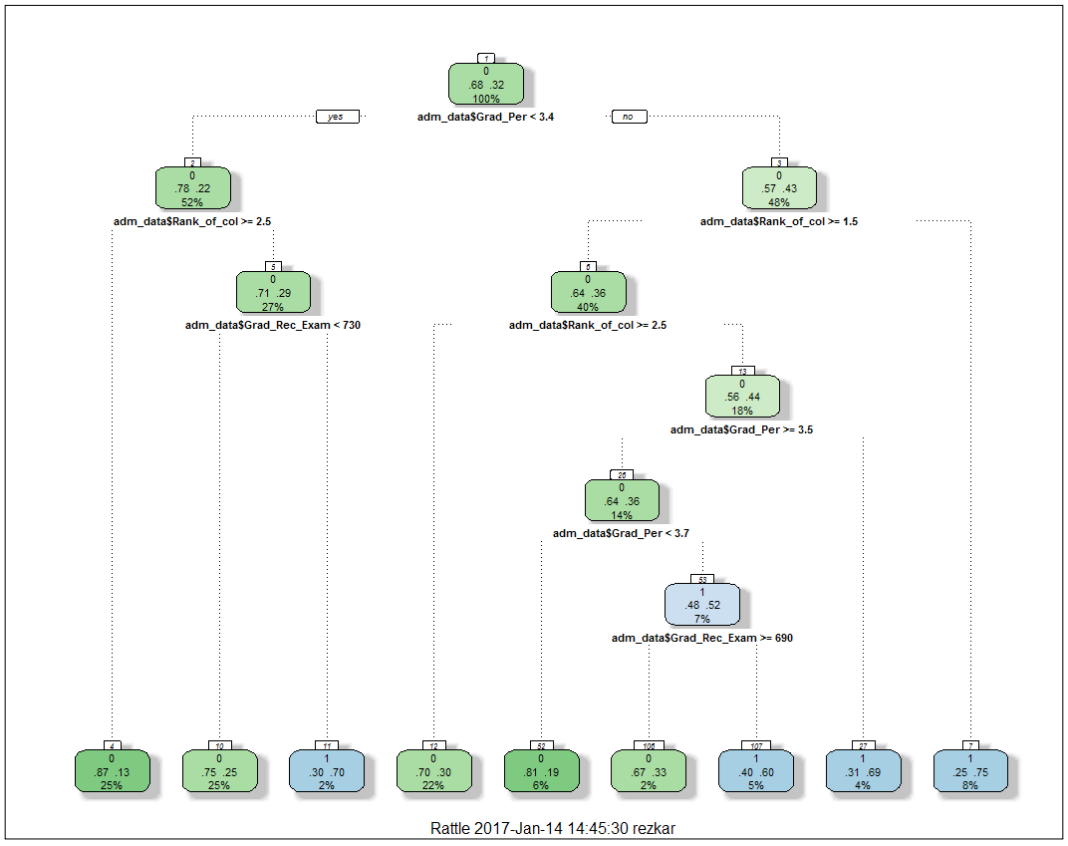


Figure 16: A sample decision tree on the admission test dataset using the rattle package of R

Compared to LR or SVM, the DTs are far more robust classification algorithms. The tree infers predicted labels or classes after splitting available features to the training data based to produce a good generalization. Most interestingly, the algorithm can handle both the binary as well as multiclass classification problems.

For instance, the decision trees in figure 16 learn from the admission data to approximate a sine curve with a set of `if...else` decision rules. The dataset contains the record of each student who applied for admission, say to an American university. Each record contains the graduate record exam score, CGPA score and the rank of the column. Now we will have to predict who is competent based on these three features (variables).

DTs can be utilized to solve this kind of problem after training the DT model and pruning the unwanted branch of the tree. In general, a deeper tree signifies more complex decision rules and a better-fitted model. Therefore, the deeper the tree, the more complex the decision rules, and the more fitted the model.



If you would like to draw the above figure, just use my R script and execute on RStudio and feed the admission dataset. The script and the dataset can be found in my GitHub repository at:

`https://github.com/rezacsedu/AdmissionUsingDecisionTree`.

Well, so far we have acquired enough background knowledge for creating a Random Forest (RF) model, now it's time to implement it.

1. Import the required packages and modules:

```
import os
import shutil
import random
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from feature import *
import tensorflow as tf
from tensorflow.contrib.learn.python.learn.estimators import
estimator
from tensorflow.contrib.tensor_forest.client import random_forest
from tensorflow.contrib.tensor_forest.python import tensor_forest
```

2. Dataset preparation for building an RF model.

   Now, the data preparation for building an RF model is more or less the same as an LR model. So please refer to the logistic regression section.

3. Building a random forest estimator.

   The following function builds a random forest estimator. It creates 1,000 trees with maximum 1,000 nodes and 10-fold cross-validation. Since it's a binary classification problem, I put number of classes as `2`:

```
def build_rf_estimator(model_dir, feature_count):
    params = tensor_forest.ForestHParams(
        num_classes=2,
        num_features=feature_count,
        num_trees=1000,
        max_nodes=1000,
        min_split_samples=10)
    graph_builder_class = tensor_forest.RandomForestGraphs
    return estimator.SKCompat(random_forest.TensorForestEstimator(
        params, graph_builder_class=graph_builder_class,
        model_dir=model_dir))
```

4. Training the RF model.

   Here, we train the above RF estimator. Once the `fit()` method does the trick and the `predict()` method computes the prediction on the training set containing the feature, that is, `x_train` and the label, that is, `y_train`:

```
rf = build_rf_estimator('rf/', feature_count)
rf.fit(x_train, y_train, batch_size=100)
rf_pred = rf.predict(x_test)
rf_pred = rf_pred['classes']
```

5. Evaluating the model.

   Now let's evaluate the performance of the RF model:

```
    target_names = ['Not Survived', 'Survived']
    print("RandomForest Report")
    print(classification_report(test['Survived'], rf_pred, target_
names=target_names))

>>>
```

```
RandomForest Report
                         precision    recall  f1-score   support
-------------------------------------------------------
Not Survived       0.92         0.85       0.88            117
Survived           0.76         0.85       0.80            62
-------------------------------------------------------
avg / total        0.86         0.85       0.86            179
```

Thus, using RF, the accuracy is 87% which is higher than that of the LR and SVM models. Well, similar to the LR and SVM model, we'll draw and observe the confusion matrix:

```
    print("Random Forest Confusion Matrix")
    cm = confusion_matrix(test['Survived'], rf_pred)
    df_cm = pd.DataFrame(cm, index=[i for i in ['Not Survived',
'Survived']],
                         columns=[i for i in ['Not Survived',
'Survived']])
    print(df_cm)
>>>
Random Forest Confusion Matrix
                       Not Survived  Survived
-------------------------------------------------------
Not Survived            100             17
Survived                 9              53
```

Then, let's draw the count plot to see the ratio visually:

```
sol = pd.DataFrame()
sol['PassengerId'] = test['PassengerId']
sol['Survived'] = pd.Series(svm_pred).values
sns.plt.suptitle("Titanic Survival prediction using RF with
TensorFlow")
count_plot = sns.countplot(sol.Survived)
```

The output is as follows:

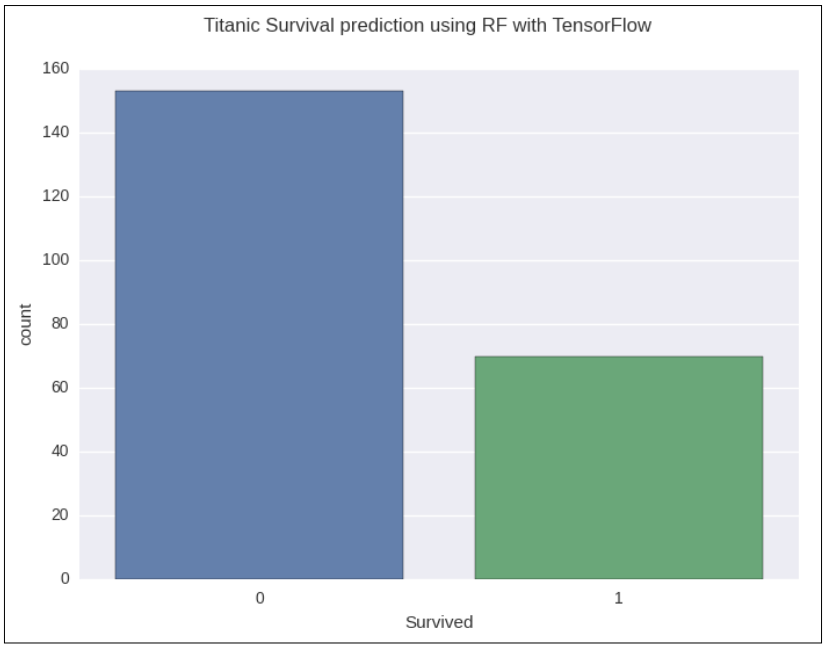


Figure 17: Titanic survival prediction using random forest with TensorFlow

Now, the count for each one:

```
print("Predicted Counts")
print(sol.Survived.value_counts())

>>>
Predicted Counts
-------------------------
0    109
1     70
```

A comparative analysis

From the classification reports, we can see that random forest has the best overall performance. The reason for this may be that it works better with categorical features than the other two methods. Also, since it uses implicit feature selection, overfitting was reduced significantly. Using logistic regression is a convenient probability score for observations. However, it doesn't perform well when feature space is too large that is, doesn't handle a large number of categorical features/variables well. It also solely relies on transformations for non-linear features.

Finally, using SVM we can handle a large feature space with non-linear feature interactions without relying on the entire dataset. However, it is not very well with a large number of observations. Nevertheless, it can be tricky to find an appropriate kernel sometimes.

Summary

In this chapter, we have discussed supervised learning from the theoretical and practical perspective. In particular, we have revisited the linear regression model for regression analysis. We have seen how to use regression for predicting continuous values. Later in this chapter, we have discussed some other supervised learning algorithms for predictive analytics. We have seen how to use logistic regression, SVM, and random forests for survival prediction on the Titanic dataset. Finally, we have seen a comparative analysis between these classifiers. We have also seen that random forest, which is based on decision trees ensembles, outperforms logistic regression and linear SVM models.

In *Chapter 5*, *Clustering Your Data - Unsupervised Learning for Predictive Analytics*, we will provide some practical examples of unsupervised learning. Particularly, the clustering technique using TensorFlow will be provided for neighborhood clustering and audio clustering from audio features.

More specifically, we will provide an exploratory analysis of the dataset then we will develop a cluster of the neighborhood using K-means, K-NN, and bisecting K-means with sufficient performance metrics such as cluster cost, accuracy, and so on. In the second part of the chapter, we will see how to do audio feature clustering. Finally, we will provide a comparative analysis of clustering algorithms.

5
Clustering Your Data - Unsupervised Learning for Predictive Analytics

In this chapter, we will dig deeper into predictive analytics and find out how we can take advantage of it to cluster records belonging to a certain group or class for a dataset of unsupervised observations. We will provide some practical examples of unsupervised learning; in particular, clustering techniques using TensorFlow will be discussed with some hands-on examples.

The following topics will be covered in this chapter:

- Unsupervised learning and clustering
- Using K-means for predicting neighborhood
- Using K-means for clustering audio files
- Using unsupervised **k-nearest neighborhood** (**kNN**) for predicting nearest neighbors

Unsupervised learning and clustering

In this section, we will provide a brief introduction to the unsupervised **machine learning** (**ML**) technique. Unsupervised learning is a type of ML algorithm used for grouping related data objects and finding hidden patterns by inferencing from unlabeled datasets, that is, a training set consisting of input data without labels.

Let's see a real-life example. Suppose you have a large collection of not-pirated-totally-legal MP3s in a crowded and massive folder on your hard drive. Now, what if you can build a predictive model that helps automatically group together similar songs and organize them into your favorite categories such as country, rap, and rock?

This is an act of assigning an item to a group so that an MP3 is added to the respective playlist in an unsupervised way. In *Chapter 3*, *From Data to Decisions - Getting Started with TensorFlow*, on classification, we assumed that you're given a training dataset of correctly labeled data. Unfortunately, we don't always have that extravagance when we collect data in the real world. For example, suppose we would like to divide a huge collection of music into interesting playlists. How can we possibly group together songs if we don't have direct access to their metadata? One possible approach is a mixture of various ML techniques, but clustering is often at the heart of the solution.

In other words, the main objective of the unsupervised learning algorithms is to explore the unknown/hidden patterns in the input data that are unlabeled. Unsupervised learning, however, also comprehends other techniques to explain the key features of the data in an exploratory way toward finding the hidden patterns. To overcome this challenge, clustering techniques are used widely to group unlabeled data points based on certain similarity measures in an unsupervised way.

In a clustering task, an algorithm groups related features into categories by analyzing similarities between input examples, where similar features are clustered and marked using circles.

Clustering uses include but are not limited to the following points:

- Anomaly detection for suspicious pattern finding in an unsupervised way
- Text categorization for finding useful patterns in the tests for NLP
- Social network analysis for finding coherent groups
- Data center computing clusters for finding a way of putting related computers together
- Real estate data analysis for identifying neighborhoods based on similar features

Clustering analysis is about dividing data samples or data points and putting them into corresponding homogeneous classes or clusters. Thus, a trivial definition of clustering can be thought of as the process of organizing objects into groups whose members are similar in some way, as shown in figure 1:

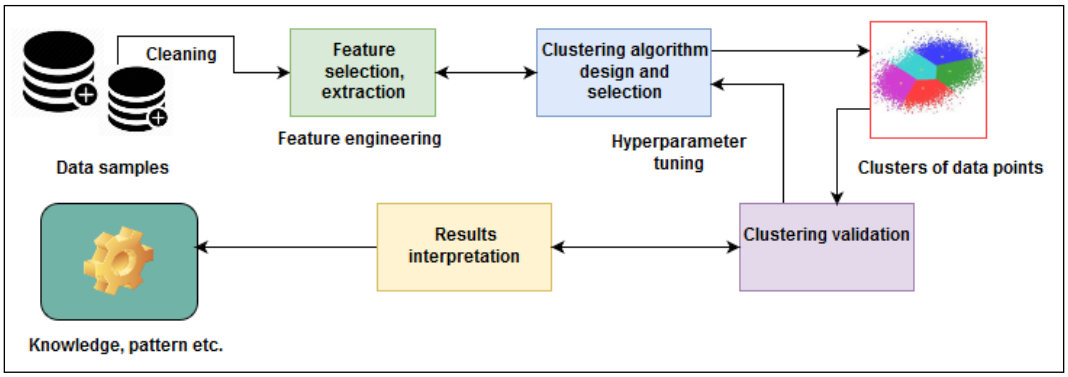


Figure 1: A typical data pipelines for clustering raw data

A cluster is, therefore, a collection of objects that have a similarity between them and are dissimilar to the objects belonging to other clusters. If a collection of objects is provided, clustering algorithms put these objects into groups based on similarity. For example, a clustering algorithm such as K-means locates the centroid of the groups of data points.

However, to make clustering accurate and effective, the algorithm evaluates the distance between each point from the centroid of the cluster. Eventually, the goal of clustering is to determine intrinsic grouping in a set of unlabeled data. For example, the K-means algorithm tries to cluster related data points within the predefined **3** (that is $k = 3$) clusters, as shown in figure 2:

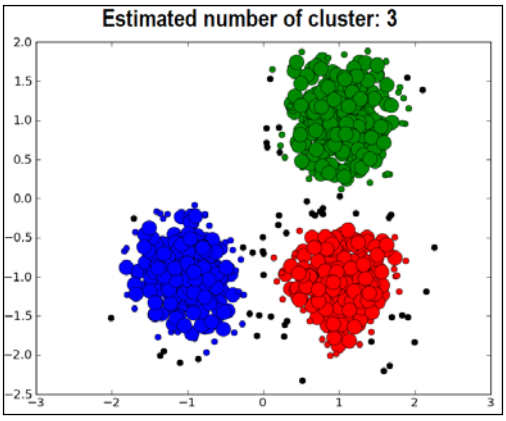


Figure 2: The results of a typical clustering algorithm and a representation of the cluster centers

Clustering is a process of intelligently categorizing items in your dataset. The overall idea is that two items in the same cluster are closer to each other than items that belong to separate clusters. This is a general definition, leaving the interpretation of closeness open. For example, perhaps cheetahs and leopards belong to the same cluster, whereas elephants belong to another when closeness is measured by the similarity of two species in the hierarchy of biological classification (family, genus, and species).

Using K-means for predictive analytics

K-means is a clustering algorithm that tries to cluster related data points together. However, we should know its working principle and mathematical operations.

How K-means works

Suppose we have n data points, x_i, $i = 1...n$, that need to be partitioned into `k` clusters. Now that the target here is to assign a cluster to each data point, K-means aims to find the positions, μ_i, $i=1...k$, of the clusters that minimize the distance from the data points to the cluster. Mathematically, the K-means algorithm tries to achieve the goal by solving an equation that is an optimization problem:

$$\arg\min_{\mathbf{c}} \sum_{i=1}^{k} \sum_{\mathbf{x} \in \boldsymbol{c}_i} d\left(\mathbf{x}, \mu_i\right) = \arg\min_{\mathbf{c}} \sum_{i=1}^{k} \sum_{\mathbf{x} \in \boldsymbol{c}_i} \left\| \mathbf{x} - \mu_i \right\|_2^2$$

In the previous equation, c_i is a set of data points, which when assigned to cluster i and $d\left(\mathbf{x}, \mu_i\right) = \left\| \mathbf{x} - \mu_i \right\|_2^2$ is the Euclidean distance to be calculated (we will explain why we should use this distance measurement shortly). Therefore, we can see that the overall clustering operation using K-means is not a trivial one, but a NP-hard optimization problem. This also means that the K-means algorithm not only tries to find the global minima but often gets stuck in different solutions.

Clustering using the K-means algorithm begins by initializing all the coordinates to the centroids. With every pass of the algorithm, each point is assigned to its nearest centroid based on some distance metric, usually the Euclidean distance stated earlier.



Distance calculation: There are other ways to calculate the distance as well. For example, the Chebyshev distance can be used to measure the distance by considering only the most notable dimensions. The Hamming distance algorithm can identify the difference between two strings. Mahalanobis distance can be used to normalize the covariance matrix. The Manhattan distance is used to measure the distance by considering only axis-aligned directions. The Haversine distance is used to measure the great-circle distances between two points on a sphere from the location.

Considering these distance-measuring algorithms, it is clear that the Euclidean distance algorithm would be the most appropriate to solve our purpose of distance calculation in the K-means algorithm. The centroids are then updated to be the centers of all the points assigned to it in that iteration. This repeats until there is a minimal change in the centers. In short, the K-means algorithm is an iterative algorithm and works in two steps:

1. **Cluster assignment step**: K-means goes through each of the *n* data points in the dataset that is assigned to a cluster closest to the k centroids, then the least distant one is picked.
2. **Update step**: For each cluster, a new centroid is calculated for all the data points in the cluster. The overall workflow of K-means can be explained using a flowchart, as follows:

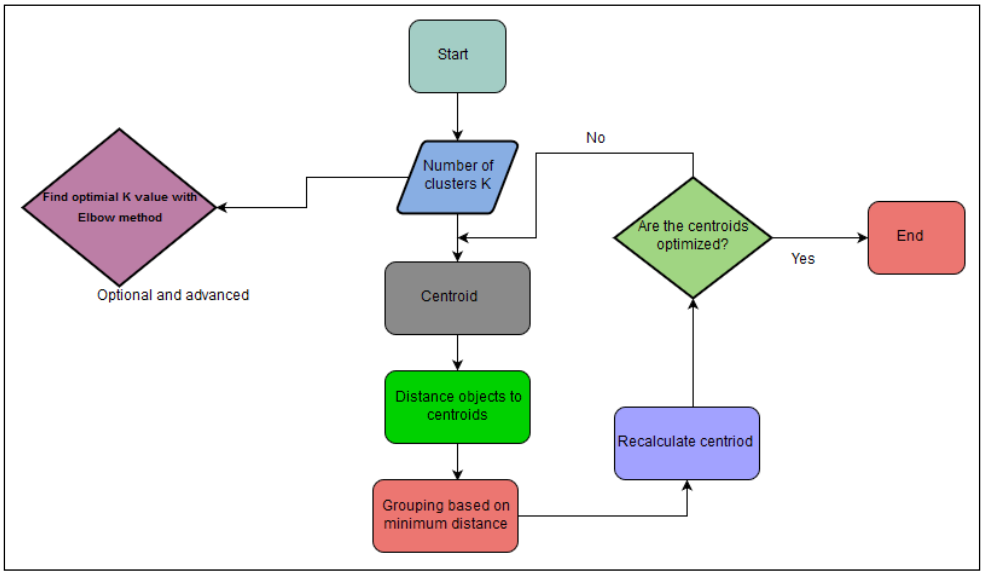


Figure 4: Flowchart of the K-means algorithm (Elbow method is an optional but also an advanced option)

Using K-means for predicting neighborhoods

Now, to show an example of clustering using K-means, we will use the Saratoga NY Homes dataset downloaded from `http://course1.winona.edu/bdeppa/Stat%20425/Datasets.html` as an unsupervised learning technique. The dataset contains several features of the houses located in the suburb of the New York City; for example, 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, only a few features have been shown in *Table 1*:

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 1: A sample data from the Saratoga NY Homes dataset

The target of this clustering technique is to show an exploratory analysis based on the features of each house in the city for finding possible neighborhoods' of the house located in the same area. Before performing the feature extraction, we need to load and parse the Saratoga NY Homes dataset. However, we will look at this example with step-by-step source codes for better understanding:

1. Loading required libraries and packages.

   We need some built-in Python libraries, such as `os`, `random`, `numpy`, and `pandas`, for data manipulation; `PCA` for dimensionality reduction; `matplotlib` for plotting; and of course, `tensorflow`:

```
import os
import random
from random import choice, shuffle
import pandas as pd
import numpy as np
import tensorflow as tf
```

```
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d, Axes3D
```

2. Loading, parsing, and preparing a training set.

   Here, the first line is used to ensure the reproducibility of the result. The second line basically reads the dataset from your location and converts it into the Pandas data frame:

```
random.seed(12345)
train = pd.read_csv(os.path.join('input', 'saratoga.csv'))
x_train = np.array(train.iloc[:, 1:], dtype='float32')
```

   If you now print the data frame (using `print(train)`), you should find the dataframe containing headers and data as shown in figure 3:

Price	LotSize	Waterfront	Age	LandValue	NewConstruct	CentralAir	FuelType	HeatType	SewerType	LivingArea	PctCollege	Bedrooms	Fireplaces	Bathrooms	rooms
132500.0	0.09	0.0	42.0	50000.0	0.0	0.0	3.0	4.0	2.0	906.0	35.0	2.0	1.0	1.0	5.0
181115.0	0.92	0.0	0.0	22300.0	0.0	0.0	2.0	3.0	2.0	1953.0	51.0	3.0	0.0	2.5	6.0
109000.0	0.19	0.0	133.0	7300.0	0.0	0.0	2.0	3.0	3.0	1944.0	51.0	4.0	1.0	1.0	8.0
155000.0	0.41	0.0	13.0	18700.0	0.0	0.0	2.0	2.0	2.0	1944.0	51.0	3.0	1.0	1.5	5.0
86060.0	0.11	0.0	0.0	15000.0	1.0	1.0	2.0	2.0	3.0	840.0	51.0	2.0	0.0	1.0	3.0
120000.0	0.68	0.0	31.0	14000.0	0.0	0.0	2.0	2.0	2.0	1152.0	22.0	4.0	1.0	1.0	8.0
153000.0	0.4	0.0	33.0	23300.0	0.0	0.0	4.0	3.0	2.0	2752.0	51.0	4.0	1.0	1.5	8.0
170000.0	1.21	0.0	23.0	14600.0	0.0	0.0	4.0	2.0	2.0	1662.0	35.0	4.0	1.0	1.5	9.0
90000.0	0.83	0.0	36.0	22200.0	0.0	0.0	3.0	4.0	2.0	1632.0	51.0	3.0	0.0	1.5	8.0
122900.0	1.94	0.0	4.0	21200.0	0.0	0.0	2.0	2.0	1.0	1416.0	44.0	3.0	0.0	1.5	6.0
325000.0	2.29	0.0	123.0	12600.0	0.0	0.0	4.0	2.0	2.0	2894.0	51.0	7.0	0.0	1.0	12.0
120000.0	0.92	0.0	1.0	22300.0	0.0	0.0	2.0	2.0	2.0	1624.0	51.0	3.0	0.0	2.0	6.0
85860.0	8.97	0.0	13.0	4800.0	0.0	0.0	3.0	4.0	2.0	704.0	41.0	2.0	0.0	1.0	4.0
97000.0	0.11	0.0	153.0	3100.0	0.0	0.0	2.0	3.0	3.0	1383.0	57.0	3.0	0.0	2.0	5.0
127000.0	0.14	0.0	9.0	300.0	0.0	0.0	4.0	2.0	2.0	1300.0	41.0	3.0	0.0	1.5	8.0
89900.0	0.0	0.0	88.0	2500.0	0.0	0.0	2.0	3.0	3.0	936.0	57.0	3.0	0.0	1.0	4.0
155000.0	0.13	0.0	9.0	300.0	0.0	0.0	4.0	2.0	2.0	1300.0	41.0	3.0	0.0	1.5	7.0
253750.0	2.0	0.0	0.0	49800.0	0.0	1.0	2.0	2.0	1.0	2816.0	71.0	4.0	1.0	2.5	12.0
60000.0	0.21	0.0	82.0	8500.0	0.0	0.0	4.0	3.0	2.0	924.0	35.0	2.0	0.0	1.0	6.0
87500.0	0.88	0.0	17.0	19400.0	0.0	0.0	4.0	2.0	2.0	1092.0	35.0	3.0	0.0	1.0	6.0

only showing top 20 rows

Figure 5: A snapshot of the Saratoga NY Homes dataset

Well, we have managed to prepare the dataset. Now, the next task is to conceptualize our K-means and write a function/class for it.

3. Implementing K-means.

   The following is the source code of K-means, which is simple in a TensorFlow way:

```
def kmeans(x, n_features, n_clusters, n_max_steps=1000, early_
stop=0.0):
    input_vec = tf.constant(x, dtype=tf.float32)
    centroids = tf.Variable(tf.slice(tf.random_shuffle(input_vec),
[0, 0], [n_clusters, -1]), dtype=tf.float32)
    old_centroids = tf.Variable(tf.zeros([n_clusters, n_
features]), dtype=tf.float32)
    centroid_distance = tf.Variable(tf.zeros([n_clusters, n_
features]))
```

```
    expanded_vectors = tf.expand_dims(input_vec, 0)
    exanded_centroids = tf.expand_dims(centroids, 1)
    distances = tf.reduce_sum(tf.square(tf.subtract(expanded_
vectors, expanded_centroids)), 2)
    assignments = tf.argmin(distances, 0)
    means = tf.concat([tf.reduce_mean(
        tf.gather(input_vec, tf.reshape(tf.where(tf.
equal(assignments, c)), [1, -1])),
        reduction_indices=[1]) for c in range(n_clusters)], 0)
    save_old_centroids = tf.assign(old_centroids, centroids)
    update_centroids = tf.assign(centroids, means)
    init_op = tf.global_variables_initializer()
    performance = tf.assign(centroid_distance,
tf.subtract(centroids, old_centroids))
    check_stop = tf.reduce_sum(tf.abs(performance))
    with tf.Session() as sess:
        sess.run(init_op)
        for step in range(n_max_steps):
            sess.run(save_old_centroids)
            _, centroid_values, assignment_values = sess.run(
                [update_centroids, centroids, assignments])
            sess.run(check_stop)
            current_stop_coeficient = check_stop.eval()
            if current_stop_coeficient <= early_stop:
                break
    return centroid_values, assignment_values
```

The previous code contains all the steps required to develop the K-means model, including the distance-based centroid calculation, centroid update, and training parameters required.

4. Clustering the houses.

   Now the previous function can be invoked with real values, for example, our housing dataset. Since there are many houses with their respective features, it would be difficult to plot the clusters along with all the properties. This is the **Principal Component Analysis** (**PCA**) that we discussed in the previous chapters:

```
centers, cluster_assignments = kmeans(x_train, len(x_train[0]),
10)
pca_model = PCA(n_components=3)
reduced_data = pca_model.fit_transform(x_train)
reduced_centers = pca_model.transform(centers)
```

Well, now we are all set. It would be even better to visualize the clusters as shown in figure 6. For this, we will use `mpl_toolkits.mplot3d` for 3D projection, as follows:

```
plt.subplot(212, projection='3d')
plt.scatter(reduced_data[:, 0], reduced_data[:, 1], reduced_
data[:, 2], c=cluster_assignments)
plt.title("Clusters")
plt.show()
>>>
```

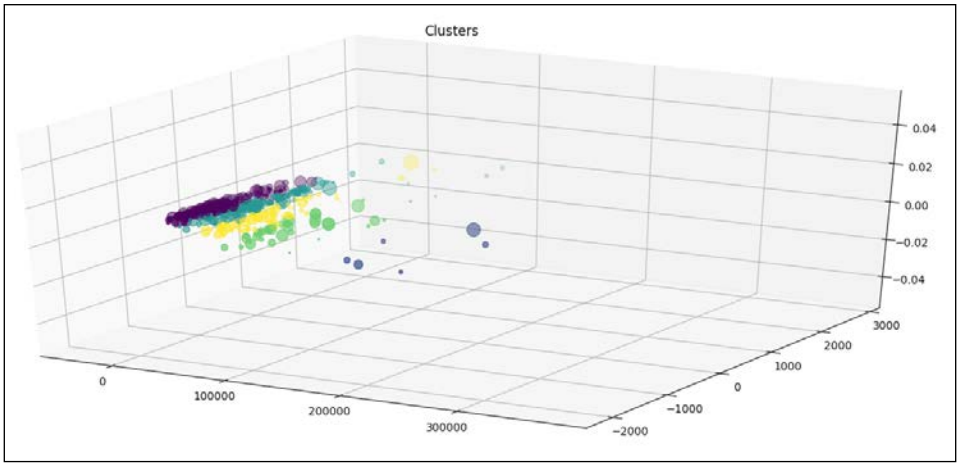


Figure 6: Clustering the houses with similar properties, for example, price

Here, we can see that most houses fall in **0** to **100,000** range. The second highest houses fall in the range of **100000** to **200000**. However, it's really difficult to separate them. Moreover, the number of predefined clusters that we used is 10, which might not be the most optimal one. Therefore, we need to tune this parameter.

5. Fine tuning and finding the optimal number of clusters.

   Choosing the right number of clusters often depends on the task. For example, suppose you're planning an event for hundreds of people, both young and old. If you have a budget for only two entertainment options, then you can use the K-means clustering with $k = 2$ to separate the guests into two age groups. Other times, it's not as obvious what the value of `k` should be. Automatically figuring out the value of `k` is a bit more complicated.

   As mentioned earlier, the K-means algorithm tries to minimize the sum of squares of the distance (that is, Euclidean distance), in terms of **Within-Cluster Sum of Squares** (**WCSS**).



However, if you want to minimize the sum of squares of the distance between the points of each set manually or automatically, you would end up with a model where each cluster is its own cluster center; in this case, this measure would be 0, but it would hardly be a generic enough model.

Therefore, once you have trained your model by specifying the parameters, you can evaluate the result using WCSS. Technically, it is same as the sum of distances of each observation in each K cluster. The beauty of clustering algorithms as a 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 raw data and would like to know the patterns in that data.

However, deciding the number of clusters prior to conducting the experiment might not be successful but sometimes may lead to an overfitting problem or an under-fitting one. Also, informally, determining the number of clusters is a separate but an optimization problem to be solved. So, based on this, we can redesign our K-means considering the WCSS value computation, as follows:

```
def kmeans(x, n_features, n_clusters, n_max_steps=1000, early_
stop=0.0):
    input_vec = tf.constant(x, dtype=tf.float32)
    centroids = tf.Variable(tf.slice(tf.random_shuffle(input_vec),
[0, 0], [n_clusters, -1]), dtype=tf.float32)
    old_centroids = tf.Variable(tf.zeros([n_clusters, n_
features]), dtype=tf.float32)
    centroid_distance = tf.Variable(tf.zeros([n_clusters, n_
features]))
    expanded_vectors = tf.expand_dims(input_vec, 0)
    expanded_centroids = tf.expand_dims(centroids, 1)
    distances = tf.reduce_sum(tf.square(tf.subtract(expanded_
vectors, expanded_centroids)), 2)
    assignments = tf.argmin(distances, 0)
    means = tf.concat([tf.reduce_mean(        tf.gather(input_vec,
tf.reshape(tf.where(tf.equal(assignments, c)), [1, -1])),
        reduction_indices=[1]) for c in range(n_clusters)], 0)
    save_old_centroids = tf.assign(old_centroids, centroids)
    update_centroids = tf.assign(centroids, means)
    init_op = tf.global_variables_initializer()

    performance = tf.assign(centroid_distance,
tf.subtract(centroids, old_centroids))
    check_stop = tf.reduce_sum(tf.abs(performance))
    calc_wss = tf.reduce_sum(tf.reduce_min(distances, 0))
```

```
    with tf.Session() as sess:
        sess.run(init_op)
        for step in range(n_max_steps):
            sess.run(save_old_centroids)
            _, centroid_values, assignment_values = sess.run(
                [update_centroids, centroids, assignments])
            sess.run(calc_wss)
            sess.run(check_stop)
            current_stop_coeficient = check_stop.eval()
            wss = calc_wss.eval()
            print(step, current_stop_coeficient)
            if current_stop_coeficient <= early_stop:
                break
    return centroid_values, assignment_values, wss
```

To fine tune the clustering performance, we can use a heuristic approach called Elbow method. We start from *K* = 2. Then, we run the K-means algorithm by increasing `K` and observe the value of the cost function (CF) using WCSS. At some point, we should experience a big drop with respect to CF. Nevertheless, the improvement then becomes marginal with an increasing value of `K`.

In summary, we can pick the `K` after the last big drop of WCSS as the optimal one. The K-means includes various parameters such as withiness and betweenness, analyzing which you can find out the performance of K-means:

- **Betweenness**: This is the between sum of squares, also called the intra-cluster similarity
- **Withiness**: This is the within sum of squares, also called the inter-cluster similarity
- **Totwithiness**: This is the sum of all the withiness of all the clusters, also called the total intra-cluster similarity

Note that a robust and accurate clustering model will have a lower value of withiness and a higher value of betweenness. However, these values depend on the number of clusters that is `K`, which is chosen before building the model. Now, based on this, we will train the K-means model for different `K` values that are a number of predefined clusters. We will start *K* = 2 to `10`, as follows:

```
wcss_list = []
for i in range(2, 10):
    centers, cluster_assignments, wcss = kmeans(x_train, len(x_
train[0]), i)
    wcss_list.append(wcss)
```

Now, let's discuss how we can take the advantage of the Elbow method for determining the number of clusters. We calculated the cost function WCSS as a function of a number of clusters for the K-means algorithm applied to home data based on all the features, as follows:

```
plt.figure(figsize=(12, 24))
plt.subplot(211)
plt.plot(range(2, 10), wcss_list)
plt.xlabel('No of Clusters')
plt.ylabel('WCSS')
plt.title("WCSS vs Clusters")
>>>
```

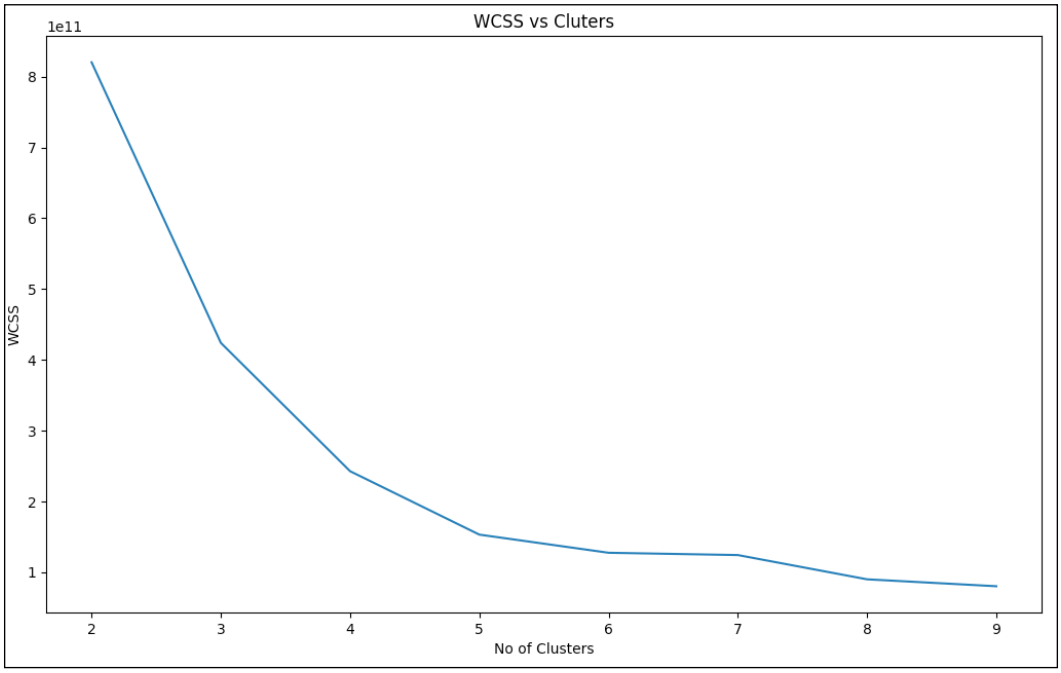


Figure 7: Number of clusters as a function of WCSS

We will try to reuse this lesson in upcoming examples using K-means too. Now, it can be observed that a big drop occurs when $k = 5$. Therefore, we chose the number of clusters to be **5** as discussed in figure 7. Basically, this is the one after the last big drop. This means that the optimal number of cluster for our dataset that we need to set before we start training the K-means model is **5**.

6. Clustering analysis.

   From figure 8, it is clear that most houses fall in **cluster 3** (**4655 houses**) and then in **cluster 4** (**3356 houses**). The x-axis shows the price and the y-axis shows the lot size for each house. We can also observe that the **cluster 1** has only a few houses and potentially in longer distances, but it is also expensive. So, it is most likely that you will not find a nearer neighborhood to interact with if you buy a house that falls in this cluster. However, if you like more human interaction and budget is a constraint, you should probably try buying a house from cluster 2, 3, 4, or 5:

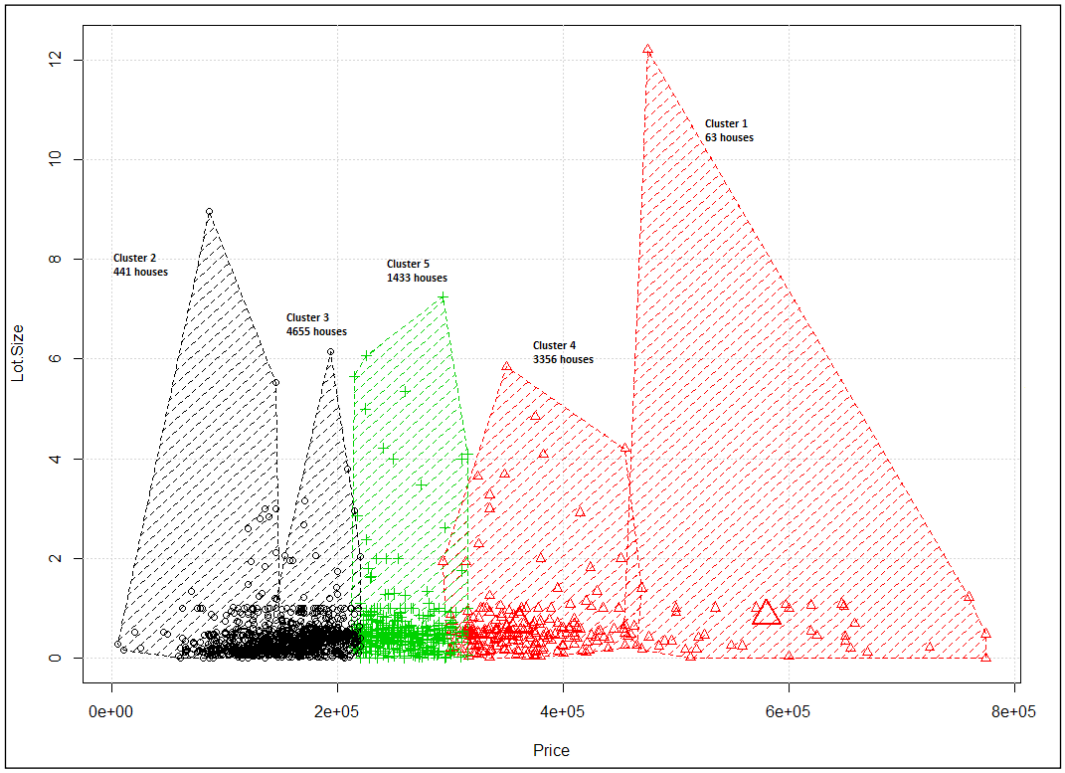


Figure 8: Clusters of neighborhoods, that is,. homogeneous houses fall in same clusters

To make the analysis, we dumped the output in RStudio and generated the clusters shown in figure 6. The R script can be found on my GitHub repositories at `https://github.com/rezacsedu/ScalaAndSparkForBigDataAnalytics`. Alternatively, you can write your own script and do the visualization accordingly.

Predictive models for clustering audio files

For clustering music with audio data, the data points are the feature vectors from the audio files. If two points are close together, it means that their audio features are similar. We want to discover which audio files belong to the same neighborhood because these clusters will probably be a good way to organize your music files:

1. Loading audio files with TensorFlow and Python.

   Some common input types in ML algorithms are audio and image files. This shouldn't come as a surprise because sound recordings and photographs are raw, redundant, and often noisy representations of semantic concepts. ML is a tool to help handle these complications. These data files have various implementations, for example, an audio file can be an MP3 or WAV.

   Reading files from a disk isn't exactly a ML-specific ability. You can use a variety of Python libraries to load files onto the memory, such as Numpy or Scipy. Some developers like to treat the data preprocessing step separately from the ML step. However, I believe that this is also a part of the whole analytics process.

   Since this is a TensorFlow book, I will try to use something from the TensorFlow built-in operator to list files in a directory called `tf.train.match_filenames_once()`. We can then pass this information along to a `tf.train.string_input_producer()queue` operator. This way, we can access a filename one at a time, without loading everything at once. Here's the structure of this method:

   ```
   match_filenames_once(
       pattern,
       name=None)
   ```

   This method takes two parameters: `pattern` and `name`. `pattern` signifies a file pattern or 1D tensor of file patterns. The `name` is used to signify the name of the operations. However, this parameter is optional. Once invoked, this method saves the list of matching patterns, so as the name implies, it is only computed once.

Finally, a variable that is initialized to the list of files matching the pattern(s) is returned by this method. Once we have finished reading the metadata and the audio files, we can decode the file to retrieve usable data from the given filename. Now, let's get started. First, we need to import necessary packages and Python modules, as follows:

```
import tensorflow as tf
import numpy as np
from bregman.suite import *
from tensorflow.python.framework import ops
import warnings
import random
```

Now we can start reading the audio files from the directory specified. First, we need to store filenames that match a pattern containing a particular file extension, for example, `.mp3`, `.wav`, and so on. Then, we need to set up a pipeline for retrieving filenames randomly. Now, the code natively reads a file in TensorFlow. Then, we run the reader to extract the file data. Use can use following code for this task:

```
filenames = tf.train.match_filenames_once('./audio_dataset/*.wav')
count_num_files = tf.size(filenames)
filename_queue = tf.train.string_input_producer(filenames)
reader = tf.WholeFileReader()
filename, file_contents = reader.read(filename_queue)
chromo = tf.placeholder(tf.float32)
max_freqs = tf.argmax(chromo, 0)
```

Well, once we have read the data and metadata about all the audio files, the next and immediate tasks are to capture the audio features that will be used by K-means for the clustering purpose.

2. Extracting features and preparing feature vectors.

   ML algorithms are typically designed to use feature vectors as input; however, sound files are a very different format. We need a way to extract features from sound files to create feature vectors.

   It helps to understand how these files are represented. If you've ever seen a vinyl record, you've probably noticed the representation of audio as grooves indented in the disk. Our ears interpret audio from a series of vibrations through the air. By recording the vibration properties, our algorithm can store sound in a data format. The real world is continuous but computers store data in discrete values.

The sound is digitalized into a discrete representation through an **Analog to Digital Converter** (**ADC**). You can think about sound as a fluctuation of a wave over time. However, this data is too noisy and difficult to comprehend. An equivalent way to represent a wave is by examining the frequencies that make it up at each time interval. This perspective is called the frequency domain.

It's easy to convert between time domain and frequency domain using a mathematical operation called a discrete Fourier transform (commonly known as the Fast Fourier transform). We will use this technique to extract a feature vector out of our sound.

A sound may produce 12 kinds of pitch. In music terminology, the 12 pitches are C, C#, D, D#, E, F, F#, G, G#, A, A#, and B. Figure 9 shows how to retrieve the contribution of each pitch in a 0.1-second interval, resulting in a matrix with 12 rows. The number of columns grows as the length of the audio file increases. Specifically, there will be *10*t* columns for a *t* second audio.

This matrix is also called a chromogram of the audio. But first, we need to have a placeholder for TensorFlow to hold the chromogram of the audio and the maximum frequency:

```
chromo = tf.placeholder(tf.float32)
max_freqs = tf.argmax(chromo, 0)
```

The next task that we can perform is that we can write a method that can extract these chromograms for the audio files. It can look as follows:

```
def get_next_chromogram(sess):
    audio_file = sess.run(filename)
    F = Chromagram(audio_file, nfft=16384, wfft=8192, nhop=2205)
    return F.X, audio_file
```

The workflow of the previous code is as follows:

- First, pass in the filename and use these parameters to describe 12 pitches every 0.1 seconds.
- Finally, represent the values of a 12-dimensional vector 10 times a second.

The chromogram output that we extract using the previous method will be a matrix, as visualized in figure 10. A sound clip can be read as a chromogram, and a chromogram is a recipe for generating a sound clip. Now, we have a way to convert audio and matrices. As you have learned, most ML algorithms accept feature vectors as a valid form of data. That being said, the first ML algorithm we'll look at is K-means clustering:

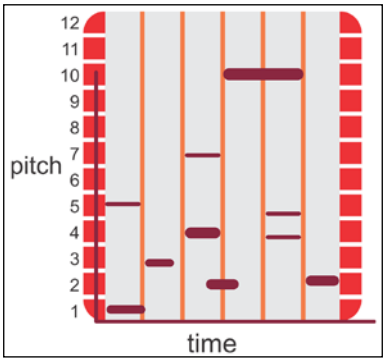


Figure 9: The visualization of the chromogram matrix where the x-axis represents time and the y-axis represents pitch class. The green markings indicate a presence of that pitch at that time

To run the ML algorithms on our chromogram, we first need to decide how we're going to represent a feature vector. One idea is to simplify the audio by only looking at the most significant pitch class per time interval, as shown in figure 10:

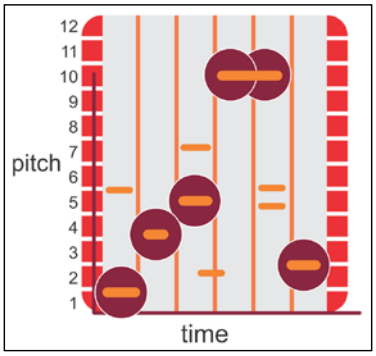


Figure 10: The most influential pitch at every time interval is highlighted. You can think of it as the loudest pitch at each time interval

Now, we will count the number of times each pitch shows up in the audio file. Figure 11 shows this data as a histogram, forming a 12-dimensional vector. If we normalize the vector so that all the counts add up to **1**, then we can easily compare audio of different lengths:

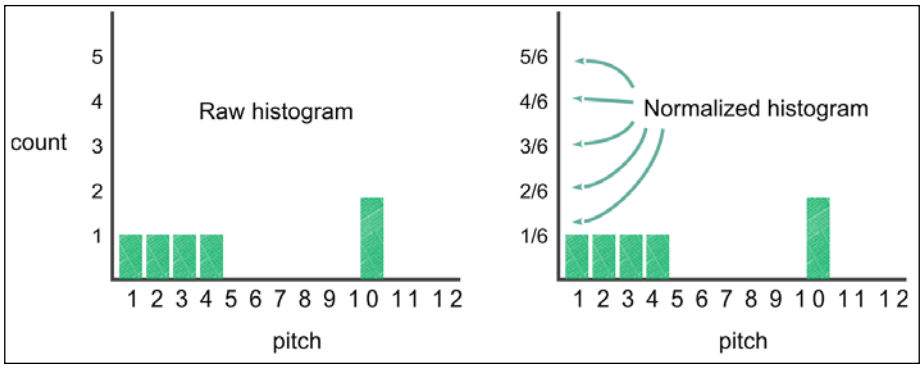


Figure 11: We count the frequency of the loudest pitches heard at each interval to generate this histogram, which acts as our feature vector

Now that we have the chromagram, we need to use it to extract the audio feature to construct a feature vector. You can use the following method for this:

```
def extract_feature_vector(sess, chromo_data):
    num_features, num_samples = np.shape(chromo_data)
    freq_vals = sess.run(max_freqs, feed_dict={chromo: chromo_
data})
    hist, bins = np.histogram(freq_vals, bins=range(num_features +
1))
    normalized_hist = hist.astype(float) / num_samples
    return normalized_hist
```

The workflow of the previous code is as follows:

- Create an operation to identify the pitch with the biggest contribution.
- Now, convert the chromogram into a feature vector.
- After this, we will construct a matrix where each row is a data item.
- Now, if you can hear the audio clip, you can imagine and differentiate between the different audio files. However, this is just intuition.

Therefore, we cannot rely on this, but we should inspect them visually. So, we will invoke the previous method to extract the feature vector from each audio file and plot the feature. The whole operation should look as follows:

```
def get_dataset(sess):
    num_files = sess.run(count_num_files)
    coord = tf.train.Coordinator()
    threads = tf.train.start_queue_runners(coord=coord)
    xs = list()
    names = list()
    plt.figure()
    for _ in range(num_files):
        chromo_data, filename = get_next_chromogram(sess)
        plt.subplot(1, 2, 1)
        plt.imshow(chromo_data, cmap='Greys',
interpolation='nearest')
        plt.title('Visualization of Sound Spectrum')
        plt.subplot(1, 2, 2)
        freq_vals = sess.run(max_freqs, feed_dict={chromo: chromo_
data})
        plt.hist(freq_vals)
        plt.title('Histogram of Notes')
        plt.xlabel('Musical Note')
        plt.ylabel('Count')
        plt.savefig('{}.png'.format(filename))
        plt.clf()
        plt.clf()
        names.append(filename)
        x = extract_feature_vector(sess, chromo_data)
        xs.append(x)
    xs = np.asmatrix(xs)
    return xs, names
```

The previous code should plot the audio features of each audio file in the histogram as follows:

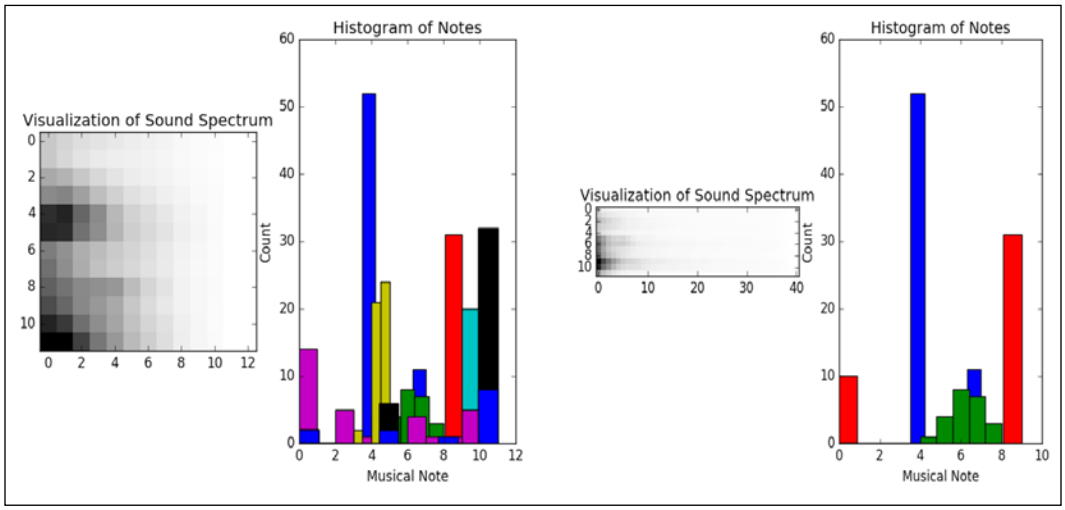


Figure 12: The ride audio files show a similar histogram

You can see some examples of audio files that we are trying to cluster based on their audio features. As you can see, the two on the right appear to have similar histograms. The two on the left also have similar sound spectrums:

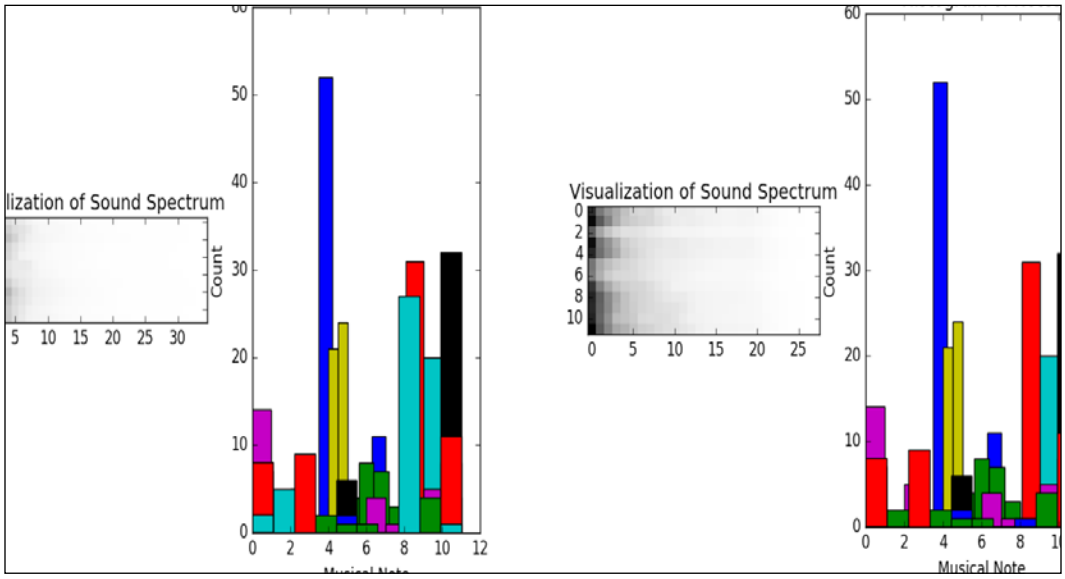


Figure 13: The crash cymbal audio files show a similar histogram

Now, the target is to develop K-means so that it is able to group these sounds together accurately. We will look at the high-level view of the cough audio files, as shown in the following figure:

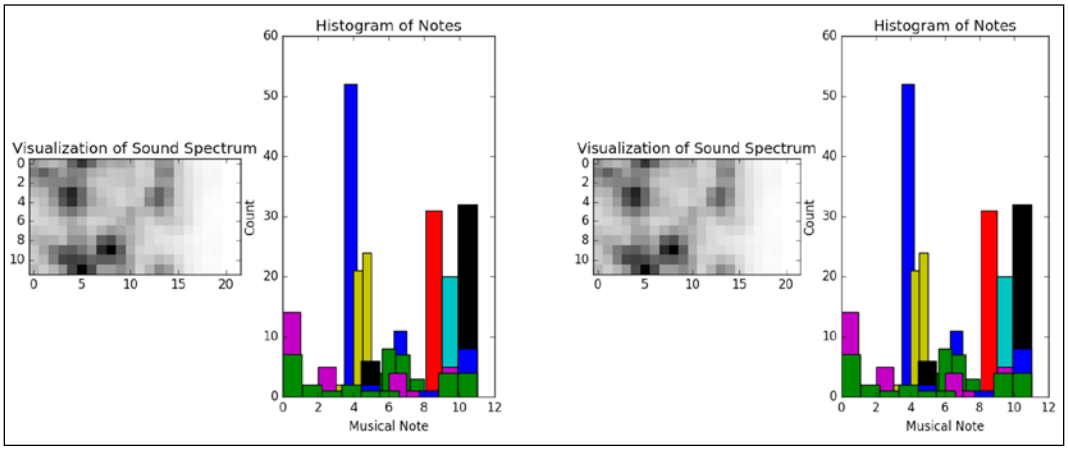


Figure 14: The cough audio files show a similar histogram

Finally, we have the scream audio files that have a similar histogram and audio spectrum, but of course are different compared to others:

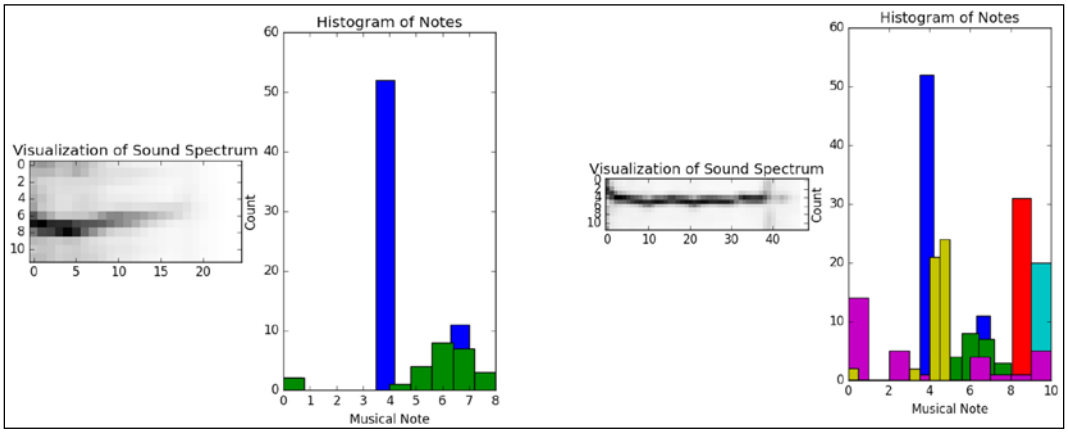


Figure 15: The scream audio files show a similar histogram and audio spectrum

Now, we can imagine our problem. We have the features ready for training the K-means model. Let's start doing it.

3. Training K-means model.

   Now that the feature vector is ready, it's time to feed this to the K-means model for clustering the feature presented in figure 10. The idea is that the midpoint of all the points in a cluster is called a centroid.

   Depending on the audio features we choose to extract, a centroid can capture concepts such as loud sound, high-pitched sound, or saxophone-like sound. Therefore, it's important to note that the K-means algorithm assigns non-descript labels, such as cluster 1, cluster 2, or cluster 3. First, we can write a method that computes the initial cluster centroids as follows:

```
def initial_cluster_centroids(X, k):
    return X[0:k, :]
```

   Now, the next task is to randomly assign the cluster number to each data point based on the initial cluster assignment. This time we can use another method:

```
def assign_cluster(X, centroids):
    expanded_vectors = tf.expand_dims(X, 0)
    expanded_centroids = tf.expand_dims(centroids, 1)
    distances = tf.reduce_sum(tf.square(tf.subtract(expanded_
vectors, expanded_centroids)), 2)
    calc_wss = tf.reduce_sum(tf.reduce_min(distances, 0))
    mins = tf.argmin(distances, 0)
    return mins, calc_wss
```

   The previous method computes the minimum distance and WCSS for the clustering evaluation in the later steps. Then, we need to update the centroid to check and make sure if there are any changes that occur in the cluster assignment:

```
def recompute_centroids(X, Y):
    sums = tf.unsorted_segment_sum(X, Y, k)
    counts = tf.unsorted_segment_sum(tf.ones_like(X), Y, k)
    return sums / counts
```

   Now that we have defined many variables, it's time to initialize them using `local_variable_initializer()`, as follows:

```
init_op = tf.local_variables_initializer()
```

Finally, we can perform the training. Forthis, the `audioClusterin()` method takes the number of tentative clusters k and iterates the training up to the maximum iteration, as follows:

```
def audioClustering(k, max_iterations ):
    with tf.Session() as sess:
        sess.run(init_op)
        X, names = get_dataset(sess)
        centroids = initial_cluster_centroids(X, k)
        i, converged = 0, False
        while not converged and i < max_iterations:
            i += 1
            Y, wcss_updated = assign_cluster(X, centroids)
            centroids = sess.run(recompute_centroids(X, Y))
        wcss = wcss_updated.eval()
        print(zip(sess.run(Y)), names)
    return wcss
```

The previous method returns the cluster cost, WCSS, and also prints the cluster number against each audio file. So, we have been able to finish the training step. Now, the next task is to evaluate the K-means clustering quality.

4. Evaluating the model.

   Here, we will evaluate the clustering quality from two perspectives. First, we will observe the predicted cluster number. Secondly, we will also try to find the optimal value of `k` as a function of WCSS. So, we will iterate the training for *K = 2* to say *10* and observe the clustering result. However, first, let's create two empty lists to hold the values of `K` and WCSS in each step:

   ```
   wcss_list = []
   k_list = []
   ```

   Now, let's iterate the training using the `for` loop as follows:

   ```
   for k in range(2, 9):
       random.seed(12345)
       wcss = audioClustering(k, 100)
       wcss_list.append(wcss)
       k_list.append(k)
   ```

This prints the following output:

```
([(0,), (1,), (1,), (0,), (1,), (0,), (0,), (0,), (0,), (0,), (0,)],
['./audio_dataset/scream_1.wav', './audio_dataset/Crash-Cymbal-3.
wav', './audio_dataset/Ride_Cymbal_1.wav', './audio_dataset/Ride_
Cymbal_2.wav', './audio_dataset/Crash-Cymbal-2.wav', './audio_
dataset/Ride_Cymbal_3.wav', './audio_dataset/scream_3.wav', './
audio_dataset/scream_2.wav', './audio_dataset/cough_2.wav', './audio_
dataset/cough_1.wav', './audio_dataset/Crash-Cymbal-1.wav'])


([(0,), (1,), (2,), (2,), (2,), (2,), (2,), (1,), (2,), (2,), (2,)],
['./audio_dataset/Ride_Cymbal_2.wav', './audio_dataset/Crash-
Cymbal-3.wav', './audio_dataset/cough_1.wav', './audio_dataset/Crash-
Cymbal-2.wav', './audio_dataset/scream_2.wav', './audio_dataset/
Ride_Cymbal_3.wav', './audio_dataset/Crash-Cymbal-1.wav', './audio_
dataset/Ride_Cymbal_1.wav', './audio_dataset/cough_2.wav', './audio_
dataset/scream_1.wav', './audio_dataset/scream_3.wav'])


([(0,), (1,), (2,), (3,), (2,), (2,), (2,), (2,), (2,), (2,), (2,)],
['./audio_dataset/Ride_Cymbal_2.wav', './audio_dataset/Ride_Cymbal_3.
wav', './audio_dataset/cough_1.wav', './audio_dataset/Crash-Cymbal-1.
wav', './audio_dataset/scream_3.wav', './audio_dataset/cough_2.wav',
'./audio_dataset/Crash-Cymbal-2.wav', './audio_dataset/Ride_Cymbal_1.
wav', './audio_dataset/Crash-Cymbal-3.wav', './audio_dataset/
scream_1.wav', './audio_dataset/scream_2.wav'])


([(0,), (1,), (2,), (3,), (4,), (0,), (0,), (4,), (0,), (0,), (0,)],
['./audio_dataset/cough_1.wav', './audio_dataset/scream_1.wav', './
audio_dataset/Crash-Cymbal-1.wav', './audio_dataset/Ride_Cymbal_2.
wav', './audio_dataset/Crash-Cymbal-3.wav', './audio_dataset/
scream_2.wav', './audio_dataset/cough_2.wav', './audio_dataset/
Ride_Cymbal_1.wav', './audio_dataset/Crash-Cymbal-2.wav', './audio_
dataset/Ride_Cymbal_3.wav', './audio_dataset/scream_3.wav'])


([(0,), (1,), (2,), (3,), (4,), (5,), (2,), (2,), (2,), (4,), (2,)],
['./audio_dataset/scream_3.wav', './audio_dataset/Ride_Cymbal_2.wav',
'./audio_dataset/cough_1.wav', './audio_dataset/Crash-Cymbal-2.wav',
'./audio_dataset/Crash-Cymbal-3.wav', './audio_dataset/scream_2.wav',
'./audio_dataset/Crash-Cymbal-1.wav', './audio_dataset/cough_2.wav',
'./audio_dataset/Ride_Cymbal_3.wav', './audio_dataset/Ride_Cymbal_1.
wav', './audio_dataset/scream_1.wav'])
```

```
([(0,), (1,), (2,), (3,), (4,), (5,), (6,), (5,), (6,), (5,), (5,)],
['./audio_dataset/cough_2.wav', './audio_dataset/Ride_Cymbal_3.wav',
'./audio_dataset/scream_1.wav', './audio_dataset/Ride_Cymbal_2.wav',
'./audio_dataset/Crash-Cymbal-1.wav', './audio_dataset/cough_1.wav',
'./audio_dataset/scream_2.wav', './audio_dataset/Crash-Cymbal-3.wav',
'./audio_dataset/scream_3.wav', './audio_dataset/Ride_Cymbal_1.wav',
'./audio_dataset/Crash-Cymbal-2.wav'])


([(0,), (1,), (2,), (3,), (4,), (5,), (6,), (7,), (6,), (6,), (1,)],
['./audio_dataset/Crash-Cymbal-1.wav', './audio_dataset/scream_3.
wav', './audio_dataset/Ride_Cymbal_3.wav', './audio_dataset/
Crash-Cymbal-3.wav', './audio_dataset/Crash-Cymbal-2.wav', './
audio_dataset/cough_2.wav', './audio_dataset/cough_1.wav', './audio_
dataset/Ride_Cymbal_1.wav', './audio_dataset/Ride_Cymbal_2.wav', './
audio_dataset/scream_1.wav', './audio_dataset/scream_2.wav'])


([(0,), (1,), (2,), (3,), (4,), (5,), (6,), (7,), (8,), (1,), (7,)],
['./audio_dataset/scream_2.wav', './audio_dataset/Ride_Cymbal_1.wav',
'./audio_dataset/Crash-Cymbal-2.wav', './audio_dataset/Ride_Cymbal_3.
wav', './audio_dataset/Ride_Cymbal_2.wav', './audio_dataset/scream_3.
wav', './audio_dataset/Crash-Cymbal-1.wav', './audio_dataset/cough_1.
wav', './audio_dataset/cough_2.wav', './audio_dataset/Crash-Cymbal-3.
wav', './audio_dataset/scream_1.wav'])
```

These values signify that each audio file is clustered and the cluster number has been assigned (the first bracket is the cluster number, the contents in the second bracket is the filename). However, it is difficult to judge the accuracy from this output. One naïve approach would be to compare each file with figure 12 to figure 15. Alternatively, let's adopt a better approach that we used in the first example that is the elbow method. For this, I have created a dictionary using two lists that are `k_list` and `wcss_list` computed previously, as follows:

```
dict_list = zip(k_list, wcss_list)
my_dict = dict(dict_list)
print(my_dict)
```

The previous code produces the following output:

```
{2: 2.8408628007260428, 3: 2.3755930780867365, 4: 0.9031724736903582,
5: 0.7849431270192495, 6: 0.872767581979385, 7: 0.62019339653673422,
8: 0.70075249251166494, 9: 0.86645706880532057}
```

From the previous output, you can see a sharp drop in WCSS for $k = 4$, and this is generated in the third iteration. So, based on this minimum evaluation, we can take a decision about the following clustering assignment:

```
Ride_Cymbal_1.wav => 2
Ride_Cymbal_2.wav => 0
cough_1.wav => 2
cough_2.wav =>2
Crash-Cymbal-1.wav =>3
Crash-Cymbal-2.wav => 2
scream_1.wav => 2
scream_2.wav => 2
```

Now that we have seen two complete examples of using K-means, there is another example called kNN. This is typically a supervised ML algorithm. In the next section, we will see how we can train this algorithm in an unsupervised way for a regression task.

Using kNN for predictive analytics

kNN is non-parametric and instance-based and is used in supervised learning. It is a robust and versatile classifier, frequently used as a benchmark for complex classifiers such as **Neural Networks** (**NNs**) and **Support Vector Machines** (**SVMs**). kNN is commonly used in economic forecasting, data compression, and genetics based on their expression profiling.

Working principles of kNN

The idea of kNN is that from a set of features x we try to predict the labels y. Thus, kNN falls in a supervised learning family of algorithms. Informally, this means that we are given a labeled dataset consisting of training observations (x, y). Now, the task is to model the relationship between x and y so that the function $f: X \rightarrow Y$ learns from the unseen observation x. The function $f(x)$ can confidently predict the corresponding label y prediction on a point z by looking at a set of nearest neighbors.

However, the actual method of prediction depends on whether or not we are doing regression (continuous) or classification (discrete). For discrete classification targets, the prediction may be given by a maximum voting scheme weighted by the distance to the prediction point:

$$f(z) = \max_{j} \sum_{i=1}^{k} \varphi(d_{ij}) l_{ij}$$

Here, our prediction, *f(z)*, is the maximum weighted value of all classes, *j*, where the weighted distance from the prediction point to the training point, *i*, is given by $\varphi(d_{ij})$, where *d* indicates the distance between two points. On the other hand, I_{ij} is just an indicator function if point *i* is in class *j*.

For continuous regression targets, the prediction is given by a weighted average of all *k* points nearest to the prediction:

$$f(z) = \frac{1}{k}\sum_{i=1}^{k}\varphi(d_i)$$

From the previous two equations, it is clear that the prediction is heavily dependent on the choice of the distance metric, *d*. There are many different specifications of distance metrics such as L1 and L2 metrics can be used for the textual distances:

A straightforward way to weigh the distances is by the distance itself. Points that are further away from our prediction should have less impact than the nearer points. The most common way to weigh is the normalized inverse of the distance. We will implement this method in the next section.

Implementing a kNN-based predictive model

To illustrate how making predictions with the nearest neighbors works in TensorFlow, we will use the 1970s Boston housing dataset, which is available through the UCI ML repository at `https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.data`. The following table shows the basic description of the dataset:

Header	**Significance**
CRIM	Per capita crime rate by town
N	Proportion of the residential land zones
INDUS	Proportion of non-retail business acres
CHAS	Charles river dummy variable
NOX	Nitric oxide concentration/10 M
RM	Average # of rooms per building
AGE	Proportion of buildings built prior to 1940
DIS	Weighted distances to employment centers
RAD	Index of radian highway access
TAX	Full tax rate value per $10k

Header	Significance
PTRATIO	Pupil/teacher ratio by town
B	1,000*(Bk-0.63)^2, Bk=prop. of blocks
LSTAT	% lower status of pop
MEDV	Median value of homes in $1,000s

Here, we will predict the median neighborhood housing value that is the last value named MEDV as a function of several features. Since we consider the training set the trained model, we will find kNNs to the prediction points and do a weighted average of the target value. Let's get started:

1. Loading required libraries and packages.

   As an entry point, we import necessary libraries and packages that will be needed to do predictive analytics using kNN with TensorFlow:

   ```
   import matplotlib.pyplot as plt
   import numpy as np
   import random
   import os
   import tensorflow as tf
   import requests
   from tensorflow.python.framework import ops
   import warnings
   ```

2. Resetting the default graph and disabling the TensorFlow warning.

   We need to reset the default TensorFlow graph using the `reset_default_graph()` function from TensorFlow. You must also disable all warnings due to the absence of GPU on your device:

   ```
   warnings.filterwarnings("ignore")
   os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
   ops.reset_default_graph()
   ```

3. Loading and preprocessing the dataset.

   First, we will load and parse the dataset using the `get()` function from the requests package as follows:

   ```
   housing_url = 'https://archive.ics.uci.edu/ml/machine-learning-
   databases/housing/housing.data'
   housing_header = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM',
   'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV']
   num_features = len(housing_header)
   housing_file = requests.get(housing_url)
   housing_data = [[float(x) for x in y.split(' ') if len(x)>=1] for
   y in housing_file.text.split('\n') if len(y)>=1]
   ```



For more information on how the previous code works, please see the documentation of requests package at `http://docs.python-requests.org/en/master/user/quickstart/`.

Then, we will separate features (predictor) from the labels:

```
y_vals = np.transpose([np.array([y[len(housing_header)-1] for y in
housing_data])])
x_vals = np.array([[x for i,x in enumerate(y) if housing_header[i]
in housing_header] for y in housing_data])
```

Now, to get some idea of the features and labels, let's print them as follows:

```
print(y_vals)
>>>
[[ 24. ]
 [ 21.6]
 [ 34.7]
 [ 33.4]
 [ 36.2]
 [ 28.7]
 [ 22.9]
 [ 27.1]
 [ 16.5]
 [ 18.9]
 [ 15. ]
…]
```

So, the labels are okay to work with, and these are also continuous values. Now, let's see the features:

```
print(x_vals)
>>>
[[  6.32000000e-03   1.80000000e+01   2.31000000e+00 ...,
3.96900000e+02
    4.98000000e+00   2.40000000e+01]
 [  2.73100000e-02   0.00000000e+00   7.07000000e+00 ...,
3.96900000e+02
    9.14000000e+00   2.16000000e+01]
 [  2.72900000e-02   0.00000000e+00   7.07000000e+00 ...,
3.92830000e+02
    4.03000000e+00   3.47000000e+01]
 ...,
 [  6.07600000e-02   0.00000000e+00   1.19300000e+01 ...,
3.96900000e+02
    5.64000000e+00   2.39000000e+01]
 [  1.09590000e-01   0.00000000e+00   1.19300000e+01 ...,
3.93450000e+02
    6.48000000e+00   2.20000000e+01]
```

```
 [  4.74100000e-02   0.00000000e+00   1.19300000e+01 ...,
3.96900000e+02
    7.88000000e+00   1.19000000e+01]]
```

Well, if you see these values, they are pretty unscaled to be fed to a predictive model. Thus, we need to apply the min-max scaling to get a better structure of the features so that an estimator scales and translates each feature individually, and it ensures that it is in the given range on the training set, that is, between zero and one. Since features are most important in predictive analytics, we should take special care of them. The following line of code does the min-max scaling:

```
x_vals = (x_vals - x_vals.min(0)) / x_vals.ptp(0)
```

Now let's print them again to check to make sure what's changed:

```
print(x_vals)
>>>
[[  0.00000000e+00   1.80000000e-01   6.78152493e-02 ...,
1.00000000e+00
    8.96799117e-02   4.22222222e-01]
 [  2.35922539e-04   0.00000000e+00   2.42302053e-01 ...,
1.00000000e+00
    2.04470199e-01   3.68888889e-01]
 [  2.35697744e-04   0.00000000e+00   2.42302053e-01 ...,
9.89737254e-01
    6.34657837e-02   6.60000000e-01]
 ...,
 [  6.11892474e-04   0.00000000e+00   4.20454545e-01 ...,
1.00000000e+00
    1.07891832e-01   4.20000000e-01]
 [  1.16072990e-03   0.00000000e+00   4.20454545e-01 ...,
9.91300620e-01
    1.31070640e-01   3.77777778e-01]
 [  4.61841693e-04   0.00000000e+00   4.20454545e-01 ...,
1.00000000e+00
    1.69701987e-01   1.53333333e-01]]
```

4. Preparing the training and test set.

   Since our features are already scaled, now it's time to split the data into train and test sets. Now, we split the x and y values into the train and test sets. We will create the training set by selecting about 75% of the rows at random and leave the remaining 25% for the test set:

```
train_indices = np.random.choice(len(x_vals), int(len(x_
vals)*0.75), replace=False)
test_indices = np.array(list(set(range(len(x_vals))) - set(train_
indices)))
x_vals_train = x_vals[train_indices]
```

```
x_vals_test = x_vals[test_indices]
y_vals_train = y_vals[train_indices]
y_vals_test = y_vals[test_indices]
```

5. Preparing the placeholders for the tensors.

   First, we will declare the batch size. Ideally, the batch size should be equal to the size of features in the test set:

   ```
   batch_size=len(x_vals_test)
   ```

   Then, we need to declare the placeholders for the TensorFlow tensors, as follows:

   ```
   x_data_train = tf.placeholder(shape=[None, num_features],
   dtype=tf.float32)
   x_data_test = tf.placeholder(shape=[None, num_features], dtype=tf.
   float32)
   y_target_train = tf.placeholder(shape=[None, 1], dtype=tf.float32)
   y_target_test = tf.placeholder(shape=[None, 1], dtype=tf.float32)
   ```

6. Defining the distance metrics.

   For this example, we are going to use the L1 distance. The reason is that using L2 did not give a better result in my case:

   ```
   distance = tf.reduce_sum(tf.abs(tf.subtract(x_data_train,
   tf.expand_dims(x_data_test,1))), axis=2)
   ```

7. Implementing kNN.

   Now, it's time to implement kNN. This will predict the nearest neighbors by getting the minimum distance index. The `kNN()` method does the trick. There are several steps for doing this, as follows:

   1. Get the minimum distance index.
   2. Compute the prediction function. To do this, we will use the `top_k()`, function, which returns the values and indices of the largest values in a tensor. Since we want the indices of the smallest distances, we will instead find the k-biggest negative distances. Since we are predicting continuous values that is regression task, we also declare the predictions and **Mean Squared Error** (**MSE**) of the target values.
   3. Calculate the number of loops over training data.
   4. Initialize the global variables.
   5. Iterate the training over the number of loops calculated in step 3.

Now, here's the function of kNN. It takes the number of initial neighbors and starts the computation. Note that although it is a widely used convention, here I will make it a variable to do some tuning, as shown in the following code:

```
def kNN(k):
    topK_X, topK_indices = tf.nn.top_k(tf.negative(distance), k=k)
    x_sums = tf.expand_dims(tf.reduce_sum(topK_X, 1), 1)
    x_sums_repeated = tf.matmul(x_sums,tf.ones([1, k],
tf.float32))
    x_val_weights = tf.expand_dims(tf.div(topK_X, x_sums_
repeated), 1)
    topK_Y = tf.gather(y_target_train, topK_indices)
    prediction = tf.squeeze(tf.matmul(x_val_weights,topK_Y),
axis=[1])
   mse = tf.div(tf.reduce_sum(tf.square(tf.subtract(prediction, y_
target_test))), batch_size)
    num_loops = int(np.ceil(len(x_vals_test)/batch_size))
    init_op = tf.global_variables_initializer()
    with tf.Session() as sess:
            sess.run(init_op)
            for i in range(num_loops):
                min_index = i*batch_size
                max_index = min((i+1)*batch_size,len(x_vals_
train))
                x_batch = x_vals_test[min_index:max_index]
                y_batch = y_vals_test[min_index:max_index]
                predictions = sess.run(prediction, feed_dict={x_
data_train: x_vals_train, x_data_test: x_batch, y_target_train:
y_vals_train, y_target_test: y_batch})
                batch_mse = sess.run(mse, feed_dict={x_data_train:
x_vals_train, x_data_test: x_batch, y_target_train: y_vals_train,
y_target_test: y_batch})
    return batch_mse
```

8. Evaluating the classification/regression.

   Note that this function does not return the optimal `mse` value, that is, the lowest `mse` value, but varies over different `k` values, so this is a hyperparameter to be tuned. One potential technique would be to iterate the method for $k = 2$ to, say, `11` and keeping track of the optimal `k` value that forces `kNN()` to produce the lowest `mse` value. First, we define a method that iterates several times from `2` to `11` and returns two separate lists for `mse` and `k` respectively:

```
mse_list = []
k_list = []
def getOptimalMSE_K():
```

```
    mse = 0.0
    for k in range(2, 11):
        mse = kNN(k)
        mse_list.append(mse)
        k_list.append(k)
    return k_list, mse_list
```

Now, it's time to invoke the previous method and find the optimal `k` value for which the kNN produces the lowest `mse` value. Upon receiving the two lists, we create a dictionary and use the `min()` method to return the optimal `k` value, as follows:

```
k_list, mse_list  = getOptimalMSE_K()
dict_list = zip(k_list, mse_list)
my_dict = dict(dict_list)
print(my_dict)
optimal_k = min(my_dict, key=my_dict.get)
>>>
{2: 7.6624126, 3: 10.184645, 4: 8.9112329, 5: 11.29573, 6:
13.341181, 7: 14.406253, 8: 13.923589, 9: 14.915736, 10:
13.920851}
```

Now, let's print `the Optimal k value` for which we get the lowest `mse` value:

```
print("Optimal K value: ", optimal_k)
mse = min(mse_list)
print("Minimum mean square error: ", mse)
>>>
Optimal K value: 2 minimum mean square error: 7.66241
```

9. Running the best kNN.

   Now we have the optimal `k`, so we will entertain calculating the nearest neighbor. This time we will try to return the matrices for the predicted and actual labels:

```
def bestKNN(k):
    topK_X, topK_indices = tf.nn.top_k(tf.negative(distance), k=k)
    x_sums = tf.expand_dims(tf.reduce_sum(topK_X, 1), 1)
    x_sums_repeated = tf.matmul(x_sums,tf.ones([1, k],
tf.float32))
    x_val_weights = tf.expand_dims(tf.div(topK_X, x_sums_
repeated), 1)
    topK_Y = tf.gather(y_target_train, topK_indices)
    prediction = tf.squeeze(tf.matmul(x_val_weights,topK_Y),
axis=[1])
    num_loops = int(np.ceil(len(x_vals_test)/batch_size))
```

```
    init_op = tf.global_variables_initializer()
    with tf.Session() as sess:
            sess.run(init_op)
            for i in range(num_loops):
                min_index = i*batch_size
                max_index = min((i+1)*batch_size,len(x_vals_
train))
                x_batch = x_vals_test[min_index:max_index]
                y_batch = y_vals_test[min_index:max_index]
```

predictions = sess.run(prediction, feed_dict={x_data_train: x_vals_train, x_data_test: x_batch, y_target_train: y_vals_train, y_target_test: y_batch})

```
    return predictions, y_batch
```

10. Evaluating the best kNN.

    Now, we will invoke the `bestKNN()` method with the optimal value of `k` that was calculated in the previous step, as follows:

```
predicted_labels, actual_labels = bestKNN(optimal_k)
```

    Now, I would like to measure the prediction accuracy. Are you wondering why? I know the reason. You're right. There is no significant reason for calculating the accuracy or precision since we are predicting the continuous values that is labels. Even so, I would like to show you whether it works or not:

```
def getAccuracy(testSet, predictions):
 correct = 0
 for x in range(len(testSet)):
     if(np.round(testSet[x]) == np.round(predictions[x])):
                 correct += 1
 return (correct/float(len(testSet))) * 100.0
accuracy = getAccuracy(actual_labels, predicted_labels)
print('Accuracy: ' + repr(accuracy) + '%')
>>>
Accuracy: 17.322834645669293%
```

    The previous `getAccuracy()` method computes the accuracy, which is quite low. This is obvious and there is no exertion. This also implies that the previous method is pointless. However, if you are about to predict discrete values, this method will obviously help you. Try it yourself with suitable data and combinations of the previous code.

But do not to be disappointed; we have another way of looking at how our predictive model performs. We can still plot a histogram showing the predicted versus actual labels that are a prediction and actual distribution:

```
bins = np.linspace(5, 50, 45)
plt.hist(predicted_labels, bins, alpha=1.0, facecolor='red',
label='Prediction')
plt.hist(actual_labels, bins, alpha=1.0, facecolor='green',
label='Actual')
plt.title('predicted vs actual values')
plt.xlabel('Median house price in $1,000s')
plt.ylabel('count')
plt.legend(loc='upper right')
plt.show()
>>>
```

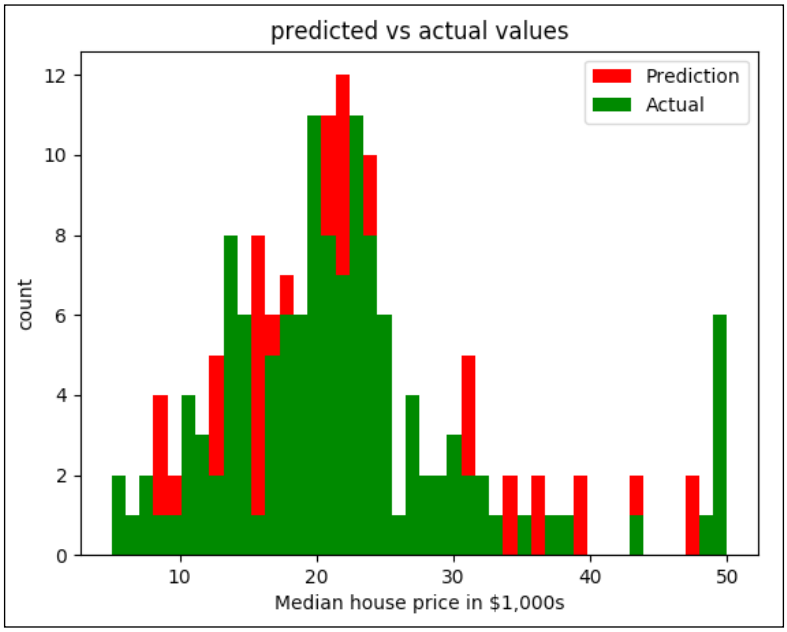


Figure 16: Predicted versus actual median prices of the houses in $1,000s

Summary

In this chapter, we have discussed unsupervised learning from a theoretical and practical perspective. We have seen how we can make use of predictive analytics and find out how we can take advantage of it to cluster records belonging to a certain group or class for a dataset of unsupervised observations. We have discussed unsupervised learning and clustering using K-means. In addition, we have seen how we can fine tune the clustering using the Elbow method for better predictive accuracy. We have also seen how to predict neighborhoods using K-means, and then, we have seen another example of clustering audio clips based on their audio features. Finally, we have seen how we can use unsupervised kNN for predicting the nearest neighbors.

In the next chapter, we will discuss the wonderful field of text analytics using TensorFlow. Text analytics is a wide area in **natural language processing** (**NLP**), and ML is useful in many use cases, such as sentiment analysis, chatbots, email spam detection, text mining, and natural language processing. You will learn how to use TensorFlow for text analytics with a focus on use cases of text classification from the unstructured spam prediction and movie review dataset. Based on the spam filtering dataset, we will develop predictive models using a LR algorithm with TensorFlow.

6
Predictive Analytics Pipelines for NLP

In this chapter, we will discuss the wonderful field of text analytics using TensorFlow for **natural language processing** (**NLP**), and it is useful in many use cases such as sentiment analysis, email spam detection, text mining, NLP, and much more. We will learn how to use TensorFlow with a focus on use cases of text classification from unstructured spam prediction and movie review datasets. Particularly, we will use **bag-of-words** (**BOW**) and TF-IDF algorithms for spam prediction that are more than 90% accurate.

Later in this chapter, we will see how to develop large-scale predictive models for predicting sentiment from the movie review dataset using **continuous bag-of-words** (**CBOW**) and continuous skip-gram algorithms, jointly referred to as a **Word2vec** method; this model has more than 50% prediction accuracy.

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

- NLP analytics pipelines
- Transformers and estimators
- Working with bag-of-words for spam prediction
- Implementing TF-IDF for predictive analytics
- Making predictions using Word2vec

NLP analytics pipelines

So far, we have explored the world of predictive analytics using numerical dataset most of the cases. However, what is to be done in the case of unstructured text datasets? This section provides some answers to this question. As we discussed, like general purpose machine learning, predictive analytics has a workflow as follows:

- Loading or ingesting data
- Cleansing the data
- Extracting features from the data
- Training a model on the data to generate desired outcomes based on features
- Evaluating or predicting some outcomes based on the data.

A simplified view of a typical pipeline containing the preceding steps is shown in the following figure:

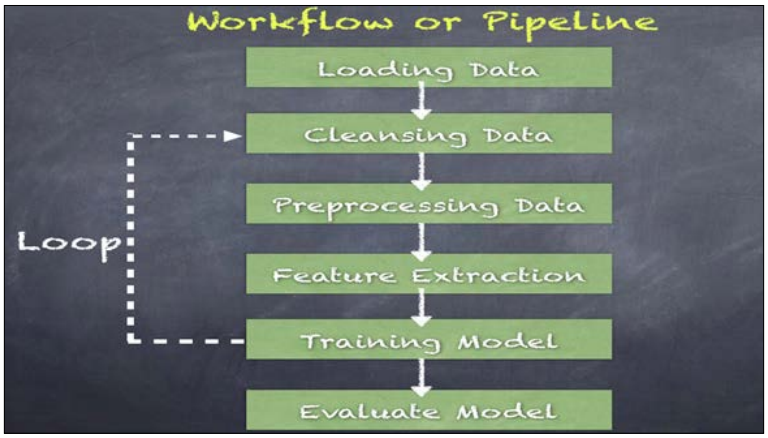


Figure 1: A simplified view of a typical ML pipeline (Karim et al., Scala and Spark for Big Data Analytics, Packt Publishing Ltd., 2017.)

However, if we want to use text, particularly unstructured text, we must find a robust and efficient way of feature engineering to convert the text into numbers. There are many ways to do this, and we will explore a few common ways this is achieved.

Hence, there are several stages of transformation of data possible before a model is trained and then subsequently deployed, finally doing the predictive analytics. Moreover, we should expect the refinement of the features and model attributes. We could even explore a completely different algorithm repeating the entire sequence of tasks as part of a new workflow.

When you look at a line of text, you see sentences, phrases, words, nouns, verbs, punctuation, and so on, which when put together have a meaning and purpose. Humans are very good at understanding sentences, words, slang, annotations, or context extremely well. This comes from years of practice and learning how to read/write proper grammar, punctuation, exclamations, and so on. So how can we write a TensorFlow program to try to replicate this kind of capability?

In the previous two chapters, we have looked at some procedures that convert any sentence to a fixed length numerical vector. Using this method has two disadvantages.

NLP models often fail to interpret the semantics in a complex sentence especially for long sentences or even words.

For example, the two sentences "TensorFlow makes predictive analytics easy" and "predictive analytics makes TensorFlow easy" might result in the same sentence vector having the same length equal to the size of the vocabulary that we pick.

The second drawback is that the words "is" and "TensorFlow" have the same numerical index value of one. We can imagine that the word "is" might be less important than the occurrence of the word "TensorFlow". Therefore, we should remove these types of words from the regular text or sentences.

We will explore different types of embedding in this chapter, that attempt to address these ideas. However, before getting into this deeper, let's see some potential use cases of text analytics.

Using text analytics

Text analytics is the way to unlock the meaning from all of this unstructured text by providing means to uncover patterns and themes and derive contextual meaning and relationships. Text analytics utilizes several broad categories of techniques as follows:

- **Sentiment analysis**: Analyzing the political opinions of people on Facebook, Twitter, and other social media is a good example of sentiment analysis.
- **Topic modelling**: Topic modelling is a useful technique to detect the topics or themes in a corpus of documents. **Latent Drichilet Allocation** (**LDA**) is a popular clustering model using unsupervised algorithms.
- **Term frequency - inverse document frequency** (**TF-IDF**): TF-IDF measures how frequently words appear in documents and the relative frequency across the set of documents. This information can be used in building classifiers and predictive models.

- **Named entity recognition**: Named entity recognition detects the usage of words and nouns in sentences to extract information about persons, organizations, locations, and so on.
- **Event extraction**: Event extraction expands on **named-entity recognition** (**NER**), establishing relationships between the entities detected. This can be used to make inferences on the relationship between two entities.

So far we have seen some potential use cases where predictive analytics can be used to handle unstructured text. Now it's time to go deeper before we start with an implementation of a bag-of-words that will be used for spam prediction. However, to deal with the unstructured text, we do need two arsenals called transformer and estimator.

Transformers and estimators

A transformer is a function object that transforms one dataset to another by applying the transformation logic (function) to the input dataset yielding an output dataset. There are two types of transformers: the standard transformer and the estimator transformer.

Standard transformer

A standard transformer transforms the input dataset into the output dataset by explicitly applying a transformation function to the input data. There is no dependency on the input data other than reading the input column and generating the output column.

Examples of standard transformers include Tokenizer, RegexTokenizer, StopWordsRemover, Binarizer, n-gram, HashingTF, SQLTransformer, VectorAssembler, and so on.

An illustration of a standard transformer is shown in the following figure, where the input column from an input dataset is transformed into an output column generating an output dataset:

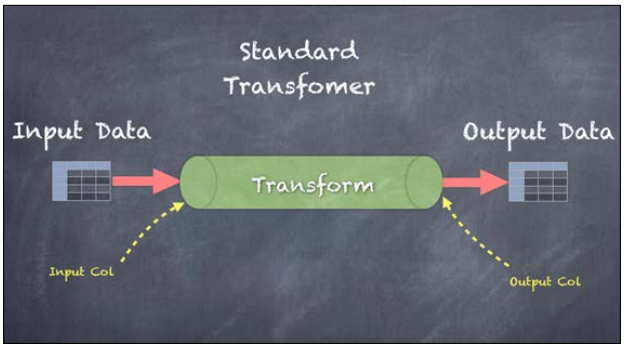


Figure 2: Using a transformer to transform text data (Karim et al., Scala and Spark for Big Data Analytics, Packt Publishing Ltd., 2017.)

Estimator transformer

An estimator transformer transforms the input dataset into the output dataset by first generating a transformer based on the input dataset. Then the transformer processes the input data, reading the input column and generating an output column in the output dataset. Such transformers are invoked as follows:

```
transformer = algorithm.fit(input_text)
output_text = transformer.transform(input_text)
```

Examples of estimator transformers include TF, IDF, TF-IDF, LDA, Word2Vec, and so on. An illustration of an estimator transformer is shown in the following figure, where the input column from an input dataset is transformed into an output column, generating an output dataset:

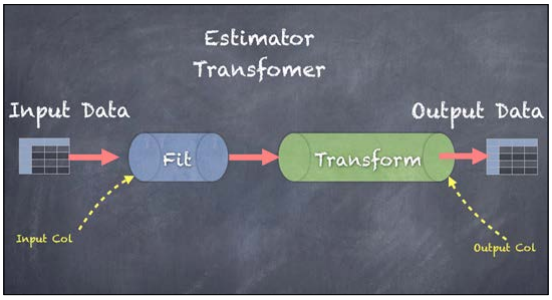


Figure 3: Using estimator to transform text data (Karim et al., Scala and Spark for Big Data Analytics, Packt Publishing Ltd., 2017.)

In the next few sections we will look deeper into text analytics using a simple example dataset that consists of lines of text (sentences) as shown in the following figure:

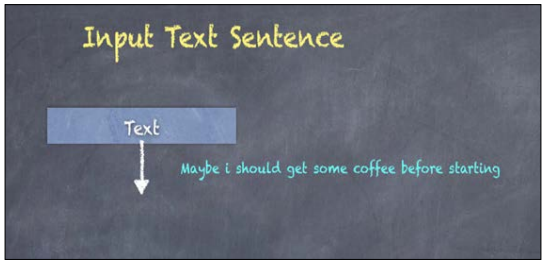


Figure 4: Raw input text to be transformed (Karim et al., Scala and Spark for Big Data Analytics, Packt Publishing Ltd., 2017.)

On the other hand, if you wanted to set up a regular expression based tokenizer, you would have to use the `RegexTokenizer` instead of `Tokenizer`. For this, you need to initialize a `RegexTokenizer` specifying the input column and the output column along with the regex pattern to be used:

```
>>> texts = ['Maybe i should get some coffee before starting']
>>> texts = [x.lower() for x in texts]
>>> texts = [''.join(c for c in x if c not in string.punctuation) for
x in texts]
>>> texts = [''.join(c for c in x if c not in '0123456789') for x in
texts]
>>> texts = [' '.join(x.split()) for x in texts]
```

A diagram of the tokenizer follows, which shows the sentence from input text split into words by using the space delimiter:

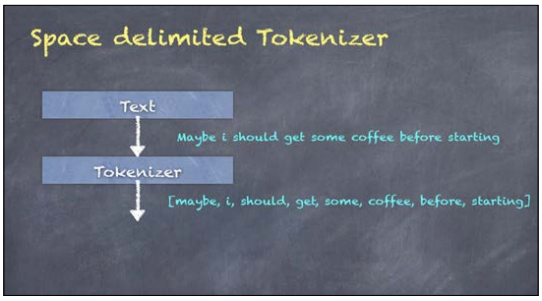


Figure 5: Using tokenizer as a transformer for NLP (Karim et al., Scala and Spark for Big Data Analytics, Packt Publishing Ltd., 2017.)

StopWordsRemover

StopWordsRemover is a transformer that takes a string array of words and returns a string array after removing all defined stop words. Some examples of stop words are I, you, my, and, or, and so on, which are fairly commonly used in the English language. You can override or extend the set of stop words to suit the purpose of the use case. Without this cleansing process, the subsequent algorithms might be biased because of the common words. First, you need to initialize a StopWordsRemover specifying the input column and the output column.

Here, we are choosing the words column created by the tokenizer and generating an output column for the filtered words after removal of stop words. Now let's remove the stop words:

```
>>> texts = 'Maybe i should get some coffee before starting'
>>> stop = set(['i', 'should', 'some', 'before'])
>>> texts = [i for i in texts.lower().split() if i not in stop]
```

The output is as follows:

```
[maybe, get, coffee, starting]
```

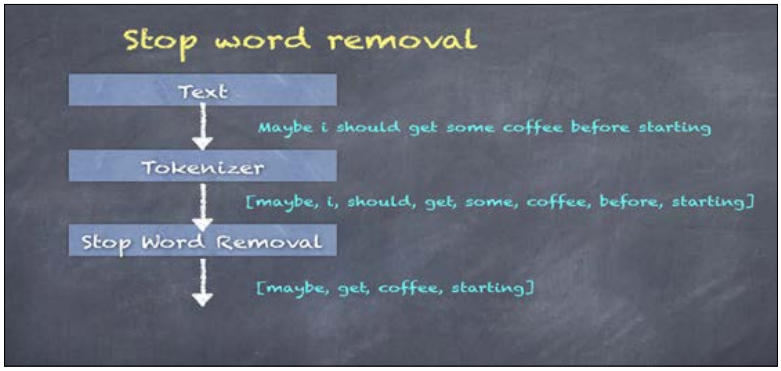


Figure 6: Using stop remover as a transformer for NLP (Karim et al., Scala and Spark for Big Data Analytics, Packt Publishing Ltd., 2017.)

Now finally, the filtered texts should be as follows:

```
filtered_texts = ' '.join(texts)
print(filtered_texts)
>>>
  maybe get coffee starting
```

N-gram

N-grams are word combinations created as sequences of words. N stands for the number of words in the sequence. For example, 2-gram is two words together and 3-gram is three words together. `setN()` is used to specify the value of N.

Now let's see some example code that shows bigram. Let's suppose our original text is "Maybe i should get some coffee before starting". The following code does it:

```
from nltk import ngrams
sentence = 'Maybe i should get some coffee before starting'
n = 2
sixgrams = ngrams(sentence.split(), n)
for grams in sixgrams:
  print(grams)

>>>
('Maybe', 'i')
('i', 'should')
('should', 'get')
('get', 'some')
('some', 'coffee')
('coffee', 'before')
('before', 'starting')
```

A diagram of the n-gram is as follows, which shows 2-grams generated from the sentence after tokenizing and removing stop words:

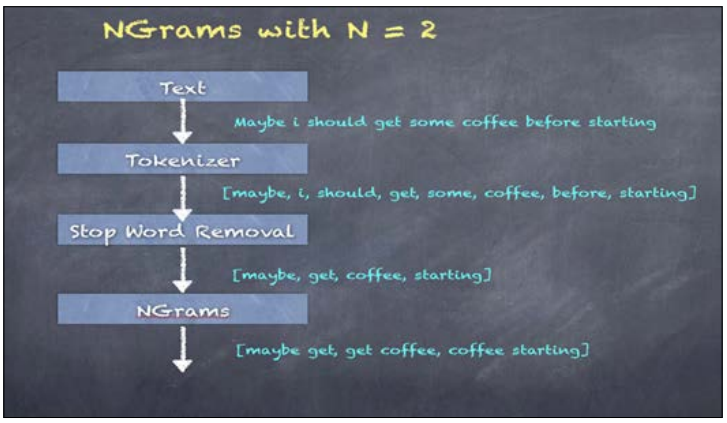


Figure 7: Using n-gram as a transformer for NLP (Karim et al., Scala and Spark for Big Data Analytics, Packt Publishing Ltd., 2017.)

Now that we have some background knowledge and importantly we know how to prepare and feed the data into TensorFlow, using these, can we develop an NLP pipeline for predictive analytics?

Using BOW for predictive analytics

In this section, we will see how to perform a bit more complex predictive analytics using the bag-of-words concept of NLP with TensorFlow. At first, we will formalize the problem, and then will explore the dataset that will be used. Finally, we will apply some visual analytics and machine learning techniques.

Bag-of-words

The BOW model is a simplifying representation where text (such as a sentence or a document) is represented as the bag of its words, disregarding grammar and even word order, but keeping multiplicity. The BOW model is commonly used in methods of document classification where the occurrence of each word is used as a feature for training a classifier.

The following models a text document using bag-of-words. Here are two simple text documents:

1. Asif loves watching action movies. Mary loves movies too.
2. Asif also loves watching football games.

   Based on these two text documents, a list can be constructed as follows:

```
[
    "Asif",
    "loves",
    "watching",
    "movies",
    "Mary",
    "too",
    "also",
    "football",
    "games"
]
```

In the preceding list, `loves` occurred three times, but we have counted only one time. You can think of a list as a set of only unique words. This list can be thought of as a bag-of-words. This bag is really very powerful and can help you develop good NLP applications.

The problem definition

In this section, we will build a spam classifier for classifying the raw text as spam or ham. The ultimate target is to predict if a given message is spam. The dataset that will be used is a collection of phone SMS that are spam or not spam (ham). I will show you how to download this dataset, store it for future use, and then proceed with the bag-of-words method to predict whether text is spam or not.

The model that will operate on the bag-of-words will be a logistic model with no hidden layers. We will use stochastic training, with a batch size of one, and compute the accuracy on a held-out test set at the end. We will also show you how to evaluate such models. At the end, the spam classifier model will help you distinguish between spam and ham messages. Figure 8 shows a conceptual view of two messages (spam and ham respectively):

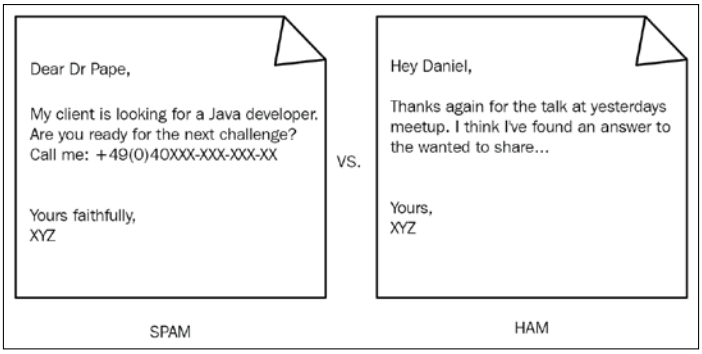


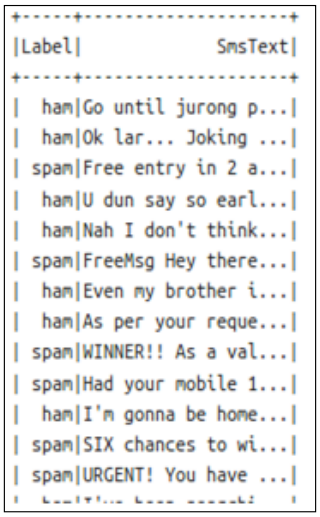
Figure 8: Spam and ham example

(Source: `https://blog.codecentric.de/en/2016/06/spam-classification-using-sparks-dataframes-ml-zeppelin-part-1/`)

We power some basic machine learning techniques to build and evaluate such a classifier for this kind of problem. In particular, logistic regression algorithms will be used for this problem.

The dataset description and exploration

The Spam dataset that we downloaded from `https://archive.ics.uci.edu/ml/datasets/SMS+Spam+Collection` consists of 5,574 SMS, which have been classified by hand as either ham or spam. Only 13.4% of these SMS are spam. This means the dataset is skewed and provides only a few examples of spam. This is something to keep in mind as it can introduce bias when training models:



Label	SmsText
ham	Go until jurong p...
ham	Ok lar... Joking ...
spam	Free entry in 2 a...
ham	U dun say so earl...
ham	Nah I don't think...
spam	FreeMsg Hey there...
ham	Even my brother i...
ham	As per your reque...
spam	WINNER!! As a val...
spam	Had your mobile 1...
ham	I'm gonna be home...
spam	SIX chances to wi...
spam	URGENT! You have ...

Figure 9: A snapshot of the SMS dataset

So what does this data look like? You might have seen that social media text can really get dirty and contains slang words such as "xxx", "fuck", "drug", "Viagra", misspelled words, missing whitespaces, abbreviated words such as "u", "urs", "yrs", and often violation of grammar rules. It may sometimes contain trivial words in the messages. Thus, we need to take care of these issues as well.

In the following steps, we will encounter these issues for better interpretation of the analytics using the LR classifier and BOW with TensorFlow.

Spam prediction using LR and BOW with TensorFlow

In this example, we will start by getting the data, normalizing and splitting the text, running it through an embedding function, and training the logistic function to predict spam. Now let's get started:

1. Importing necessary modules.

   The first task will be to import the necessary libraries for this task. Other than the usual libraries, we will need a module that can parse the zipped file in `.zip` format. We will use the zip file module to unzip the data from the UCI machine learning website link mentioned previously:

   ```
   import tensorflow as tf
   import matplotlib.pyplot as plt
   import os
   ```

```
import numpy as np
import pandas as pd
import csv
import string
import requests
import io
from zipfile import ZipFile
from tensorflow.contrib import learn
from tensorflow.python.framework import ops
ops.reset_default_graph()
```

2. Data download and parsing.

   Instead of downloading the text data every time the script is run, we will save it and check whether the file has been saved before. This prevents us from repeatedly downloading the data over and over if we want to change the script parameters. After downloading, we will extract the input and target data and change the target to one for spam and zero for ham:

```
save_file_name = os.path.join('temp','temp_spam_data.csv')
if not os.path.exists('temp'):
    os.makedirs('temp')

if os.path.isfile(save_file_name):
    text_data = []
    with open(save_file_name, 'r') as temp_output_file:
        reader = csv.reader(temp_output_file)
        for row in reader:
            text_data.append(row)
else:
    zip_url = 'http://archive.ics.uci.edu/ml/machine-learning-
databases/00228/smsspamcollection.zip'
    r = requests.get(zip_url)
    z = ZipFile(io.BytesIO(r.content))
    file = z.read('SMSSpamCollection')
    # Format Data (In python 2.7.13 on Mac, decode('utf8') is
required.
    text_data = file.decode()
    text_data = text_data.encode('ascii',errors='ignore')
    text_data = text_data.decode().split('\n')
    text_data = [x.split('\t') for x in text_data if len(x)>=1]

    with open(save_file_name, 'w') as temp_output_file:
        writer = csv.writer(temp_output_file)
        writer.writerows(text_data)
```

3. Text to numerical labeling.

   At first we separate the feature, that is, the text, and the labels as follows:

   ```
   texts = [x[1] for x in text_data]
   label = [x[0] for x in text_data]
   ```

   Then let's relabel `spam` as `1` and `ham` as `0` as follows:

   ```
   target = [1 if x=='spam' else 0 for x in label]
   ```

4. Creating bag-of-words through pre-processing.

   As you know, words might have many unwanted characters, punctuation, and special characters. Moreover, the text might mix both lower and uppercase letters. Thus, we need to handle such issues. The ultimate target is to reduce the potential vocabulary size by normalizing the text. At first, let's convert the text to lowercase as follows:

   ```
   texts = [x.lower() for x in texts]
   ```

   Now let's remove the punctuation and numbers, including special characters as follows:

   ```
   texts = [''.join(c for c in x if c not in string.punctuation) for
   x in texts]
   texts = [''.join(c for c in x if c not in '0123456789') for x in
   texts]
   ```

   Note that the preceding two lines of code will introduce some unwanted whitespace and we need to take care of that,too. Let's do it as follows:

   ```
   texts = [' '.join(x.split()) for x in texts]
   ```

5. Observing number of words in each SMS.

   We must also determine the maximum sentence size. To do this, we will look at a histogram of text length in the dataset. We can see that a good cut-off might be around 25 words. Use the following code:

   ```
   text_lengths = [len(x.split()) for x in texts]
   text_lengths = [x for x in text_lengths if x < 50]
   plt.hist(text_lengths, bins=50)
   plt.title('Histogram of # of Words in Texts')
   plt.show()
   ```

The output is a histogram of the number of words in each text in our data. We use this to establish a maximum length of words to consider in each text. We set this as 25 words, but it can easily be set as 30 or 40 as follows:

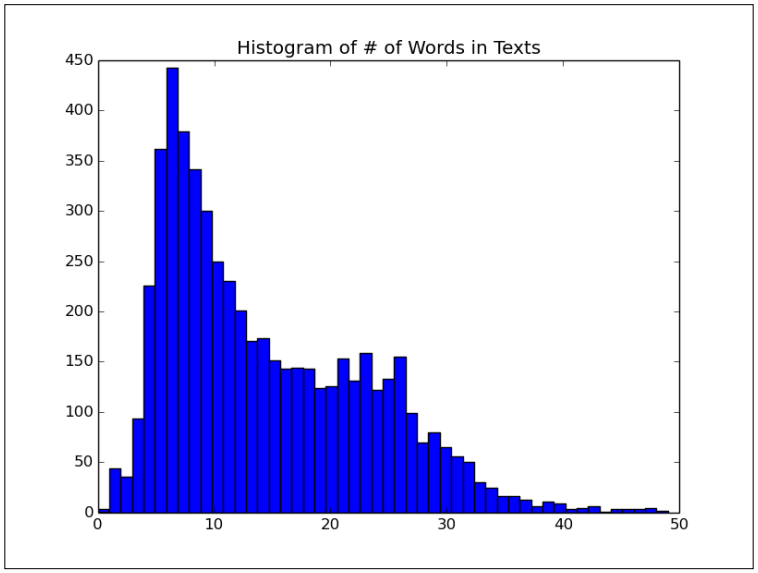


Figure 10: A histogram of the number of words in each text in our data

6. Vocabulary embedding.

   I used the TensorFlow built-in processing tool for determining vocabulary embedding called `VocabularyProcessor()`:

```
text_size = 25
min_word_freq = 3
```



Setting the prior max text word length

In the preceding code, we have set the max text word length as 25. This is common practice with a bag-of-words because it limits the effect of text length on the prediction, since spam may be hard to find and ham may be easy to find. Because of this fact, the vocabulary that we create might be heavily skewed towards words represented in the ham part of our data. If we allow unlimited lengths for texts, then spammers might take advantage of this and create very long texts, which have a higher probability of triggering non-spam word factors in our logistic model.

We need to set up the vocabulary processor:

```
processor = learn.preprocessing.VocabularyProcessor(text_size,
min_frequency=min_word_freq)
```

Then we use the transformer's `fit_transform()` function to get the length of unique words:

```
vocab_processor= processor.fit_transform(texts)
```

Finally, we compute the length of the embedded vocabulary that will be used for one-hot-encoding:

```
embedding_size = len(vocab_processor.vocabulary_)
```

7. Training and test set preparation.

   I split the pre-processed text data into training and test sets. More precisely, 75% goes to training and the remaining 25% goes to test set. However, we keep the features separate from the labels. At first, we create the test and train indices as follows:

   ```
   train_indices = np.random.choice(len(texts),
   round(len(texts)*0.75), replace=False)

   test_indices = np.array(list(set(range(len(texts))) - set(train_
   indices)))
   ```

   Let's create two separate sets as training and test set (75% for training and 15% for testing) using `train_test_split()` function from sklearn:

8. Creating encoded vectors using up one-hot-encoding:

   ```
   texts_train, texts_test, target_train, target_test = train_test_
   split(texts, target, train_size = 0.75)
   ```

   In the preceding code, I have declared the embedding matrix for the bag-of-words. Words in each sentence will be translated into indices. These indices will be translated into one-hot-encoded vectors using an identity matrix. Therefore, the size of the word embedding will be equal to the length of the encoded vector.

   Later on, we will use this matrix to look up the sparse vector for each word and add them together for the sparse sentence vector. Let's set up an identity matrix for one-hot-encoding:

   ```
   identity_mat = tf.diag(tf.ones(shape=[embedding_size]))
   ```

9. Creating variables and initializing placeholders for LR.

   Since I'm going to use LR classifier to predict the probability of spam, let's declare the variables through TensorFlow's placeholder's list:

   ```
   A = tf.Variable(tf.random_normal(shape=[embedding_size,1]))
   b = tf.Variable(tf.random_normal(shape=[1,1]))
   ```

   Now, before initializing those variables, if you look at the preceding code carefully, you should understand why `x_data` input placeholder should be of type integer. The reason is that we will use this tolook up the row index of our identity matrix. Moreover, TensorFlow requires that lookup to be an integer:

   ```
   x_data = tf.placeholder(shape=[text_size], dtype=tf.int32)
   ```

   However, the predictors should be of type float since they contain the labels that are in floating point type:

   ```
   y_target = tf.placeholder(shape=[1, 1], dtype=tf.float32)
   ```

10. Text-vocabulary embedding using lookup.

    Now, we use TensorFlow's embedding lookup function that will map the indices of the words in the sentence to the one-hot-encoded vectors of our identity matrix:

    ```
    x_embed = tf.nn.embedding_lookup(identity_mat, x_data)
    ```

    We have that matrix; we should create the sentence vector by summing up the aforementioned word vectors as follows:

    ```
    x_sums = tf.reduce_sum(x_embed, 0)
    ```

11. Declaring model operations.

    In step 10, we saw that our fixed-length sentence vectors for each sentence were ready. However, before we train the LR classifier, we have to declare the actual model operation that signifies what the input and the corresponding output should be:

    ```
    x_sums_2D = tf.expand_dims(x_sums, 0)
    model_output = tf.add(tf.matmul(x_sums_2D, A), b)
    ```

    Here, we will be doing the stochastic training, so we have to expand the dimensions of our input and perform regression operations on it.

12. Declaring the objective, that is, loss function:

    ```
    loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_
    logits(logits=model_output, labels=y_target))
    ```

13. Defining the prediction operation and optimizer.

    Since we are talking about the LR classifier, we will use the sigmoid function as the prediction operation as follows:

```
prediction = tf.sigmoid(model_output)
redictions_correct = tf.cast(tf.equal(prediction, y_target),
tf.float32)
accuracy = tf.reduce_mean(predictions_correct)
```

    Now let's use the GradientDescentOptimizer, which is the optimizer for heavy lifting, and with the following setting, we gain better predictive accuracy:

```
train_op = tf.train.GradientDescentOptimizer(0.01)
train_step =train_op.minimize(loss)
```

    The `0.01` that we used here is the step `GradientDescentOptimizer` takes to try to learn a better value.



Which optimizer to be used

You can use other optimizers as well. Here I have listed some widely used optimizers:

`AdagradOptimizer`

`MomentumOptimizer`

`AdamOptimizer`

`FtrlOptimizer`

`RMSPropOptimizer`

However, I recommend using `GradientDescentOptimizer` unless it is failing. In this section, and in upcoming sections, we will see that in some cases `AdamOptimizer` works pretty well too. Mostly, it depends on the nature of the data. If the data you will be using for the predictive modeling is good, both Gradient Descent and Adam work well. Furthermore, it would be worth trying and switching between different optimizers and choosing the one that generalizes better.

14. Create a TensorFlow session and initialize global variables:

```
sess = tf.Session()
Init_op = tf.global_variables_initializer()
sess.run(init_op)
```

15. Training the LR model.

    Now we will start the iteration over each text, that is, sentence. The `fit()` transformer is a generator that operates one sentence at a time in a stochastic way.

    I observed the best result after 75 iterations with accuracy of 85%, whereas after 50 iterations the accuracy was90% and 88% with Adam and Gradient Descent optimizer respectively.

    We will see, in the next step while evaluating the model:

```
print('Starting Training Over {} Sentences.'.format(len(texts_
train)))
loss_vec_train = []
train_acc_all = []
train_acc_avg = []
for ix, t in enumerate(vocab_processor.fit_transform(texts_
train)):
    y_data = [[target_train[ix]]]
    sess.run(train_step, feed_dict={x_data: t, y_target: y_data})
    temp_loss = sess.run(loss, feed_dict={x_data: t, y_target: y_
data})
    loss_vec_train.append(temp_loss)
    if (ix+1)%10==0:
        print('Training Observation #' + str(ix+1) + ': Loss = ' +
str(temp_loss))
    [[temp_pred]] = sess.run(prediction, feed_dict={x_data:t, y_
target:y_data})
    train_acc_temp = target_train[ix]==np.round(temp_pred)
    train_acc_all.append(train_acc_temp)
    if len(train_acc_all) >= 75:
        train_acc_avg.append(np.mean(train_acc_all[-75:]))
```

    You should observe the preceding output.

16. Plotting the error in training.

    In the preceding code segment, we computed the `loss_vec_test` that stores the value of training loss for each step. Now it's time to observe it visually.

    Just execute the following code:

```
with warnings.catch_warnings():
  warnings.simplefilter("ignore", category=RuntimeWarning)
  plt.plot([np.mean(loss_vec_train[i-50:i]) for i in
range(len(loss_vec_train))])
  plt.show()
```

Here, we keep the plotting function inside the warning catch block so that it can handle possible runtime warnings. From the following figure, it is clear that the training loss, that is, error, decreases when iteration increases:

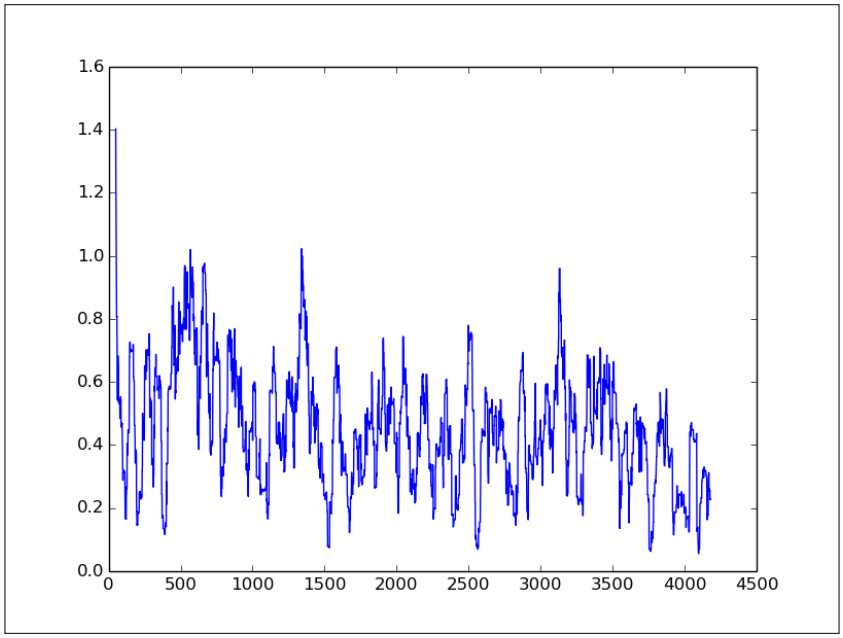


Figure 11: Training errors across all the iterations with the Gradient Descent optimizer

In the preceding code segment, you can see that I have taken the windowed average using `np.mean(errors[i-50: i])` instead of just using `errors[i]`. The reason is that we are only performing a single test inside the loop. Therefore, while the error tends to decrease, it bounces around quite a bit. Taking this average smooths it out a bit, but it still jumps around in a bumpy way.

However, changing the optimizer to `AdamOptimizer` produces the following graph where the jumping around is a more bumpy:

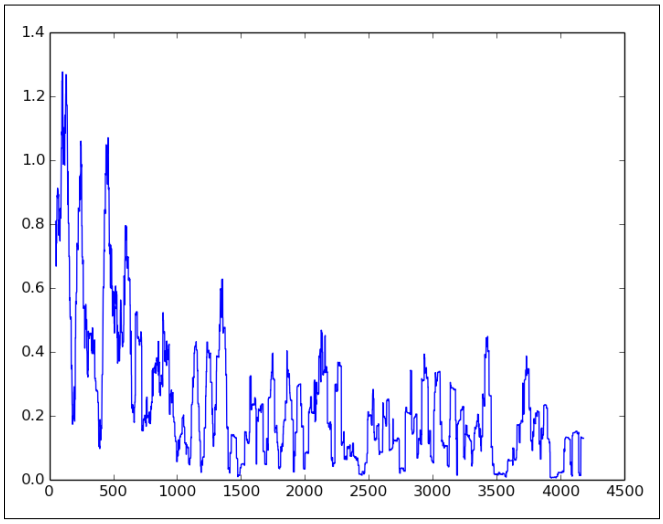


Figure 12: Training errors across all the iterations with the Adam optimizer

Note that if you want to switch to Adam optimizer, just change one line in step 13 as follows (also applies for the subsequent steps in this section):

```
train_op = tf.train.AdamOptimizer(0.01)
```

17. Evaluating the model on the test set.

    To get the predictive power of the LR model in terms of accuracy, we repeat the previous step, except on the prediction operation and with the test set only:

```
print('Test set accuracy over {} sentences:- '.format(len(texts_
test)))
test_acc_all = []
for ix, t in enumerate(vocab_processor. transform(texts_test)):
    y_data = [[target_test[ix]]]
    if (ix+1)%10==0:
        print('Test observation: ' + str(ix+1))
    [[temp_pred]] = sess.run(prediction, feed_dict={x_data:t, y_
target:y_data})
    test_acc_temp = target_test[ix]==np.round(temp_pred)
    test_acc_all.append(test_acc_temp)

print('\nOverall accuracy on test set (%): {}'.format(np.
mean(test_acc_all)*100.0))
```

Let's see in how many cases our LR model predicted correctly whether or not a text was spam or ham:

```
number_of_correct_predicted_words = len(texts_train) -
np.sum(test_acc_all)
print('Number of correctly predicted texts on test set: {}'.
format(number_of_correct_predicted_words))

>>>
Overall accuracy on test set (%): 88.30703012912483
Number of wrongly predicted texts on test set: 163
```

This is outstanding. However, using `GradientDescentOptimizer`, the accuracy that I observed was `84.02008608321378%`.`AdamOptimizer` doesn't generalize in some cases too. We will see an example of that in the next section.

18. Plotting the predictive accuracy over time:

```
plt.plot(range(len(train_acc_avg)), train_acc_avg, 'k-',
label='Training accuracy')
plt.title('Average training accuracy over 75 iterations')
plt.xlabel('Iteration')
plt.ylabel('Training accuracy')
plt.show()
>>>
```

The output is as follows:

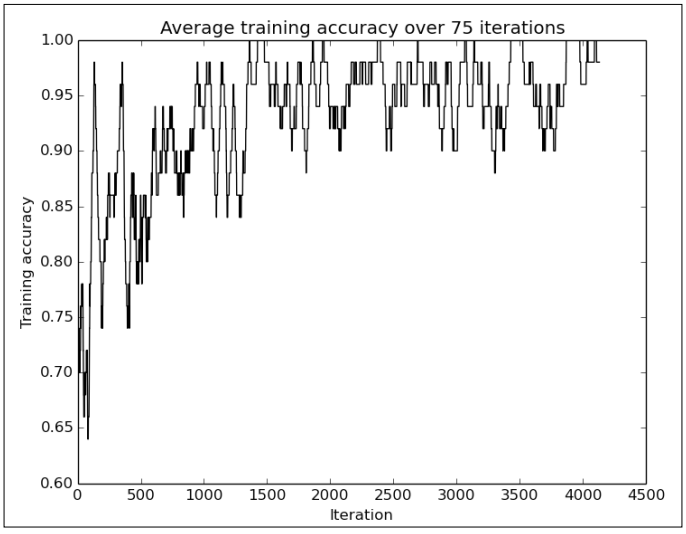


Figure 13: Test set accuracy over time using Adam optimizer

On the other hand, figure 14 shows the training errors across all the iterations with the Adam optimizer:

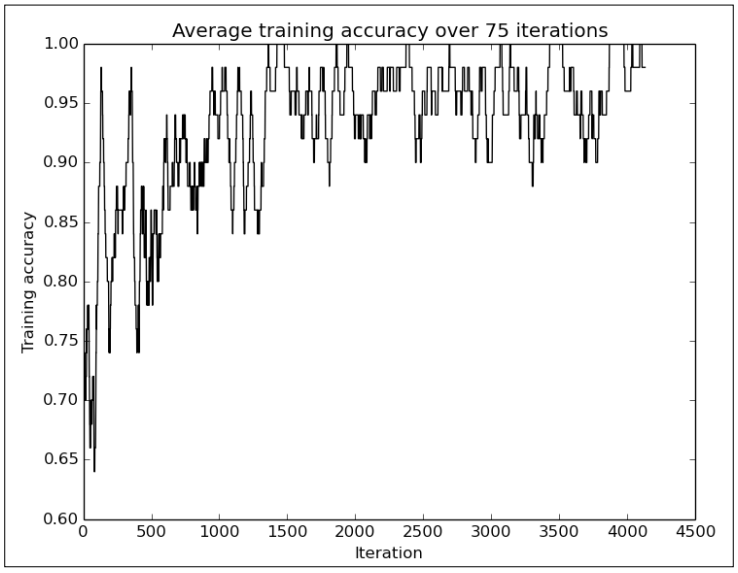


Figure 14: Training errors across all the iterations with the Adam optimizer

From the preceding figures, it is clear that the vocabulary that we have created might be heavily skewed towards words represented in the ham part of our data. One potential solution is allowing unlimited length for texts. However, spammers might take advantage of this and create very long texts, which have a higher probability of triggering non-spam word factors in our logistic model.

To tackle this issue in a better way, in the next section, we will see how to use frequency of word occurrences to determine the values of word embedding.

TF-IDF model for predictive analytics

TF-IDF measures how important a word is in a document or in a collection of documents. It is used extensively in informational retrieval and reflects the weight of the word in the document. TF-IDF values increase in proportion to a number of occurrences of the words, otherwise known as the frequency of the word/term, and consists of two key elements, the term frequency and the inverse document frequency.

How to compute TF, IDF, and TFIDF?

TF is the term frequency, which is the frequency of a word/term in the document. For a term t, TF measures the number of times term t occurs in document d. The TF can be implemented using hashing where a term is mapped into an index by applying a hash function. On the other hand, IDF is the inverse document frequency that represents the information a term provides about the tendency of the term to appear in documents.

IDF is a log scaled inverse function of documents containing a particular term or word:

```
IDF = TotalDocuments / Documents containing Term
```

Once we have TF and IDF, we can compute the TF-IDF value by multiplying them:

```
TF-IDF = TF * IDF
```

A diagram of the TF-IDF is shown as follows, which shows the generation of TF-IDF features:

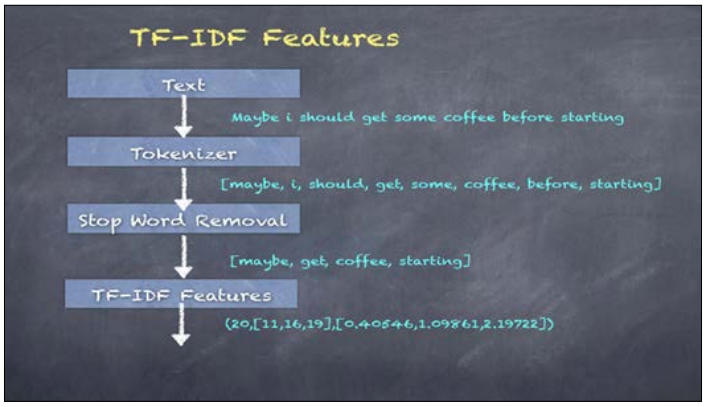


Figure 15: Using TF-IDF to generate feature vectors for NLP (Karim et al., Scala and Spark for Big Data Analytics, Packt Publishing Ltd., 2017.)

We will now look at how we can generate TF using a user-defined function to be used as a transformer in Python. Execute TF_IDF_example.py for this example in Python 3. Note that you need to install a Python module named `textblob` for breaking up the text into words and getting the word counts as follows:

```
import math
from textblob import TextBlob as tb
```

The method for computing TF iss as follows:

```
def tf(word, doc):
    return doc.words.count(word) / len(doc.words)
```

The preceding function computes "term frequency" which is the number of times a word appears in a document doc, normalized by dividing by the number of words in the document. Now we can compute the IDF with the following function:

```
def idf(word, doclist):
    return math.log(len(doclist) / (1 + n_contatin(word, doclist)))
```

The preceding function computes "inverse document frequency". This signifies how common a word is in a list of documents. More naively, the more common a word is, the lower its `idf`. In the preceding code, we took the ratio of the total number of documents in the list to the number of documents containing that particular word. Finally, we have taken the log of the outcome.

In the preceding function, we have used another user-defined function named `n_contain()`. This function takes a word from the list of documents and returns the number of documents containing that word. This function is as follows:

```
def n_contain(word, doclist):
    return sum(1 for doc in doclist if word in blob.words)
```

Finally, we can compute the `tfidf` since we have both the function for computing `tf` and `idf` respectively. The `tfidf` function is as follows:

```
def tfidf(word, doc, doclist):
    return tf(word, doc) * idf(word, doclist)
```

The preceding fucntion computes the TF-IDF score. It is simply the product of `tf` and `idf` that we computed using the preceding `tf` and `idf` functions. Now let's look at an example of the preceding implementation:

```
for i, blob in enumerate(blobList):
    print("Top words in document {}".format(i+1))
    scores = {word: tfidf(word, blob, blobList) for word in blob.
words}
    sorted_words = sorted(scores.items(), key=lambda x: x[1],
reverse=True)
    for word, score in sorted_words[:5]:
        print("\tWord: {}, TF-IDF: {}".format(word, round(score, 10)))
```

Suppose we have three documents containing some words as follows (the following text has been reused from the Wikipedia page of TensorFlow, at `https://en.wikipedia.org/wiki/TensorFlow`):

- `doc1 = tb`: TensorFlow is an open source software library for machine learning across a range of tasks, and developed by Google to meet their needs for systems capable of building and training neural networks to detect and decipher patterns and correlations, analogous to the learning and reasoning that humans use.
- `doc2 = tb`: It is currently used for both research and production at Google products, often replacing the role of its closed-source predecessor, `DistBelief`. TensorFlow was originally developed by the Google brain team for internal Google use before being released under the Apache 2.0 open source license on November 9, 2015.
- `doc3 = tb`: Starting in 2011, Google brain built `DistBelief` as a proprietary machine learning system based on deep learning neural networks. Its use grew rapidly across diverse Alphabet companies in both research and commercial applications.

Now we should create the document as a list:

```
blobList = [doc1, doc2, doc3]

for i, blob in enumerate(blobList):
    print("Top words in document {}".format(i+1))
    scores = {word: tfidf(word, blob, blobList) for word in blob.
words}
    sorted_words = sorted(scores.items(), key=lambda x: x[1],
reverse=True)
    for word, score in sorted_words[:5]:
        print("\tWord: {}, TF-IDF: {}".format(word, round(score, 10)))
>>>
Top words in document 1
  Word: TensorFlow, TF-IDF: -0.0059933765
  Word: is, TF-IDF: -0.0059933765
  Word: an, TF-IDF: -0.0059933765
  Word: open, TF-IDF: -0.0059933765
  Word: source, TF-IDF: -0.0059933765
Top words in document 2
  Word: It, TF-IDF: -0.0058710627
  Word: is, TF-IDF: -0.0058710627
  Word: currently, TF-IDF: -0.0058710627
  Word: used, TF-IDF: -0.0058710627
  Word: both, TF-IDF: -0.0058710627
```

```
Top words in document 3
  Word: Starting, TF-IDF: -0.0087176386
  Word: 2011, TF-IDF: -0.0087176386
  Word: Google, TF-IDF: -0.0087176386
  Word: Brain, TF-IDF: -0.0087176386
  Word: built, TF-IDF: -0.0087176386
```

However, as you can see, the preceding output contains the stop words and numbers. Those can be removed too. Fortunately, the sklearn Python module provides `TfidfVectorizer` as a powerful feature-extraction method for text that takes care of these automatically and creates a feature vector that can be fed by the predictive models.

This function takes several parameters, as follows:

```
TfidfVectorizer(analyzer='word', binary=False, decode_
error='strict',dtype=<class 'numpy.int64'>, encoding='utf-8',
input='content',lowercase=True, max_df=1.0, max_features=1000,
min_df=1,ngram_range=(1, 1), norm='l2', preprocessor=None, smooth_
idf=True,stop_words='english', strip_accents=None, sublinear_tf=False,
token_pattern='(?u)\\b\\w\\w+\\b', tokenizer=<function tokenizer at
0x7f680a89e0d0>, use_idf=True,vocabulary=None)
```

Well, don't worry much about seeing so many parameters. We can even solve our problems by inputting a minimum number of parameters such as a `tokenizer`, stop words, and the number of maximum features that are going to be converted into feature vectors against each label. Something like the following:

```
tfidf = TfidfVectorizer(tokenizer=tokenizer, stop_words='english',
max_features=max_features)
```

We can ignore the rest, as your Python compiler will pickup the default value automatically. We can define a user-defined function for the `tokenizer` as follows:

```
def tokenizer(text):
    words = nltk.word_tokenize(text)
    return words
```

Implementing a TF-IDF model for spam prediction

In the previous section, we saw how we could develop a predictive model using bag-of-words that efficiently predicts the probability of a message being spam. However, we already mentioned that the simple bag-of-words has several issues. Now we will see how we can improve the previous predictive model with minimum effort using TF-IDF.

Note that I am not going to show every step, as I did previously; rather I will show the minimal code and steps for making this happen. First, we need to import the necessary packages as follows:

```
import tensorflow as tf  # TensorFlow
import matplotlib.pyplot as plt # For matplotlib for plotting
import csv # For parsing and pre-processing CSV file
import numpy as np # For the NumPy array
import os # For the regular OS support
import string # For string manipulation
import requests # For handling HTTP request
import io # For the I/O operation
import nltk # For nltk and its pre-trained tokenizer models
from nltk.corpus import stopwords # For removing stop-words
from nltk import word_tokenize,sent_tokenize # For the nltk
from zipfile import ZipFile # For handling zipped files
from sklearn.feature_extraction.text import TfidfVectorizer
from tensorflow.python.framework import ops # For TensorFlow
```

For this example, we will use `nltk` module for `nltk` and its pre-trained tokenizer models. I am assuming you have already installed `nltk` module on your system. If not, use `sudo pip3 install -U nltk` to do so (for Python2 use `pip`).

Now, to download a particular dataset or model while working with `nltk`, use the `nltk.download()` function. If you want to download specific sentence tokenizer such as `punkt`, use the following command (for the sentence tokenizer use `all`):

```
$ python3
>>> import nltk
>>> nltk.download('punkt')
```

Now, after step 4 and before step 5 (from the previous example) use the following code segment for implementing TF-IDF:

```
def tokenizer(text):
    words = nltk.word_tokenize(text)
    return words

tfidf = TfidfVectorizer(tokenizer=tokenizer, stop_words='english',
max_features=max_features)
print(tfidf)
```

Next, the training and test preparation (step 7 in the previous example) goes similarly, with minimal change. However, since now we will be dealing with the feature vector generated by the TF-IDF, we need to consider the shape instead of length, as follows:

```
train_indices = np.random.choice(sparse_tfidf_texts.shape[0],
round(0.75*sparse_tfidf_texts.shape[0]), replace=False)
test_indices = np.array(list(set(range(sparse_tfidf_texts.shape[0])) -
set(train_indices)))
```

The rest goes as follows:

```
texts_train = sparse_tfidf_texts[train_indices]
texts_test = sparse_tfidf_texts[test_indices]
target_train = np.array([x for ix, x in enumerate(target) if ix in
train_indices])
target_test = np.array([x for ix, x in enumerate(target) if ix in
test_indices])
```

We don't need to create encoded vectors using one-hot-encoding (step 8) or any other algorithm since we have seen that our text dataset has been converted into a feature vector.

Next, we create and initialize variables and placeholders for the LR model (step 9). We don't need to use TensorFlow's embedding lookup (step 10) function for mapping the indices of the words in the sentence to the one-hot-encoded vectors of our identity matrix, since they are already converted into a sparse feature vector.

So, what's next? Well, we need to declare the model operation (step 11), but since we already have our feature vector with appropriate dimensions, we don't need to expand the size of the feature vector, nor are we using stochastic training.

This means we will use our `x_data` variable straightaway without any intermediate conversion, as follows:

```
model_output = tf.add(tf.matmul(x_data, A), b)
```

Now, step 12, 13, and 14 go as same as the previous example. Now it's time to train the LR model. The following code also contains necessary steps for evaluating the predictive model on the test set:

```
train_loss = []
test_loss = []
train_acc = []
test_acc = []
i_data = []
# Start a graph session
```

```
sess = tf.Session()
for i in range(10000):
    rand_index = np.random.choice(texts_train.shape[0], size=batch_
size)
    rand_x = texts_train[rand_index].todense()
    rand_y = np.transpose([target_train[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})
    # Only record loss and accuracy every 100 generations
    if (i+1)%100==0:
        i_data.append(i+1)
        train_loss_temp = sess.run(loss, feed_dict={x_data: rand_x,
y_target: rand_y})
        train_loss.append(train_loss_temp)
        test_loss_temp = sess.run(loss, feed_dict={x_data: texts_test.
todense(), y_target:
target_test.reshape(-1,1)})
        test_loss.append(test_loss_temp)
        train_acc_temp = sess.run(accuracy, feed_dict={x_data: rand_x,
y_target: rand_y})
        train_acc.append(train_acc_temp)
        test_acc_temp = sess.run(accuracy, feed_dict={x_data: texts_
test.todense(), y_target:
target_test.reshape(-1,1)})
        test_acc.append(test_acc_temp)
    if (i+1)%500==0:
        acc_and_loss = [i+1, train_loss_temp, test_loss_temp, train_
acc_temp, test_acc_temp]
        acc_and_loss = [np.round(x,2) for x in acc_and_loss]
        print('Generation # {}. Train Loss (Test Loss): {:.2f}
({:.2f}). Train Acc (Test Acc): {:.2f} ({:.2f})'.format(*acc_and_
loss))
```

If you look at the preceding code, you will see that we had to convert the texts into dense vectors since TensorFlow cannot process your data in sparse vector format. Now you can measure the accuracy using the following code:

```
>>>
Overall accuracy on the training set (%): 78.01499962806702
Overall accuracy on the test set (%): 80.93757629394531
```

However, after switching to Adam optimizer, you should observe better accuracy as follows:

```
>>>
Overall accuracy on the training set (%): 88.60499858856201
Overall accuracy on the test set (%): 83.04159641265869
```

Now let's see how our training and testing phase went. We will plot the error in the training and test phase. Just use the following code that plots the loss over time:

```
plt.plot(i_data, train_loss, 'k-', label='Training loss')
plt.plot(i_data, test_loss, 'r--', label='Test loss', linewidth=4)
plt.title('Cross entropy loss per iteration')
plt.xlabel('Iteration')
plt.ylabel('Cross entropy loss')
plt.legend(loc='upper right')
plt.show()
>>>
```

The output is as follows:

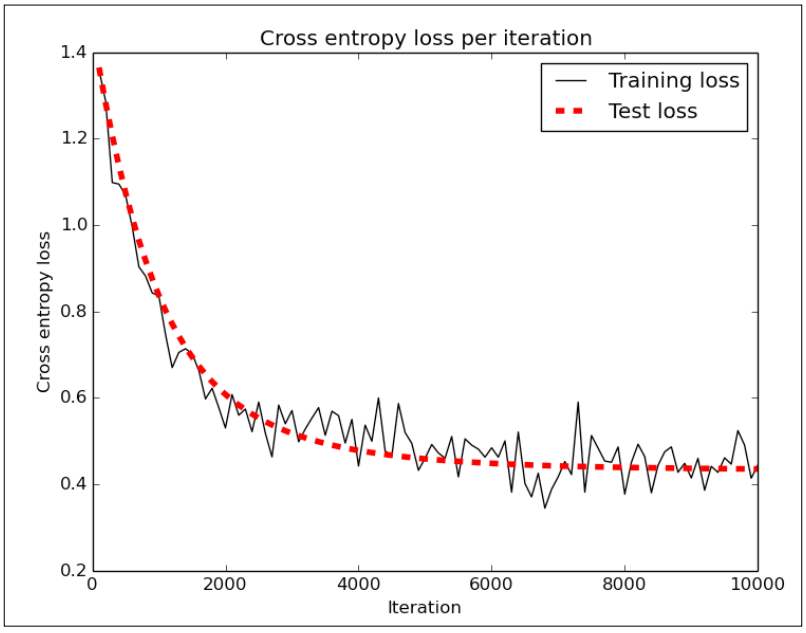


Figure 16: Cross entropy loss in the training and test set using Gradient Descent optimizer

However, changing the optimizer to Adam optimizer produces the following graph, where the jumping around is more bumpy:



Figure 17: Cross-entropy loss in the training and test set using Adam optimizer

From the preceding diagram, it is clear that although we observe better accuracy and cross entropy per iteration decreased for the training set over time, however, on the test set it performs worst. It might be because the test set has a number of words that are much smaller than that of the training set. Now let's plot accuracy over time on the train and test sets using the following code:

```
plt.plot(i_data, train_acc, 'k-', label='Accuracy on the training
set')
plt.plot(i_data, test_acc, 'r--', label='Accuracy on the test set',
linewidth=4)
plt.title('Accuracy on the train and test set')
plt.xlabel('Generation')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.show()
```

The output is as follows:

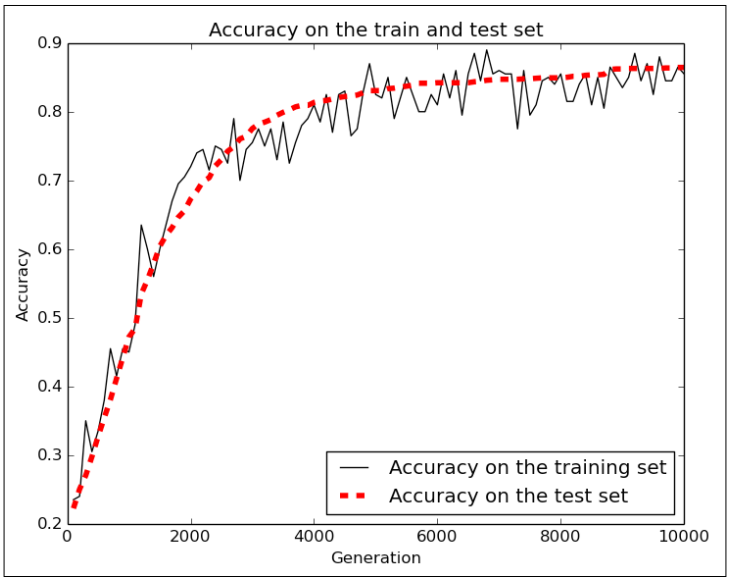


Figure 18: Prediction accuracy on the test set using LR model that is optimized by Gradient Descent optimizer

However, after switching to Adam optimizer, you should observe the same graph bit turmoil as follows:

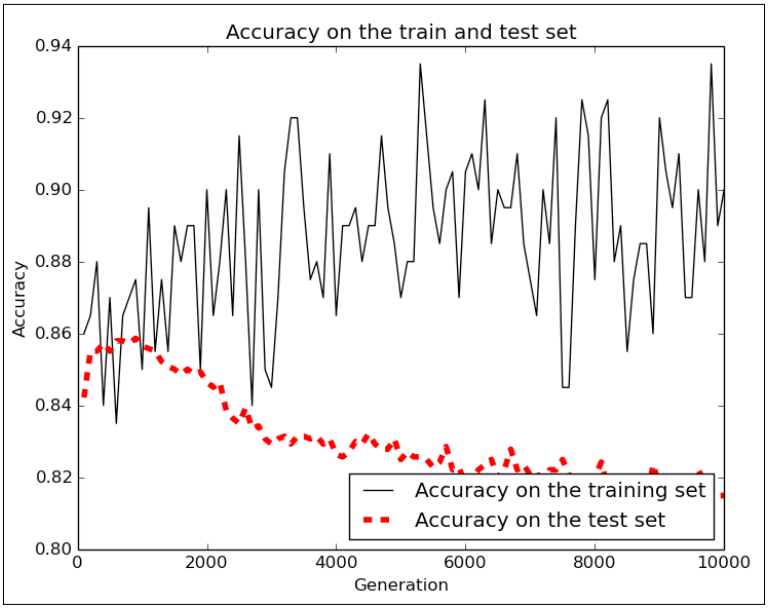


Figure 19: Prediction accuracy on the test set using LR model that is optimized by Adam optimizer

Again, over time, it performs worst on the test set. To tackle this issue in a better way, in the next section, we will see how to use frequency of word occurrence to determine the values of word embedding.

Using Word2vec for sentiment analysis

In the previous sections, we addressed the issue of word importance without considering the importance of word ordering in a collection of documents or in a single document. While using BOW and TF-IDF, we have seen that all words got projected into the same position and their vectors are averaged.

As the order of words in the history does not influence the projection, both BOW and TF-IDF have no such features that can take care of this issue. We will attempt to address this in the next few sections, which will introduce us to Word2vec techniques. More technically, we verbalized our textual embedding before training the BOW or TF-IDF models. However, using neural networks, we can make the embedding values part of the training procedure.

Continuous bag-of-words

In this section, we will see how to build a log-linear classifier utilizing four upcoming and four past words as the input in a sliding window manner; the training criterion is to correctly classify the current word using two methods, namely CBOW and skim-gram. This section is based on two sources:

- Tomas Mikolov et al, *Efficient Estimation of Word Representations in Vector Space*, `https://arxiv.org/pdf/1301.3781.pdf`
- Nick M., *TensorFlow Machine Learning Cookbook*, Packt Publishing Ltd.

Note that Tomas M. et al. proposed this model, named as continuous bag-of-words or. CBOW, since unlike the standard BOW model, it uses continuously distributed representations of the context. The training complexity of such models becomes:

$$Q = N \times D + D \times \log_2(V) \text{-------} (i)$$

In the above equation, *N* previous words are encoded using 1-of-V coding, where V is size of the vocabulary. The input layer is then projected to a projection layer P that has dimensionality *N*×*D*, using a shared projection matrix.

Continuous skip-gram

The second architecture is similar to CBOW, but instead of predicting the current word based on the context called continuous skim-gram, this method tries to maximize the classification of a word based on another word in the same sentence. More precisely, each current word is used as an input for a log-linear classifier with continuous projection layers, and predicting of words is done within a certain range before (behind) and after the current word (ahead).

An increase in the range improves the quality of the resulting word vectors, but it also increases the computational complexity. Since more distant words are usually less related to the current word than those close to it, we give less weight to the distant words by sampling less from those words in our training examples.

As a result of model building and prediction, times also increase. That can be represented as follows:

$$Q = C \times \left(D + D \times \log_2 \left(V\right)\right), -------\left(ii\right)$$

C is the maximum distance of the words, and from the algorithmic notation, this is higher than that of CBOW algorithm. And also if ($D + D_x log_2 (V)$) tends to be greater than C, the complexity tends to be exponential too.

A comparative analysis from an architecture point of view can be seen in the following figure; architecture predicts the current word based on the context, and the skip-gram predicts surrounding words given the current word:

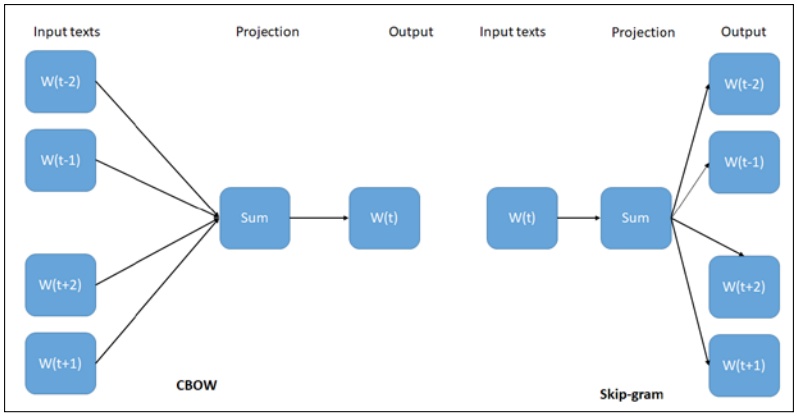


Figure 20: CBOW versus skip-gram

Due to page limitations, we will only demonstrate how to develop a predictive model using CBOW for sentiment prediction from a set of preset words called valid words.

Using CBOW for word embedding and model building

In this sub-section, we will implement a predictive model for movie review sentiment analysis and prediction using the CBOW method. The movie review data that I'm going to use is downloaded from the Cornell University movie review dataset from `http://www.cs.cornell.edu/people/pabo/movie-review-data/`. A minimal description of this dataset is a mandate.

The dataset contains 5,331 positive and 5,331 negative processed sentences/snippets donated by Introduced in Pang/Lee ACL 2005. More specifically, there are two files, namely `rt-polarity.pos` and `rt-polarity.neg`. The `rt-polarity.pos` contains 5,331 positive snippets and `rt-polarity.neg` contains 5,331 negative snippets.

Each line in these two files corresponds to a single snippet usually containing roughly one single sentence; all snippets are lowercase. The snippets were labeled automatically, but before labeling the authors assumed the snippets from the Rotten Tomatoes web page. The reviews marked with fresh are positive, and those reviews marked with rotten are negative.

CBOW model building

The workflow of this example using CBOW goes as follows: first, we will download and pre-process the movie review data. From this dataset, we will compute/fit the CBOW model of the Word2Vec algorithm. Then we will reuse the same model for sentiment prediction:

1. Load the required modules and libraries:

```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import random
import os
import pickle
import requests
import collections
import io
import tarfile
```

```
import urllib.request
import preprocessor
from nltk.corpus import stopwords
```



In Python 2, you need to import string and the `import urllib.request` should be replaced by `from urllib2 import urlopen`.

2. Create a directory for model saving:

```
data_folder_name = 'temp'
if not os.path.exists(data_folder_name):
    os.makedirs(data_folder_name)
```

3. Start a TensorFlow graph session:

```
sess = tf.Session()
```

4. Set the model parameters:

```
batch_size = 500 # CBOW looks 500 pairs of word embeddings
embedding_size = 200 # each word is a vector of length 200
vocabulary_size = 15000 # 15,000 frequent words
generations = 100000 # number of iteration
learning_rate = 0.001 # model learning rate
num_sampled = int(batch_size/2) # Number of negative examples
window_size = 4 # How many words to consider left and right as
suggested in [1]
```

5. Checkpoints for training:

```
save_embeddings_every = 5000
print_valid_every = 5000
print_loss_every = 100
```

6. Declare stop words.

   We are considering only `english` words:

```
stops = stopwords.words('english')
```

7. Picking some test words.

   We are expecting synonyms to appear against the following listed words that are adopted from `http://member.tokoha-u.ac.jp/~dixonfdm/Writing%20Topics%20htm/Movie%20Review%20Folder/movie_descrip_vocab.htm`.

Later on, we will have to transform these into indices:

```
valid_words = ['love', 'romance', 'sad', 'average', 'boring',
'entertaining', 'exciting', 'excellent', 'fantastic', 'action',
'thriller', 'horror', 'adventure', 'suspenseful']
```

8. Load movie review data:

```
print('Loading Data … ')
texts, target = preprocessor.load_movie_data()
```

Here, I have adopted a script for downloading and pre-processing the review data. The script is adopted, extended, and improved from Nick M., *TensorFlow Machine Learning Cookbook*, Packt Publishing Ltd. The `load_movie_data()` function goes as follows:

```
def load_movie_data():
    save_data_path = '/temp'
    movie_review_data_url = 'http://www.cs.cornell.edu/people/
pabo/movie-review-data/rt-polaritydata.tar.gz'
    req = requests.get(movie_review_data_url, stream=True)
    with open('temp_movie_review_temp.tar.gz', 'wb') as f:
        for chunk in req.iter_content(chunk_size=1024):
            if chunk:
                f.write(chunk)
                f.flush()
    tar = tarfile.open('temp_movie_review_temp.tar.gz', "r:gz")
    tar.extractall(path=save_data_path)
    tar.close()
    pos_file = os.path.join(save_data_path, 'rt-polaritydata',
'rt-polarity.pos')
    neg_file = os.path.join(save_data_path, 'rt-polaritydata',
'rt-polarity.neg')
    pos_data = []
    with open(pos_file, 'r', encoding='latin-1') as f:
        for line in f:
            pos_data.append(line.encode('ascii', errors='ignore').
decode())
        f.close()
        pos_data = [x.rstrip() for x in pos_data]
    neg_data = []
    with open(neg_file, 'r', encoding='latin-1') as f:
        for line in f:
            neg_data.append(line.encode('ascii', errors='ignore').
decode())
    f.close()
```

```
    neg_data = [x.rstrip() for x in neg_data]
    texts = pos_data + neg_data
    target = [1] * len(pos_data) + [0] * len(neg_data)
    return (texts, target)
```

In this function, I first saved the data file in a temporary location and then negative and positive review files are processed separately.



In Python 2, "io.open" is needed for the above function.

Once I have extracted them separately, I further concatenated them to produce a single text file containing only reviews. Later on, I computed the labels in terms of if review text was negative or positive.

9. Normalize the review text.

   Although, as I mentioned, the review text is already lowercase, it might contain numbers, special characters, and importantly, stop words. Thus we should remove them and generate a clean one. Let's do it using the `normalize_text()` function:

```
print('Normalizing Text Data …')
texts = preprocessor.normalize_text(texts, stops)
```

   Again the `normalize_text()` is a helper function that does these tricks. It goes as follows:

```
def normalize_text(texts, stops):
    # convert to lower case
    texts = [x.lower() for x in texts]
    # Remove punctuation
    texts = [''.join(c for c in x if c not in string.punctuation)
for x in texts]
    # Remove numbers
    texts = [''.join(c for c in x if c not in '0123456789') for x
in texts]
    # Remove stopwords
    texts = [' '.join([word for word in x.split() if word not in
(stops)]) for x in texts]
    # Trim extra whitespace left by the stopWords function
    texts = [' '.join(x.split()) for x in texts]
    return (texts)
```

Now that we have the cleaned text, however, each review (that is, text) has to have at least four words. Now let's ensure this in both the feature and target as follows:

```
target = [target[ix] for ix, x in enumerate(texts) if len(x.
split()) > 3]
texts = [x for x in texts if len(x.split()) > 3]
```

10. Creating the data dictionaries:

```
print('Creating Dictionary … ')
word_dictionary = preprocessor.build_dictionary(texts, vocabulary_
size)
word_dictionary_rev = dict(zip(word_dictionary.values(), word_
dictionary.keys()))
text_data = preprocessor.text_to_numbers(texts, word_dictionary)
```

Now we need to filter out sentences that aren't long enough, that is, have less than four words at least:

```
text_data = [x for x in text_data if len(x)>=(2*window_size+1)]
```

In the preceding code, we have used two user-defined functions: `build_dictionary()` builds a dictionary of words. At first, it turns sentences (lists of strings) into lists of words, then it initializes a list of [`word`, `word_count`] for each word, starting with unknown words. Then it adds most frequent words, limited to the vocabulary size.

Finally, it creates a dictionary for each word by adding the word itself and creating the value of the prior dictionary length:

```
def build_dictionary(sentences, vocabulary_size):
    split_sentences = [s.split() for s in sentences]
    words = [x for sublist in split_sentences for x in sublist]
# Initialize list of [word, word_count] for each word, starting
with unknown
    count = [['RARE', -1]]
    count.extend(collections.Counter(words).most_
common(vocabulary_size - 1))
    word_dict = {}
    for word, word_count in count:
        word_dict[word] = len(word_dict)
    return (word_dict)
```

Again, `text_to_numbers()` is a helper function that turns the text data into lists of integers from the dictionary and goes as follows:

```
def text_to_numbers(sentences, word_dict):
    # Initialize the returned data
    data = []
    for sentence in sentences:
        sentence_data = []
        # Use selected index or rare word index
        for word in sentence.split(' '):
            if word in word_dict:
                word_ix = word_dict[word]
            else:
                word_ix = 0
            sentence_data.append(word_ix)
        data.append(sentence_data)
    return (data)
```

The next task is to get the validation word keys defined from step 7:

```
valid_examples = [word_dictionary[x] for x in valid_words]
```

11. Creating the CBOW model.

    First, we need to define the word `embeddings` as follows:

    ```
    embeddings = tf.Variable(tf.random_uniform([vocabulary_size,
    embedding_size], -1.0, 1.0))
    ```

    Now that we have the word embedding for a set of continuous words, it's time to define the **Noise Contrastive Estimation** (**NCE**) loss parameters:

    ```
    nce_weights = tf.Variable(tf.truncated_normal([vocabulary_size,
    embedding_size], stddev=1.0 / np.sqrt(embedding_size)))
    nce_biases = tf.Variable(tf.zeros([vocabulary_size]))
    ```

    Before proceeding to the next code segment, a quick discussion on the NCE function is mandatory. TensorFlow has a built-in NCE loss function to handle categorical output that is too sparse for the softmax to converge. Now the question is that what is the input and output matrices in the NCE function?

    Take for example the skip-gram model for this sentence: "The quick brown fox jumped over the lazy dog."

The input and output pairs are: (quick, the), (quick, brown), (brown, quick), (brown, fox), and more combinations. Now, for this type of setting, what is the final embedding? We can extract the final embedding using {*w*} between the input and hidden layer , which can be conceptualized as follows:

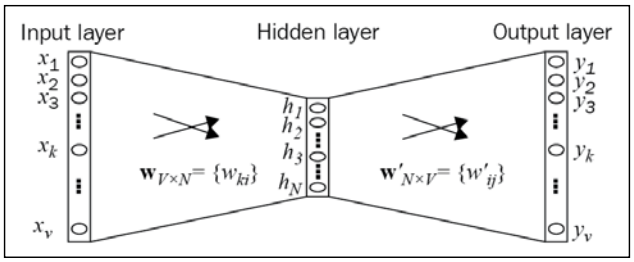


Figure 21: Using NCE function for handling categorical output in a dataset

To illustrate this idea more intuitively, refer to the following figure that extracts one hot vector; [*0, 0, 0, 1, 0*] is the input layer in the preceding graph, the output is the word embedding [*10, 12, 19*], and *W* (in the preceding graph) is the matrix in between:

$$[0\ \ 0\ \ 0\ \ 1\ \ 0] \times \begin{bmatrix} 17 & 24 & 1 \\ 23 & 5 & 7 \\ 4 & 6 & 13 \\ 10 & 12 & 19 \\ 11 & 18 & 25 \end{bmatrix} = [10\ \ 12\ \ 19]$$

Figure 22: Using the NCE function to extract one hot vector for handling categorical output in a dataset

For more detailed discussion, please refer to the TensorFlow page at `https://www.tensorflow.org/tutorials/word2vec`.

Well, we have created the word embeddings and the NCE loss function; now we need to create a space for TensorFlow for the data/target placeholders as follows:

```
x_inputs = tf.placeholder(tf.int32, shape=[batch_size, 2*window_
size])
y_target = tf.placeholder(tf.int32, shape=[batch_size, 1])
valid_dataset = tf.constant(valid_examples, dtype=tf.int32)
```

Now that our placeholders are ready to take the value and pass it back through the TensorFlow tensors, before doing so we need to create a lookup for the word embedding by adding together the window embeddings:

```
embed = tf.zeros([batch_size, embedding_size])
for element in range(2*window_size):
    embed += tf.nn.embedding_lookup(embeddings, x_inputs[:,
element])
```

We need to define the loss function for the prediction. Since we already have the weights and biases from the NCE function, we have the target, so we provide the lookup as the input in the loss function. We also need to define the number of samples and number of classes. In this case, number of classes to be predicted is the vocabulary size that we defined in step 4:

```
loss = tf.reduce_mean(tf.nn.nce_loss(weights=nce_weights,
                                     biases=nce_biases,
                                     labels=y_target,
                                     inputs=embed,
                                     num_sampled=num_sampled,
num_classes=vocabulary_size))
```

We have the objective loss function; we now need to define an optimizer that will reduce the loss in training:

```
optimizer = tf.train.GradientDescentOptimizer(learning_
rate=learning_rate).minimize(loss)
```

Here we have used the Gradient Descent optimizer, but you can use Adam or any other optimizer listed previously. Now before we create the model saving operation, we place the nearest words to our validation word dataset to get an idea of how our `embeddings` are working using the cosine similarity betweenness:

```
norm = tf.sqrt(tf.reduce_sum(tf.square(embeddings), 1, keep_
dims=True))
normalized_embeddings = embeddings / norm
valid_embeddings = tf.nn.embedding_lookup(normalized_embeddings,
valid_dataset)
similarity = tf.matmul(valid_embeddings, normalized_embeddings,
transpose_b=True)
```

Now it's time to save the word `embeddings` since we want to reuse this model later on. The built-in TensorFlow `train.Saver()` method saves the whole graph, but we can give it a positional argument asking it to save the embedding variables only:

```
saver = tf.train.Saver({"embeddings": embeddings})
```

So far we have defined and used so many variables. However, you need to formally initialize them using the global variable initializer method of TensorFlow:

```
init_op = tf.global_variables_initializer()
sess.run(init_op)
```

12. Training the CBOW model.

    Our word embedding is already ready. However, we haven't enough input and target to train the classifier, that is, model. Even before that, we need to get the batch data generated by the CBOW model so that we can feed it through TensorFlow placeholders and `feed_dict`. Later on, we will run the session graph to reduce the training loss. Once the training is finished, we will save the trained model so that it can be reused later on:

```
print('Starting Training … ')
loss_vec = []
loss_x_vec = []
for i in range(iterations):
    batch_inputs, batch_labels = preprocessor.generate_batch_
data(text_data, batch_size, window_size, method='cbow')
    feed_dict = {x_inputs : batch_inputs, y_target : batch_labels}
    sess.run(optimizer, feed_dict=feed_dict)
    if (i+1) % save_embeddings_every == 0:
        with open(os.path.join(data_folder_name,'movie_vocab.
pkl'), 'wb') as f:
            pickle.dump(word_dictionary, f)
        model_checkpoint_path = os.path.join(os.getcwd(),data_
folder_name,'cbow_movie_embeddings.ckpt')
        save_path = saver.save(sess, model_checkpoint_path)
        print('Model saved in file: {}'.format(save_path))
```

    If everything went fine, the CBOW model should be saved to your desired location. Now it's time to reuse and evaluate the model against the same dataset.

Reusing the CBOW for predicting sentiment

To do this, we will create a separate test and training set. The training set will be used for training the model and the test set will be used to evaluate the model. Then we will load the already saved CBOW embeddings and then perform LR on the average embedding of a review. Since we will be loading the model variables from the local file system onto already initialized variables in our current model, we have to make sure to store and load our vocabulary dictionary with the same mapping from words to embedding indices when using the same embedding.

The steps are more or less the same until we prepare the dataset as the training and test set. So you can simply reuse steps 1 to 9 from the model building section above. However we need to import pickle so that we can restore already saved word embedding and models:

```
import pickle
```

Also for restoring the model, we will use the TensorFlow built-in function `train.Saver()`. Now, once you have completed step 9, then the next task is to split (75% for the training and 25% for the test) the dataset into training and test sets as follows:

```
train_indices = np.random.choice(len(target), round(0.75*len(target)),
replace=False)
test_indices = np.array(list(set(range(len(target))) - set(train_
indices)))
texts_train = [x for ix, x in enumerate(texts) if ix in train_indices]
texts_test = [x for ix, x in enumerate(texts) if ix in test_indices]
target_train = np.array([x for ix, x in enumerate(target) if ix in
train_indices])
target_test = np.array([x for ix, x in enumerate(target) if ix in
test_indices])
```

We already have a dictionary and the embedding matrix saved in the `temp` folder (or in your desired location), so we can simply reuse them as follows:

```
dict_file = '/temp/movie_vocab.pkl'
word_dictionary = pickle.load(open(dict_file, 'rb'))
```

We have our word dictionary; we further need to convert the test into a list of indices up to a specific length as follows:

```
text_data_train = np.array(preprocessor.text_to_numbers(texts_train,
word_dictionary))
text_data_test = np.array(preprocessor.text_to_numbers(texts_test,
word_dictionary))

text_data_train = np.array([x[0:max_words] for x in [y+[0]*max_words
for y in text_data_train]])
text_data_test = np.array([x[0:max_words] for x in [y+[0]*max_words
for y in text_data_test]])
```

Let's start creating the model before we train it. But first, we need to create the word `embeddings`:

```
print('Creating Model... ')

embeddings = tf.Variable(tf.random_uniform([vocabulary_size,
embedding_size], -1.0, 1.0))
```

Since we will be using the logistic regression classifier, first we need to create variables for the LR model:

```
A = tf.Variable(tf.random_normal(shape=[embedding_size,1]))
b = tf.Variable(tf.random_normal(shape=[1,1]))
```

Now, to use the preceding variables, we will have to initialize the placeholders for the TensorFlow tensors passed in between:

```
x_data = tf.placeholder(shape=[None, max_words], dtype=tf.int32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
```

Then we will have to create a `lookup`, embedding vectors by averaging all word `embeddings` in the documents:

```
embed = tf.nn.embedding_lookup(embeddings, x_data)
embed_avg = tf.reduce_mean(embed, 1)
```

Now let's declare the sigmoid in the loss function:

```
model_output = tf.add(tf.matmul(embed_avg, A), b)
```

Then we declare our loss function, that is, cross entropy loss:

```
loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_
logits(logits=model_output, labels=y_target))
```

We need to hold the value to evaluate the model so that we can have a look at the actual prediction later on:

```
prediction = tf.round(tf.sigmoid(model_output))
predictions_correct = tf.cast(tf.equal(prediction, y_target),
tf.float32)
accuracy = tf.reduce_mean(predictions_correct)
```

We have the loss function defined above, so our next task is to declare an optimizer. Let's use the Gradient Descent optimizer with learning rate of `0.001`:

```
training_op = tf.train.GradientDescentOptimizer(0.001)
train_step = training_op.minimize(loss)
```

Now we need to initialize all the TensorFlow variables using the `global_variables_initializer()` function as follows:

```
init_op = tf.global_variables_initializer()
sess.run(init_op)
```

Finally, before training the model, we load and restore the `embeddings` already saved:

```
model_checkpoint_path = '/temp/cbow_movie_embeddings.ckpt'
saver = tf.train.Saver({"embeddings": embeddings})
saver.restore(sess, model_checkpoint_path)
```

Now that we have loaded and restored the embedding model, it's time to train the LR model using the already trained CBOW model:

```
print('Starting Model Training... ')
train_loss = []
test_loss = []
train_acc = []
test_acc = []
i_data = []
for i in range(10000):
    rand_index = np.random.choice(text_data_train.shape[0],
size=batch_size)
    rand_x = text_data_train[rand_index]
    rand_y = np.transpose([target_train[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})
    # We only record loss and accuracy every 100 generations
    if (i+1)%100==0:
        i_data.append(i+1)
        train_loss_temp = sess.run(loss, feed_dict={x_data: rand_x,
y_target: rand_y})
        train_loss.append(train_loss_temp)
        test_loss_temp = sess.run(loss, feed_dict={x_data: text_data_
test, y_target: np.transpose([target_test])})
        test_loss.append(test_loss_temp)
        train_acc_temp = sess.run(accuracy, feed_dict={x_data: rand_x,
y_target: rand_y})
        train_acc.append(train_acc_temp)
        test_acc_temp = sess.run(accuracy, feed_dict={x_data: text_
data_test, y_target: np.transpose([target_test])})
        test_acc.append(test_acc_temp)
    if (i+1)%500==0:
        acc_and_loss = [i+1, train_loss_temp, test_loss_temp, train_
acc_temp, test_acc_temp]
        acc_and_loss = [np.round(x,2) for x in acc_and_loss]
        print('Generation # {}. Train Loss (Test Loss): {:.2f}
({:.2f}). Train Acc (Test Acc): {:.2f} ({:.2f})'.format(*acc_and_
loss))
```

Now let's compute and print the accuracy:

```
print('\nOverall accuracy on test set (%): {}'.format(np.mean(test_
acc)*100.0))
>>>
Loading Data...
Normalizing Text Data...
Creating Model...
Starting Model Training...
Iteration # 500. Train Loss (Test Loss): 0.70 (0.70). Train Acc (Test
Acc): 0.51 (0.50)
…
Iteration # 9500. Train Loss (Test Loss): 0.71 (0.70). Train Acc (Test
Acc): 0.46 (0.50)
Iteration # 10000. Train Loss (Test Loss): 0.70 (0.70). Train Acc
(Test Acc): 0.53 (0.50)

Overall accuracy on test set (%): 50.57876205444336
```

We can see how well the training and prediction went by plotting the loss over time:

```
plt.plot(i_data, train_loss, 'k-', label='Training loss')
plt.plot(i_data, test_loss, 'r--', label='Test loss', linewidth=4)
plt.title('Cross entropy loss per iteration')
plt.xlabel('Iteration')
plt.ylabel('Cross entropy loss')
plt.legend(loc='upper right')
plt.show()
>>>
```

The output is as follows:

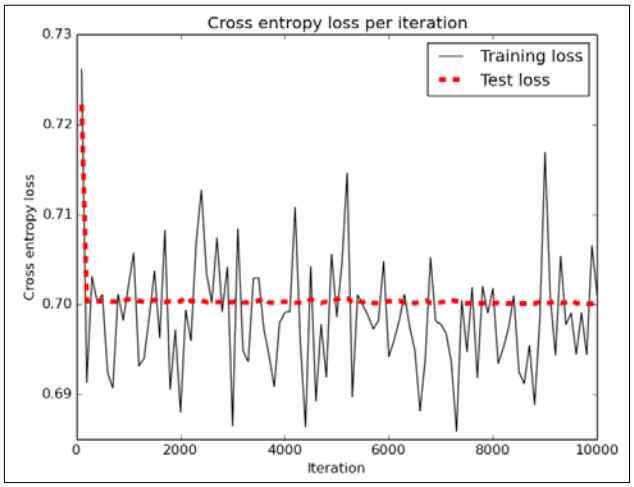


Figure 23: Cross entropy loss per iteration using LR model that is optimized by the Gradient Descent optimizer

Finally, let's plot the train and test accuracy:

```
plt.plot(i_data, train_acc, 'k-', label='Accuracy on the training
set')
plt.plot(i_data, test_acc, 'r--', label='Accuracy on the test set',
linewidth=4)
plt.title('Accuracy on the train and test set')
plt.xlabel('Iteration')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.show()
>>>
```

The output is as follows:

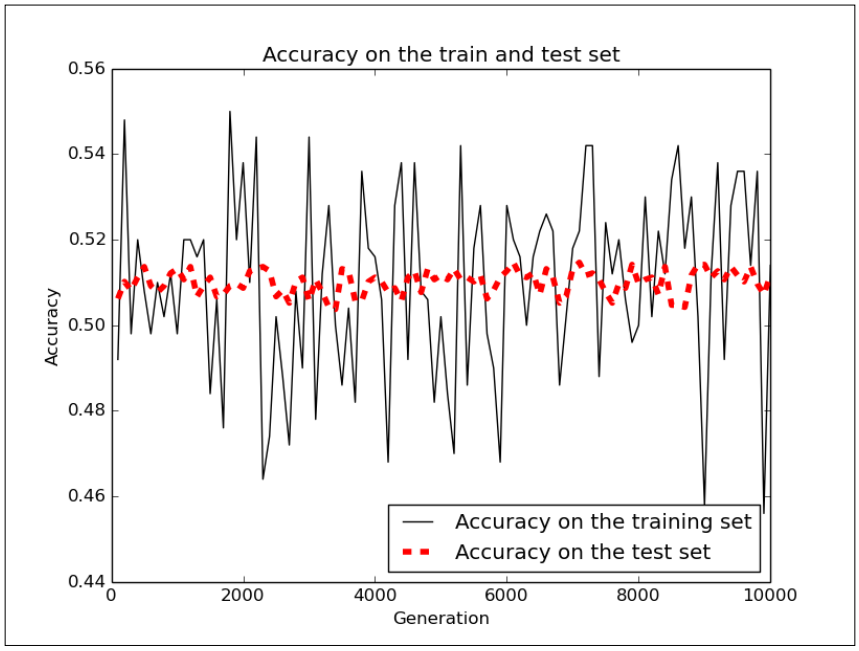


Figure 24: Accuracy on the training and test set using LR model optimized by Gradient Descent optimizer

Summary

In this chapter, we discussed the wonderful field of text analytics using TensorFlow. Text analytics is a wide area in NLP and ML and is useful in many use cases such as sentiment analysis, chat bots, email spam detection, text mining, natural language processing, and more. We learned how to use TensorFlow for text analytics with a focus on use cases of text classification from the unstructured spam prediction and movie review datasets. Based on the spam filtering dataset, we will develop predictive models using the LR algorithm with TensorFlow.

Particularly, we used BOW and TF-IDF algorithms for spam prediction with more than 90% prediction accuracy. Later in the chapter, we saw how to develop predictive models for predicting sentiment from the movie review dataset using CBOW and continuous skip-gram algorithms, jointly referred to as the Word2vec method. Using Word2vec, we attained more than 50% accuracy. It is not outstanding. However, sentiment analysis is a really hard task to do because the human language makes it very hard to grasp the subtleties and nuances or the true meaning. Therefore, 50% accuracy is not that trivial, but can still be improved using other algorithms.

In this chapter, we have seen how to build a simple classification model so any new tweet can be labeled based on the training set. Furthermore, LR is not the only model that can be used, but other algorithms such as decision trees, random forests, linear SVM, gradient boosted trees, or multilayer preceptors can be used too.

In *Chapter 7*, *Using Deep Neural Networks for Predictive Analytics*, we will move on to deep learning for advanced predictive analytics. In particular, we will provide two practical examples of using **deep belief networks** (**DBN**) and **multilayer perceptron** (**MLP**) on a bank marketing dataset.

7
Using Deep Neural Networks for Predictive Analytics

Artificial Neural Networks (**ANNs**) are at the very core of **deep learning** (**DL**). They are versatile, powerful, and scalable, making them ideal to tackle large and highly complex **Machine Learning** (**ML**) tasks, such as classifying billions of images, powering speech recognition services, and recommending hundreds of millions of users to watch the best videos every day by stacking multiple ANNs together. These multiple stacked ANNs are called **Deep Neural Networks** (**DNNs**). Using these, we can build very robust and accurate predictive models for predictive analytics.

In this chapter, we will see two examples of how to build very robust and accurate predictive models for predictive analytics using **deep belief networks** (**DBN**) and **multilayer perceptron** (**MLP**) on a bank marketing dataset. The following topics will be covered in this chapter:

- Deep learning in the big data era
- Artificial Neural Networks
- Deep Neural Networks
- DNN Architectures
- Multilayer perceptron
- Using multilayer perceptron for predictive analytics
- Deep Belief Neural Networks
- Using deep belief networks for predictive analytics

Deep learning for better predictive analytics

Simple machine learning methods that were used in normal sized data analysis won't be effective anymore and should be substituted by DNN learning methods. Although classical machine learning techniques allow researchers to identify groups or clusters of related variables, the accuracy and effectiveness of these methods diminishes for large and high–dimensional datasets, such as whole human genomes.

On the other hand, deep learning can make better representations of large–scale datasets to build models to learn these representations very extensively. DL is a branch of ML based on a set of algorithms that attempt to model high-level abstractions in data. In this regard, Ian Goodfellow et al. defined DL as follows:

> *"Deep learning is a particular kind of machine learning that achieves great power and flexibility by learning to represent the world as a nested hierarchy of concepts, with each concept defined in relation to simpler concepts, and more abstract representations computed in terms of less abstract ones."*



Machine learning versus deep learning

- **Data dependencies**: Deep learning algorithms don't perform that well with small amount data.
- **Hardware dependencies**: Deep learning algorithms heavily depend on high-end machines with GPU, especially when lots of scientific computing and precision is important.
- **Feature engineering**: Deep learning algorithms learn high-level features from data rather than from handcrafted features.
- **Problem solving approach**: Deep learning advocates to solve the problem end-to-end.
- **Execution time**: A deep learning algorithm with so many parameters and features takes a long time to train.

Let's take an example. Suppose we want to develop a predictive analytics model, say an animal recognizer, where our system has to: i) Recognize whether a given image is a cat or a dog, that is classification, ii) Cluster dog and cat photos, that is clustering. If we solve the first problem using typical machine learning problem, we will define features such as if the animal has whiskers or his ears, and so on. In other words, we will define the facial features and will write a method for identifying which features are more important in classifying a particular animal.

For the second problem, it is not possible for a classical machine learning algorithm such as K-means to classify images (since K-means is a linear model for clustering that cannot represent non-linear features). DL algorithms will take the two problems one step ahead. That is, using DL algorithm, most important features will be extracted automatically after determining which features are the most important for classification or clustering. In contrast, using a classical machine learning algorithm, we had to manually provide the features. In summary, the workflow using deep learning will now change as follows:

- A DL algorithm will first identify the edges that are most relevant to finding (or clustering) cats or dogs
- It will them build on this hierarchically to find what combination of shapes and edges exists.
- After consecutive hierarchical identification of complex concepts and features, it then:

    1. Decides which of these features are responsible for finding the answer.
    2. Take out the label column and do the unsupervised training using autoen coder and does the clustering.

Artificial Neural Networks

Take a look at the brain's architecture for inspiration (figure 1), called **biological neurons**. It is an unusual–looking cell, mostly found in the animal cerebral brain, which consists of cortexes; the cortex itself is composed of a cell body containing the nucleus and most of the cell's complex components, many branching extensions called **dendrites**, and one very long extension called the **axon**.

Near its extremity, the axon splits off into many branches called **telodendrian**, and at the top of these branches are minuscule structures called synaptic terminals (or simply **synapses**), which is connected to the dendrites of other neurons. Biological neurons receive short electrical impulses called **signals** from other neurons, then fire their own signals:

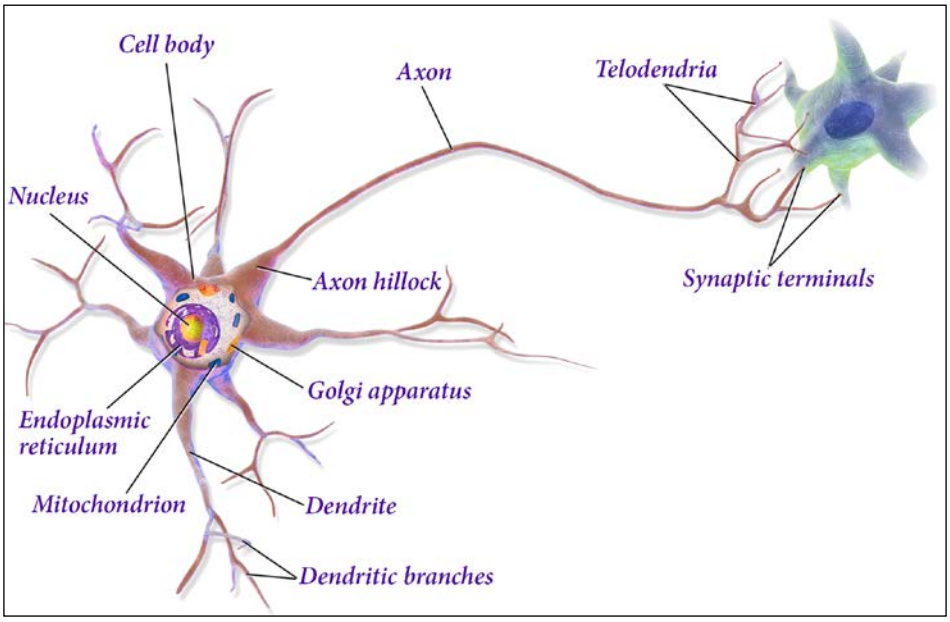


Figure 1: Biological neurons

Based on this whole idea of biological neurons, the term and idea of artificial neurons evolved, including how to build intelligent machines for deep learning based predictive analytics. This is the key idea that inspired Artificial Neural Networks.

Now that we know the ANNs, we need to know about **perceptron**. The perceptron is one of the simplest ANN architectures, called a **linear threshold unit** (**LTU)**, inspired by biological neurons. The thing is that the inputs and outputs are now numbers (instead of binary on/off values triggered by the human brain from the biological perspective) and each input connection is associated with a weight. The LTU computes a weighted sum of its inputs that can be defined by the following equation:

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = \mathbf{w}^T \cdot \mathbf{x})$$

This then applies a step function to that sum and outputs the result $hw(x) = step\ (z) = step\ (w^T.x)$:

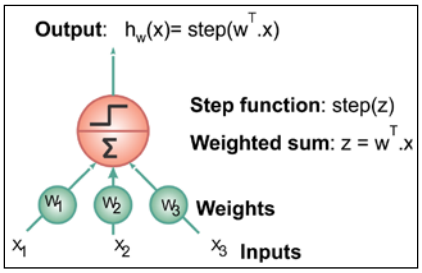


Figure 2: Linear threshold unit for the perceptron

Similar to logistic regression, or **Support Vector Machines** (**SVMs**), a single LTU can be used for simple linear and binary classification. It computes a linear combination of the inputs and if the result exceeds a threshold, it outputs the positive class or else outputs the negative class. However, training an LTU is not an easy job, but training such a simple model means finding the right values for $w0$, $w1$, and $w2$.

A perceptron is simply composed of a single layer of LTUs, with each neuron connected to all the inputs. These connections are often represented using special pass–through neurons called **input neurons**: they just output whatever input they are fed. Moreover, an extra bias feature is generally added ($x0 = 1$). This bias feature is typically represented using a special type of neuron called a **bias neuron**, which just outputs 1 all the time.

A perceptron with two inputs and three outputs is represented in figure 3. This perceptron can simultaneously classify instances into three different binary classes, which makes it a multioutput classifier:

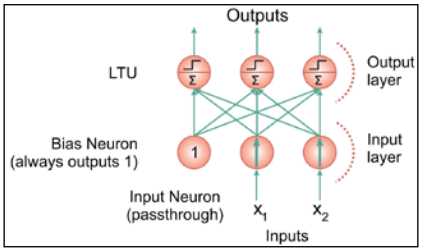


Figure 3: A perceptron with two inputs and three outputs

Since the decision boundary of each output neuron is linear, perceptrons are incapable of learning complex patterns. However, if the training instances are linearly separable, research has shown that this algorithm will converge to a solution called "perceptron **convergence theorem**."

Unfortunately, this theorem cannot make the simple neural network learn the complex relationship between the features and labels for robust and scalable predictive analytics but the Deep Neural Network can.

Deep Neural Networks

Deep learning is a term to differentiate an artificial neural network of multiple hidden layers from the simpler neural network that only consists of a small number of layers, as in figure 4. Each layer consists of a certain number of neurons shown as circles, which correspond to certain types of activation functions, such as identity or logistic functions.

The ANNs start working on some input data fed onto the input layer. Then some computation happens and results in the output layer. The output from certain layers will become the input of the next layer after being multiplied with certain weights. The connection between neurons is shown by the straight line in figure 4.

DNNs are neural networks, very similar to the ones we have discussed in the previous section, but consisting of a more complex model with a great number of neurons, hidden layers, and connections. In short, an ANN with two or more hidden layers is called a DNN:

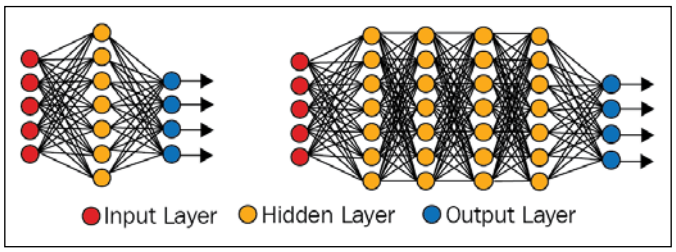


Figure 4: A simple neural network versus deep learning neural network

Now let's take a closer look at the different DNN architecture in the next section.

DNN architectures

The basic structure of DNNs consists of an input layer, multiple hidden layers, and an output layer (figure 4). Once the input data is given to the DNNs, output values are computed sequentially along the layers of the network. At each layer, the input vector comprising the output values of each unit in the layer below is multiplied by the weight vector for each unit in the current layer to produce the weighted sum. Then, a nonlinear function, such as a sigmoid, hyperbolic tangent, or **rectified linear unit** (**ReLU**), is applied to the weighted sum to compute the output values of the layer.

The computation in each layer transforms the representations in the layer below into slightly more abstract representations. Based on the types of layers used in DNNs and the corresponding learning method, DNNs can be classified as multilayer perceptrons, which are based on **feedforward neural network** (**FFNN**), **Staked Auto-Encoders** (**SAEs**), or deep belief networks:

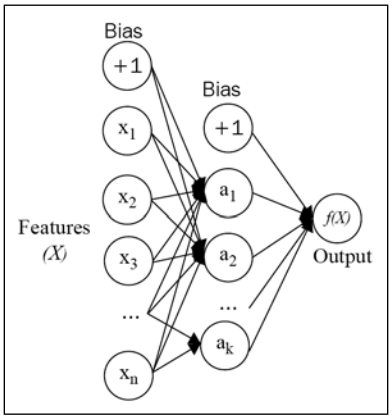


Figure 5: Left a simple one hidden layer MLP

An MLP has a similar structure to the usual neural networks but includes more stacked layers. It is trained in a purely supervised manner that uses only labeled data. Since the training method is a process of optimization in high–dimensional parameter space, DMLP is typically used when a large number of labeled data are available, as shown in figure 4.

SAE and DBN use AEs and RBMs as building blocks of the architectures, respectively. The main difference between these and MLPs is that training is executed in two phases: unsupervised pretraining and supervised fine-tuning:

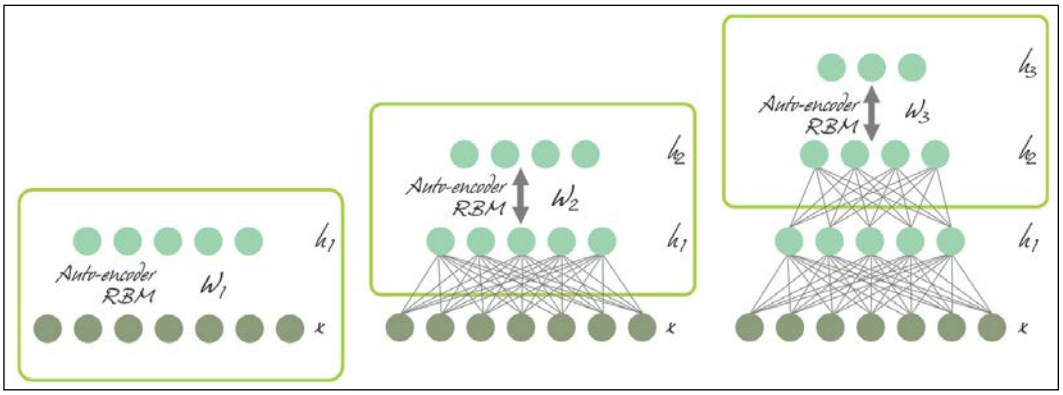


Figure 6: SAE and DBN using AE and RBM, respectively

First, in unsupervised pretraining (figure 6), the layers are stacked sequentially and trained, in a layer-wise manner as an AE or RBM using unlabeled data. Afterward, in supervised fine-tuning, an output classifier layer is stacked, and the whole neural network is optimized by retraining with labeled data.



For a brief cheat sheet on the different neural network architecture and their related publications, refer to the website of Asimov Institute at `http://www.asimovinstitute.org/neural-network-zoo/`.

However, in this chapter, we will not discuss SAEs, but will stick to MLPs and DBNs and use these two DNN architectures. We will see how to develop predictive models to deal with the high-dimensional dataset. Now, it's time to discuss MLPs more before we start using this architecture for predictive analytics.

Multilayer perceptrons

A multilayer perceptron is a feedforward neural network, which means that it is the only connection between neurons from different layers. More specifically, an MLP is composed of one (pass through) input layer, one or more layers of LTUs, called **hidden layers**, and one final layer of LTUs called the output layer. Every layer except the output layer includes a bias neuron and is connected to the next layer as a fully connected bipartite graph:

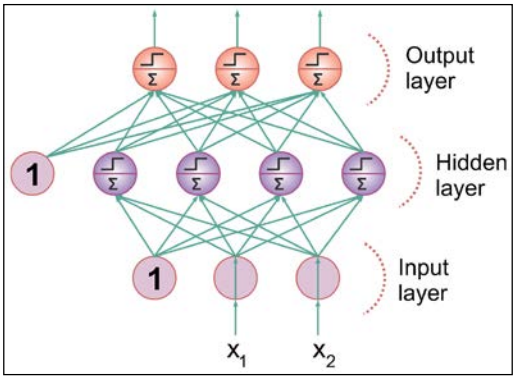


Figure 7: An MLP is composed of one input layer, one or more hidden layers, and an output layer

Training an MLP

An MLP was trained successfully using the back propagation training algorithm for the first time in 1986. However, nowadays the optimized version of this algorithm is called gradient descent (using reverse mode auto diff) and was discussed in *Chapter 3, From Data to Decisions – Getting Started with TensorFlow*. During the training phase, for each training instance, the algorithm feeds it to the network and computes the output of every neuron in each consecutive layer.

Then it measures the network's output error (that is, the difference between the desired output and the actual output of the network), and it computes how much each neuron in the last hidden layer contributed to each output neuron's error. It then proceeds to measure how much of these error contributions came from each neuron in the previously hidden layer, and so on, until the algorithm reaches the input layer. This reverse pass efficiently measures the error gradient across all the connection weights in the network by propagating the error gradient backward in the network.

More technically, the calculation of the gradient of the cost function for each layer is done by the back propagation method. The idea of gradient descent is to have a cost function that shows the difference between the predicted outputs of some neural network with the actual output:

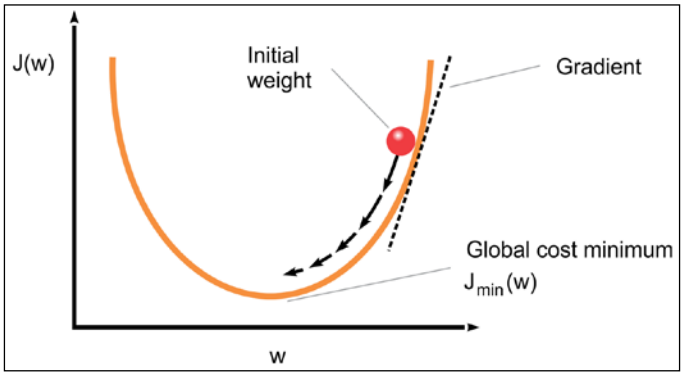


Figure 8: Sample implementation of artificial neural network for unsupervised learning

There are several known types of the cost function, such as the squared error function and the log-likelihood function, and the choice for this cost function can be based on many factors. The gradient descent method optimizes the network's weight by minimizing this cost function, and the steps are as follows:

- Weight initialization
- Calculation of neural networks predicted output, which is usually called the **forwarding propagation step**

- Calculation of cost/loss function; some common cost/loss functions include the **log-likelihood function** and **squared error function**
- Calculation of the gradient of the cost/lost function; for most DNN architecture, the most common method is backpropagation
- Weight update based on the current weight and the gradient of the cost/loss function
- Iteration of steps 2-5 until the cost function reaches a certain threshold or after a certain amount of iteration

An illustration of gradient descent can be seen in figure 8. There, we see a graph that shows a neural network's cost function based on the network's weight. In the first iteration of gradient descent, we apply the cost function on some random initial weight. With each iteration, we update the weight in direction of the gradient, which corresponds to the arrows in figure 8. The weight update is repeated until a certain number of iterations or until the cost function reaches a certain threshold.

Using MLPs

An MLP is a supervised neural network that is commonly used for classification and regression problems, although its implementation in image and video data has been gradually replaced by Convolutional Neural Networks (see more in *Chapter 8*, *Using Convolutional Neural Networks for Predictive Analytics*). Using MLPs, both the binary and multiclass classification problems can be solved.

However, for multiclass classification task and training, the output layer is typically modified by replacing the individual activation functions by a shared softmax function. The output of each neuron corresponds to the estimated probability of the corresponding class. Note that the signal flows only from the input to output but in one direction, so this architecture is an example of a FFNN:

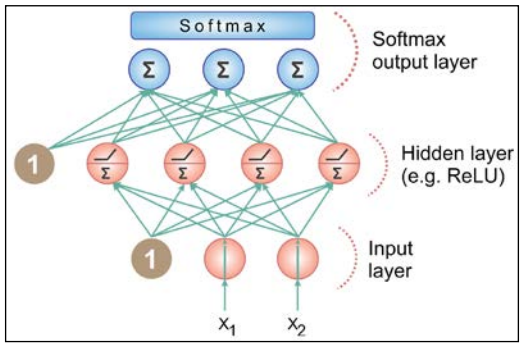


Figure 9: A modern MLP (including ReLU and Softmax) for classification

For example, if we use the labeled data as shown in figure 10, then the cost function is calculated between the actual label and the resulting label from the neural network:

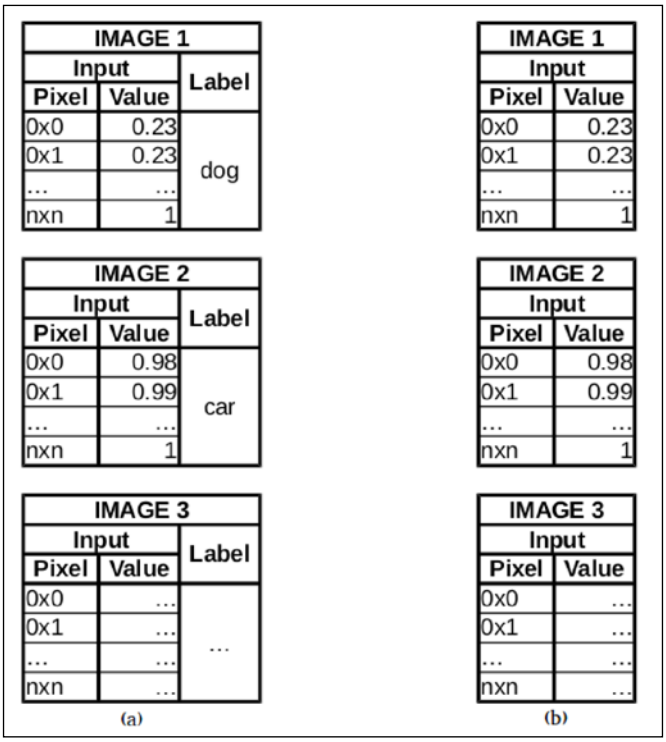


Figure 10: An example of image data representation

Here, data **(a)** is a labeled data, which classifies each image into a category. Data **(b)** is an unlabeled data which only gives the pixel values.

One important concept that is essential to know and try tuning is the hyperparameters in DNNs, which is not straightforward but needs extensive research. In the next section, we will see some pointers for this.

DNN performance analysis

The performance of our neural network is actually the performance of the application that was done by our neural network. So, the analysis of the performance of our neural network depends on the type of the application that was conducted; either it is classification, dimensionality reduction, or any other. This part will observe the analysis method for classification, regression, or clustering that will be discussed in this and the upcoming chapters.

For a classification problem, we will observe one illustration method that can also be used for performance analysis. Suppose we have a large-scale cancer genomics dataset and we would like to use a binary classification to predict cancer occurrence on two sets of subjects, which are the person who actually has cancer and the person without cancer.

The classification result from the neural network is not in binary form. Instead, it is in a continuous form where certain thresholds (or criterion values) determine whether cancer will occur or not, as demonstrated in the following figure of the classifications from these two sets of patients:

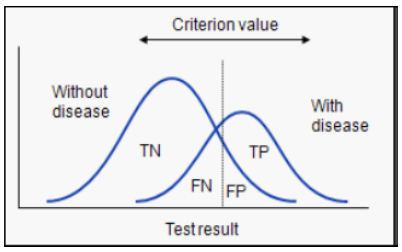


Figure 11: A curve explaining the cancer prediction from two sets of the patients

In figure 11, the left curve shows results from a set of people who don't have cancer and the right curve shows results from a set of people who actually have cancer. Classification results with a higher score than the criterion value are predicted to have cancer.

For every possible criterion value, there will be cases where the disease is correctly classified as positive (*TP = True Positive fraction*), while some cases of the disease might also be classified as negative (*FN = False Negative fraction*). On the other hand, some cases without the disease will be correctly classified as negative (*TN= True Negative fraction*), but some cases without the disease will be classified as positive (*FP = False Positive fraction*). The confusion matrix in Table 1 shows the classification result for certain criterion value:

		Prediction	
		Cancer = 1	**Cancer = 0**
Actual	**Cancer = 1**	True Positive (TP)	False Negative (FN)
	Cancer = 0	False Positive (FP)	True Negative (TN)

Table 1: Confusion matrix of cancer recurrence prediction in two sets of patients (patients who actually have cancer and patients who do not)

From the following confusion matrix, we can define the precision and recall; and, as stated earlier in *Chapter 4*, *Putting Data in Place: Supervised Learning for Predictive Analytics*, we can have another performance metrics called *F1* score, which is the harmonic mean of precision and recall. The formula for both accuracy and *F1* score can be seen next. Accuracy is the number of correct predictions out of all the predictions. To calculate *F1* score, we first need to observe two metrics, which are called precision and recall. If we create a prediction of cancer recurrence (instead of no cancer recurrence), then precision is the number of patients that are correctly predicted to have cancer out of all patients that are predicted to have cancer.

Meanwhile, recall is the number of patients that are correctly predicted to have cancer out of all patients that actually have cancer. The *F1* score is the harmonic mean of precision and recall, as is shown here:

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN}$$

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F1 = 2x\frac{Precisionx Recall}{Precision + Recall}$$

On the other hand, for the binary classification, an ROC curve from sample classification can be used. From figure 12, we can see that if the noncancerous curve (left curve) is completely separated from the cancerous curve (right curve), it means there is a criterion value that can create a perfect prediction for these particular datasets:

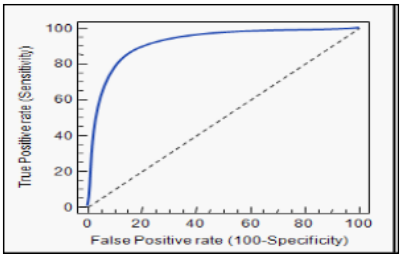


Figure 13: ROC explains the performance evaluation of binary classifier for predictive analytics

In this case, we will get a sharp ROC curve that reaches the top-left corner. The metric that is usually used to assess the performance of a classification is the **Area Under Curve** (**AUC**) of the ROC curve. The closer the AUC score is to 1 (perfect classification), the better the performance of our neural network is. One of the most common regression analysis metrics is the **Root Mean Squared Error** (**RMSE**) score and the coefficient of determination *R*2 score. RMSE is described as follows:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}\left(\hat{y}_i - y_i\right)^2}{n}}$$

$$R^2 = 1 - \frac{\sum_{i=1}^{n}\left(y_i - \hat{y}_i\right)^2}{\sum_{i=1}^{n}\left(y_i - \bar{y}\right)^2}$$

Here, unit vector *y*i as the predicted output for data *i*, *y*i as the actual output of data *i*, *y* is the mean over the actual output, and n as the number of the data. The smaller the RMSE and *R*2 scores, smaller will be the error of the prediction in comparison to the actual data. The main difference between RMSE and *R*2 is that *R*2 is scaled between *0* and *1*, whereas RMSE is not scaled to any particular values but is a continuous measurement. *R*2 can be more easily interpreted, but we can interpret more about the amount of deviation in RMSE.

Finally, if the objective of predictive analytics is doing some clustering in an unsupervised way, we can use **within–cluster sums of squares** (**WCSS**), which will necessarily maximize the distance between clusters similar to standard K-means or Bisecting K-means. For tuning the K value, we can utilize the Elbow method too in terms of WCSS. Finally, to measure the intra-clustering quality, the **Adjusted Rand Index** (**ARI**) can be used using the following formula (source: Wikipedia):

$$ARI = \frac{\Sigma_{ij}\binom{n_{ij}}{2} - \left[\Sigma_i\binom{a_i}{2}\Sigma_j\binom{b_j}{2}\right]/\binom{n}{2}}{\frac{1}{2}\left[\Sigma_i\binom{a_i}{2} + \Sigma_j\binom{b_j}{2}\right] - \left[\Sigma_i\binom{a_i}{2}\Sigma_j\binom{b_j}{2}\right]/\binom{n}{2}}$$

Finally, within the context of cluster analysis, *Purity* is an external evaluation criterion of cluster quality. It is the percent of the total number of objects (data points) that were classified correctly, in the unit range:

$$Purity = \frac{1}{N}\sum_{i=1}^{k} max_j \mid c_i \cap t_j \mid$$

Here, N = *number of objects* (data points), k = *number of clusters*, ci is a cluster in C, and t_j is the classification that has the max count for cluster c_i.

Fine-tuning DNN hyperparameters

The flexibility of neural networks is also one of their main drawbacks: there are many hyperparameters to tweak. Even in a simple MLP you can change the number of layers, the number of neurons per layer, the type of activation function to use in each layer, the weight initialization logic, dropout keep probability, and so on.

How do you know what combination of hyperparameters is the best for your task? Of course, you can use grid search with cross-validation to find the right hyperparameters for linear machine learning models, but for the DNNs there are many hyperparameters to tune, and since training a neural network on a large dataset takes a lot of time, you will only be able to explore a tiny part of the hyperparameter space in a reasonable amount of time. Here are some insights that can be followed.

Number of hidden layers

So, for many problems, you can start with just one or two hidden layers, and it will work just fine using two hidden layers with the same total amount of neurons in roughly the same amount of training time.

For more complex problems, you can gradually ramp up the number of hidden layers until you start overfitting the training set. Very complex tasks, such as large image classification or speech recognition, typically require networks with dozens of layers and they need a large amount of training data.

Number of neurons per hidden layer

Obviously, the number of neurons in the input and output layers is determined by the type of input and output your task requires. For example, if your dataset has the shape of *28 x 28*, it should expect to have input neurons with size 784 and the output neurons should be equal to the number of classes to be predicted. We will see how it works in practice in the next example using MLP, where there will be four hidden layers with 256 neurons: that's just one hyperparameter to tune instead of one per layer. Just like for the number of layers, you can try increasing the number of neurons gradually until the network starts overfitting.

Activation functions

In most cases, you can use the ReLU activation function in the hidden layers. It is a bit faster to compute than other activation functions, and gradient descent does not get stuck on plateaus as much when compared to the logistic function or the hyperbolic tangent function that usually saturate at 1. For the output layer, the softmax activation function is generally a good choice for classification tasks. For regression tasks, you can simply use no activation function at all. Other activation functions include Sigmoid and Tanh.

Weight and biases initialization

As we will see in the next example, initializing weight and biases for the hidden layers is an important hyperparameter to be taken care of:

- **Do not do all zero initialization**: A reasonable-sounding idea might be to set all the initial weights to zero, but it does not work in practice, because if every neuron in the network computes the same output, there will be no source of asymmetry between neurons if their weights are initialized to be the same.
- **Small random numbers**: It is also possible to initialize the weights of the neurons to small numbers but not identically zero. Alternatively, it is also possible to use small numbers drawn from a uniform distribution.
- **Initializing the biases**: It is possible and common to initialize the biases to be zero since the asymmetry breaking is provided by the small random numbers in the weights. Setting the biases to a small constant value such as 0.01 for all biases ensures that all ReLU units can propagate some gradient. However, it neither performs well nor does consistent improvements. Therefore, sticking with zero is recommended.

Regularization

There are several ways of controlling the training of DNNs to prevent overfitting in the training phase, for example, L2/L1 regularization, max norm constraints, and dropouts:

- **L2 regularization**: This is probably the most common form of regularization. Using the gradient descent parameter update, L2 regularization signifies that every weight will be decayed linearly towards zero.
- **L1 regularization**: For each weight w we add the term $\lambda|w|$ to the objective. However, it is also possible to combine L1 and L2 regularization *to achieve* elastic net regularization.
- **Max-norm constraints**: Issued to enforce an absolute upper boundary on the magnitude of the weight vector for each hidden layer neuron. Projected gradient descent is then can be used further to enforce the constraint.
- **Dropout**: While working with DNNs, we need another placeholder for dropout, which is a hyperparameter to be tuned and the training time but not the test time. It is implemented by only keeping a neuron active with some probability say $p<1.0$, or setting it to zero otherwise. The idea is to use a single neural net at test time without dropout. The weights of this network are scaled-down versions of the trained weights. If a unit is retained with *dropout_keep_prob* < *1.0* during training, the outgoing weights of that unit are multiplied by p at test time (figure 17):

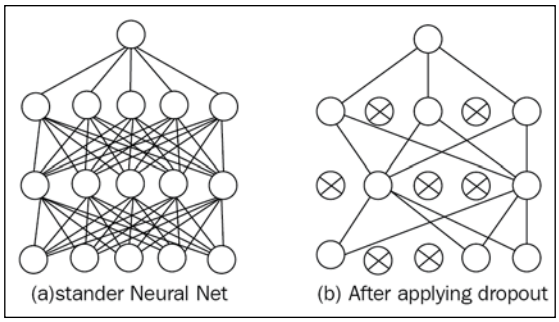


Figure 17: An example of dropout in DNNs

In the preceding figure, the figure to the left shows a standard neural net with two hidden layers. The figure to the right shows an example of a thinned net produced by applying dropout to the network on the left where all the crossed units have been dropped.

In summary, here's the overall workflow that we need to adopt while developing predictive analytics applications using DNNs:

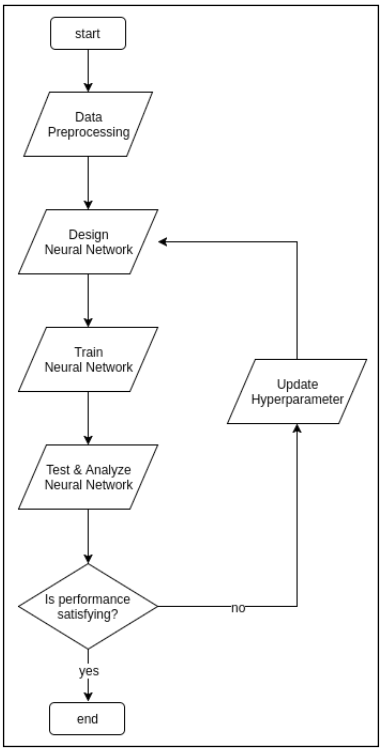


Figure 18: Pipeline of the implementation steps in developing predictive analytics applications using DNNs

Well, we have enough background theory and it's time to apply this for predictive analytics in practice. In the next section, we will show an example of using MLP for developing a predictive model.

Using multilayer perceptrons for predictive analytics

For this example, we will be using bank marketing datasets. The data is related to direct marketing campaigns of a Portuguese banking institution. The marketing campaigns were based on phone calls. Often, more than one contact to the same client was required, in order to assess whether the product (bank term deposit) would be ("yes") or would not be ("no") subscribed. The target is to use MLP to predict whether the client will subscribe a term deposit (variable y)—that is, a binary classification problem.

Dataset description

There are two sources that I would like to acknowledge. This dataset was used in a research paper published by Moro et al, *A Data-Driven Approach to Predict the Success of Bank Telemarketing*, Decision Support Systems, Elsevier, June 2014. Later on, it was donated to the UCI Machine Learning repository that can be downloaded from `https://archive.ics.uci.edu/ml/datasets/bank+marketing`. According to the dataset description, there are four datasets:

- `bank-additional-full.csv`: This includes all examples (41188) and 20 inputs, ordered by date (from May 2008 to November 2010), very close to the data analyzed in [Moro et al., 2014]
- `bank-additional.csv`: This includes 10% of the examples (4119), randomly selected from 1, and 20 inputs.
- `bank-full.csv`: This includes all the examples and 17 inputs, ordered by date (older version of this dataset with fewer inputs).
- `bank.csv`: This includes 10% of the examples and 17 inputs, randomly selected from 3 (the older version of this dataset with fewer inputs).

There are 21 attributes in the dataset. The independent variables, that is, features can be further categorized as bank client related data (attributes 1 to 7), related to the last contact with the current campaign (attributes 8 to 11), other attributes (attributes 12 to 15), and social and economic context attributes (attributes 16 to 20). The dependent variable is specified by *y*, the last attribute (21):

ID	Attribute	Explanation
1	`age`	Age in numbers.
2	`job`	This is the type of job in a categorical format with possible values: `admin`, `blue-collar`, `entrepreneur`, `housemaid`, `management`, `retired`, `self-employed`, `services`, `student`, `technician`, `unemployed`, and `unknown`.
3	`marital`	This is the marital status in a categorical format with possible values: `divorced`, `married`, `single`, and `unknown`, here, `divorced` means divorced or widowed.
4	`education`	This is the educational background in categorical format with possible values as follows: `basic.4y`, `basic.6y`, `basic.9y`, `high.school`, `illiterate`, `professional.course`, `university.degree`, `unknown`.
5	`default`	This is a categorical format with possible values in credit in default `no`, `yes`, `unknown`.
6	`housing`	Does the customer have a housing loan?

7	`loan`	The personal loan in a categorical format with possible values `no`, `yes`, and `unknown`.
8	`contact`	This is the contact communication type in a categorical format with possible values `cellular`, `telephone`.
9	`month`	This is the last contact month of the year in a categorical format with possible values `jan`, `feb`, `mar`, ..., `nov`, and `dec`.
10	`day_of_week`	This is the last contact day of the week in a categorical format with possible values `mon`, `tue`, `wed`, `thu`, and `fri`.
11	`duration`	This is the last contact duration, in seconds (numerical value). This attribute highly affects the output target (for example, if *duration=0*, then *y=no*). Yet, the duration is not known before a call is performed. Also, after the end of the call, *y* is obviously known. Thus, this input should only be included for benchmark purposes and should be discarded if the intention is to have a realistic predictive model.
12	`campaign`	This is the number of contacts performed during this campaign and for this client.
13	`pdays`	This is the number of days that passed by after the client was last contacted from a previous campaign (numeric; 999 means client was not previously contacted).
14	`previous`	This is the number of contacts performed before this campaign and for this client (numeric).
15	`poutcome`	The outcome of the previous marketing campaign (categorical: `failure`, `nonexistent`, and `success`).
16	`emp.var.rate`	This is the employment variation rate—quarterly indicator (numeric).
17	`cons.price.idx`	This is the consumer price index—monthly indicator (numeric).
18	`cons.conf.idx`	This is the consumer confidence index—monthly indicator (numeric).
19	`euribor3m`	This is the euribor 3 month rate—daily indicator (numeric).
20	`nr.employed`	This is the number of employees—quarterly indicator (numeric).
21	`y`	Signifies if the client subscribed a term deposit `eith` possible binary: `yes` and `no` values.

Table 1: Description of the bank marketing dataset

Preprocessing

Now you can see that the dataset is not ready to feed to your MLP or DBN classifier directly, since the feature a mix of numerical and categorical values. Also, the outcome variable is in a categorical value. Therefore, we need to convert them into numerical values so that the feature and the outcome variables are in numerical form. The next step shows the process. At first, we load the required packages and libraries needed for the preprocessing:

```
import pandas as pd
import numpy as np
from sklearn import preprocessing
```

Then, we load and parse the dataset:

```
data = pd.read_csv('bank-additional-full.csv', sep = ";")
```

Then, we extract variables names:

```
var_names = data.columns.tolist()
```

Now, based on the dataset description in table 1, we extract the categorical variables:

```
categs =
['job','marital','education','default','housing','loan','contact','mon
th','day_of_week','duration','poutcome','y']
```

Then, we extract the quantitative variables:

```
quantit = [i for i in var_names if i not in categs]
```

Now, let's get the dummy variables for categorical variables:

```
job = pd.get_dummies(data['job'])
marital = pd.get_dummies(data['marital'])
education = pd.get_dummies(data['education'])
default = pd.get_dummies(data['default'])
housing = pd.get_dummies(data['housing'])
loan = pd.get_dummies(data['loan'])
contact = pd.get_dummies(data['contact'])
month = pd.get_dummies(data['month'])
day = pd.get_dummies(data['day_of_week'])
duration = pd.get_dummies(data['duration'])
poutcome = pd.get_dummies(data['poutcome'])
```

Now, it's time to map variables to predict:

```
dict_map = dict()
y_map = {'yes':1,'no':0}
dict_map['y'] = y_map
data = data.replace(dict_map)
label = data['y']
df_numerical = data[quantit]
df_names = df_numerical .keys().tolist()
```

Once we have converted the categorical variables into numerical variables, the next task is to normalize the numerical variables, too. So, let's scale the quantitative variables:

```
min_max_scaler = preprocessing.MinMaxScaler()
x_scaled = min_max_scaler.fit_transform(df_numerical)
df_temp = pd.DataFrame(x_scaled)
df_temp.columns = df_names
```

Now that we have the temporary data frame for the (original) numerical variables, the next task is to combine all the data frames together and generate the normalized dataframe. We will use pandas for this:

```
normalized_df = pd.concat([df_temp,
                      job,
                      marital,
                      education,
                      default,
                      housing,
                      loan,
                      contact,
                      month,
                      day,
                      poutcome,
                      duration,
                      label], axis=1)
```

Finally, we need to save the resulting data frame as follows:

```
normalized_df.to_csv('bank_normalized.csv', index = False)
```

A TensorFlow implementation of MLP

For this example, we will be using the bank marketing dataset that we have normalized in the previous example. There are several steps to be followed:

1. Importing the required packages and modules.

   At first, we need to import TensorFlow and the other necessary packages and modules:

   ```
   import tensorflow as tf
   import pandas as pd
   import numpy as np
   import os
   from sklearn.cross_validation import train_test_split
   ```

2. Loading the normalized bank dataset.

   Now, we need to load the normalized bank marketing dataset, where all the features and the labels are numeric. For this we use the `read_csv()` method from the pandas library:

   ```
   FILE_PATH = 'bank_normalized.csv'                                    #
   Path to .csv dataset
   raw_data = pd.read_csv(FILE_PATH)                                    #
   Open raw .csv
   print("Raw data loaded successfully...\n")
   >>>
   Raw data loaded successfully...
   ```

3. Defining the required params for MLP.

   As mentioned in the previous section, tuning the hyperparameters for DNNs is not so straightforward a job. But it often depends on the dataset you are handling. For some of them, a possible workaround is setting these values based on the dataset-related statistics themselves, for example, a number of training instances, input size, and the number of classes. The reason is that DNNs are not suitable for very small and low-dimensional datasets. In this case, a better way is to use the linear models instead.

   At first, let's put a pointer to the label column itself and compute the number of instances and number of classes, and define the train/test split ratio as follows:

   ```
   Y_LABEL = 'y'                                                        #
   Name of the variable to be predicted
   KEYS = [i for i in raw_data.keys().tolist() if i != Y_LABEL]     #
   Name of predictors
   N_INSTANCES = raw_data.shape[0]                                     #
   Number of instances
   ```

```
N_INPUT = raw_data.shape[1] - 1                                  # Input
size
N_CLASSES = raw_data[Y_LABEL].unique().shape[0]                   #
Number of classes (output size)
TEST_SIZE = 0.25                                                 # Test
set size (% of dataset)
TRAIN_SIZE = int(N_INSTANCES * (1 - TEST_SIZE))                  # Train
size
```

Now, let's see the statistics of the dataset that we are going to use to train the MLP model:

```
print("Variables loaded successfully...\n")
print("Number of predictors \t%s" %(N_INPUT))
print("Number of classes \t%s" %(N_CLASSES))
print("Number of instances \t%s" %(N_INSTANCES))
print("\n")
>>>
Variables loaded successfully...
Number of predictors     1606
Number of classes          2
Number of instances      41188
```

Now, the next task is to define the other parameters, such as learning rate, training epochs, batch size, and the standard deviation for the weights. Usually, a low value of training rate will help you to learn your DNN more slowly but intensively. Note that we need to define more parameters, such as a number of hidden layers and activation function that will be defined later on:

```
LEARNING_RATE = 0.001
TRAINING_EPOCHS = 1000
BATCH_SIZE = 100
DISPLAY_STEP = 20
HIDDEN_SIZE = 256
ACTIVATION_FUNCTION_OUT = tf.nn.tanh
STDDEV = 0.1
RANDOM_STATE = 100
```

Note that preceding initialization is set on a trial-and-error basis. Therefore, depending on your use case and data types, set them wisely. Also, for the preceding code, `RANDOM_STATE` is used to signify a random state for the train and test split.

4. Prepare training and test sets.

   At first, we separate the raw features and the labels:

```
data = raw_data[KEYS].get_values()       # X data
labels = raw_data[Y_LABEL].get_values()  # y data
Now that we have the labels. However, labels have to be coded:
labels_ = np.zeros((N_INSTANCES, N_CLASSES))
labels_[np.arange(N_INSTANCES), labels] = 1
```

   Finally, we split the training and test sets. As mentioned in step 3, we keep 75% for training and the other 25% for the test set:

```
data_train, data_test, labels_train, labels_test = train_test_
split(data,labels_,test_size = TEST_SIZE,random_state = RANDOM_
STATE)
print("Data loaded and splitted successfully...\n")
>>>
Data loaded and splitted successfully...
```

5. Define TensorFlow placeholders.

   Since this is a supervised classification problem, we should have at least placeholders for the features and the labels:

```
X = tf.placeholder(tf.float32, [None, n_input])
y = tf.placeholder(tf.float32, [None, n_classes])
```

   Additionally, we need another placeholder for dropout, which is implemented by only keeping a neuron active, with some probability, say p < 1.0, or setting it to zero otherwise. Note that this is also a hyperparameter to be tuned and the training time, but not the test time:

```
dropout_keep_prob = tf.placeholder(tf.float32)
```

   Using the scaling given here enables the same network to be used for training (with `dropout_keep_prob < 1.0`) and evaluation (with `dropout_keep_prob == 1.0`).

6. Implementing an MLP with four hidden layers.

   As mentioned previously, an MLP is composed of one input layer, more than one hidden layers, and one final layer of LTUs called the output layer. For this example, I am going to show, to incorporate the training with four hidden layers. Thus, we are calling our classifier Deep Feedforward MLP.

Note that we also need to have the weight in each layer (except in the input layer) and the bias in each layer except the output layer. Usually, each hidden layer includes a bias neuron and is connected to the next layer as a fully connected bipartite graph, that is, feedforward from one hidden layer to another. Now, at first, let's define the size of the hidden layers:

```
n_input = N_INPUT                    # input n labels
n_hidden_1 = HIDDEN_SIZE             # 1st layer
n_hidden_2 = HIDDEN_SIZE             # 2nd layer
n_hidden_3 = HIDDEN_SIZE             # 3rd layer
n_hidden_4 = HIDDEN_SIZE             # 4th layer
n_classes = N_CLASSES                # output m classes
```

Now, we can define a method that implements the MLP classifier. For this, we are going to provide four parameters, such as input, weight, biases, and the dropout probability, as follows:

```
def MLPClassifier(_X, _weights, _biases, dropout_keep_prob):
    layer1 = tf.nn.dropout(tf.nn.tanh(tf.add(tf.matmul(_X, _
weights['h1']), _biases['b1'])), dropout_keep_prob)
    layer2 = tf.nn.dropout(tf.nn.tanh(tf.add(tf.matmul(layer1, _
weights['h2']), _biases['b2'])), dropout_keep_prob)
    layer3 = tf.nn.dropout(tf.nn.tanh(tf.add(tf.matmul(layer2, _
weights['h3']), _biases['b3'])), dropout_keep_prob)
    layer4 = tf.nn.dropout(tf.nn.tanh(tf.add(tf.matmul(layer3, _
weights['h4']), _biases['b4'])), dropout_keep_prob)
    out = ACTIVATION_FUNCTION_OUT(tf.add(tf.matmul(layer4, _
weights['out']), _biases['out']))
return out
```

The return value of the preceding method is the output of the activation function. Now, the preceding method is a stub implementation that did not tell anything concrete about the weights and biases. So before we start the training, we should have them defined:

```
weights = {
    'w1': tf.Variable(tf.random_normal([n_input, n_
hidden_1],stddev=STDDEV)),
    'w2': tf.Variable(tf.random_normal([n_hidden_1, n_
hidden_2],stddev=STDDEV)),
    'w3': tf.Variable(tf.random_normal([n_hidden_2, n_
hidden_3],stddev=STDDEV)),
    'w4': tf.Variable(tf.random_normal([n_hidden_3, n_
hidden_4],stddev=STDDEV)),
    'out': tf.Variable(tf.random_normal([n_hidden_4, n_
classes],stddev=STDDEV)),
}
```

```
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'b3': tf.Variable(tf.random_normal([n_hidden_3])),
    'b4': tf.Variable(tf.random_normal([n_hidden_4])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}
```

7. Building the MLP.

   Now, we can invoke the preceding implementation of the MLP with real arguments, that is, an input layer, weights, biases, and the dropout keep probability, as follows:

   ```
   pred = MLPClassifier(X, weights, biases, dropout_keep_prob)
   ```

8. Training the MLP model.

   Now, we have built the MLP model, and it's time to train the DNN. At first, we need to define the cost op, and then we will use Adam optimizer that will learn slowly and try to reduce the training loss as much as possible:

   ```
   cost_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_
   logits(logits=pred, labels=y))
   # softmax loss
   optimizer = tf.train.AdamOptimizer(learning_rate = LEARNING_RATE).
   minimize(cost_op)
   ```

   Now, we need to define additional parameters for computing the classification accuracy:

   ```
   correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
   accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
   print("MLP networks has been built successfully...")
   print("Starting training...")
   ```

   After that, we need to initialize all the variables and placeholders before launching a TensorFlow session:

   ```
   init_op = tf.global_variables_initializer()
   ```

   Now, we are very close to start the training. But before that, the last step is to create a TensorFlow session and launch it as follows:

   ```
   sess = tf.Session()
   sess.run(init_op)
   ```

Finally, we are ready to start training our MLP on the training set. We iterate over all the batches and fit using the batched data to compute the average training cost. Finally, we show the training cost and accuracy for each epoch:

```
for epoch in range(TRAINING_EPOCHS):
    avg_cost = 0.0
    total_batch = int(data_train.shape[0] / BATCH_SIZE)
    # Loop over all batches
    for i in range(total_batch):
        randidx = np.random.randint(int(TRAIN_SIZE), size = BATCH_
SIZE)
        batch_xs = data_train[randidx, :]
        batch_ys = labels_train[randidx, :]
        # Fit using batched data
        sess.run(optimizer, feed_dict={X: batch_xs, y: batch_ys,
dropout_keep_prob: 0.9})
        # Calculate average cost
        avg_cost += sess.run(cost, feed_dict={X: batch_xs, y:
batch_ys, dropout_keep_prob:1.})/total_batch
    # Display progress
    if epoch % DISPLAY_STEP == 0:
        print("Epoch: %3d/%3d cost: %.9f" % (epoch, TRAINING_
EPOCHS, avg_cost))
        train_acc = sess.run(accuracy, feed_dict={X: batch_xs, y:
batch_ys, dropout_keep_prob:1.})
        print("Training accuracy: %.3f" % (train_acc))
print("Your MLP model has been trained successfully.")
>>>
Starting training...
Epoch:   0/1000 cost: 0.356494816
Training accuracy: 0.920
…
Epoch: 180/1000 cost: 0.350044933
Training accuracy: 0.860
….
Epoch: 980/1000 cost: 0.358226758
Training accuracy: 0.910
```

Well done; our MLP model has been trained successfully. Now, what if we could see the cost and the accuracy graphically? It would be great I guess:

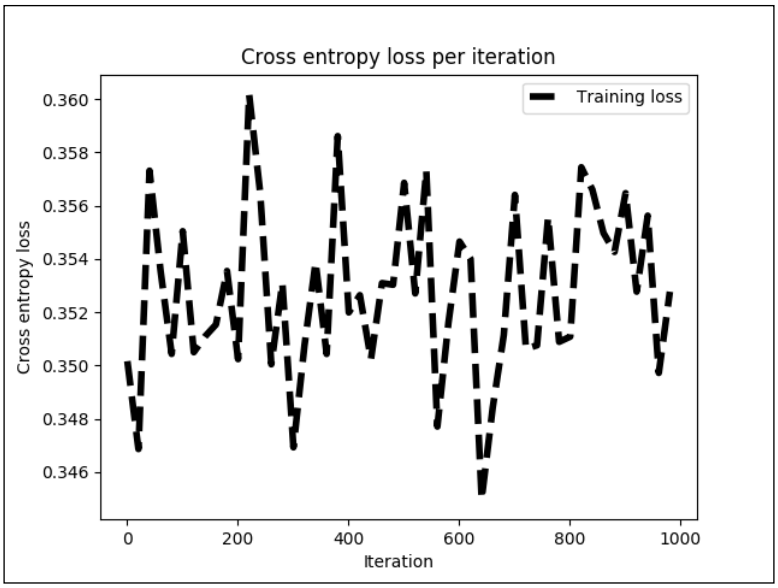


Figure 19: Cross entropy loss per iteration in the training phase

The preceding figure shows that the cross-entropy loss is more or less stable between 0.34 and 0.36, but with a little fluctuation. Now, let's see how this affects the training accuracy overall:

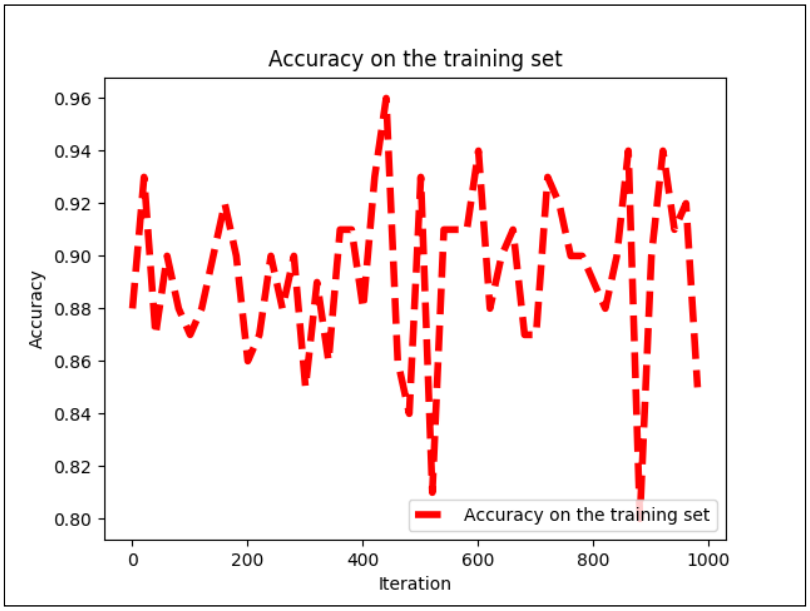


Figure 20: Accuracy on the training set on each iteration

We can see that the training accuracy fluctuates between 79% and 96% but does not increase or decrease for the most part, which is not that good either. So one possible way around it is to add more hidden layers and use different optimizers, such as Gradient Descent discussed earlier in this chapter.

9. Evaluating the model.

   As we described earlier, we will increase the dropout probability to 100%, that is, 1.0, so that we can have the same network that will be used for testing as well, that is, for evaluation with: `dropout_keep_prob == 1.0`:

```
print("Evaluating MLP on the test set...")
test_acc = sess.run(accuracy, feed_dict={X: data_test, y: labels_
test, dropout_keep_prob:1.})
print ("Prediction/clasification accuracy: %.3f" % (test_acc))
>>>
Evaluating MLP on the test set...
Prediction/classification accuracy: 0.889
Session closed!
```

   Thus, the classification accuracy is about 89%! Not bad at all. If the higher accuracy is desired, we can use another architecture of DNN called DBN, that can be trained either in a supervised or unsupervised way. The easiest way to see this is in its application as a classifier.

   If we have a DBN classifier such as in figure 14, then the pretraining method is done in an unsupervised way like the one in the autoencoder, and the classifier is trained (fine-tuned) in a supervised way exactly like the one in MLP. To get some more insight the following figure can be referred to:

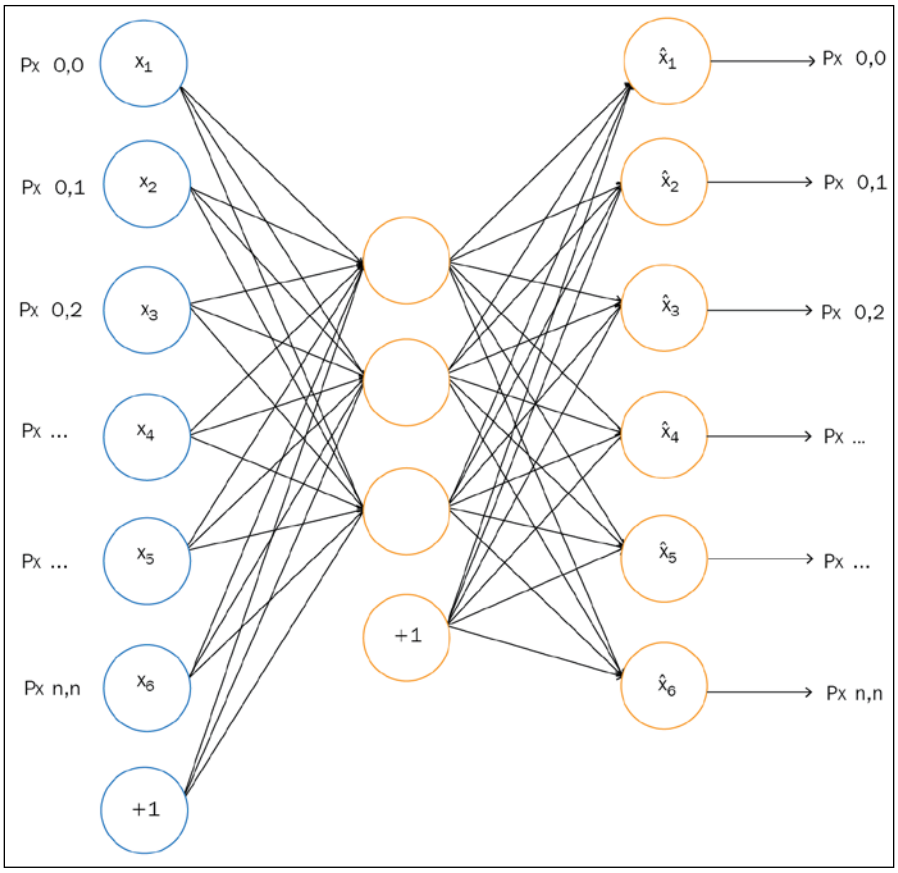


Figure 21: Sample implementation of artificial neural networks for unsupervised learning

Deep belief networks

To overcome the overfitting problem in MLP, we can set up a DBN, do unsupervised pretraining to get a decent set of feature representations for the inputs, then fine-tune on the training set to actually get predictions from the network. While weights of an MLP are initialized randomly, a DBN uses a greedy layer-by-layer pretraining algorithm to initialize the network weights through probabilistic generative models composed of a visible layer and multiple layers of stochastic, latent variables, which are called hidden units or feature detectors.

Restricted Boltzmann Machines (**RBM**) in the DBN are stacked, forming an undirected probabilistic graphical model similar to **Markov Random Fields** (**MRF**): the two layers are composed of visible neurons and then hidden neurons. The top two layers in a stacked RBM have undirected, symmetric connections between them and form an associative memory, whereas lower layers receive top-down, directed connections from the layer above:

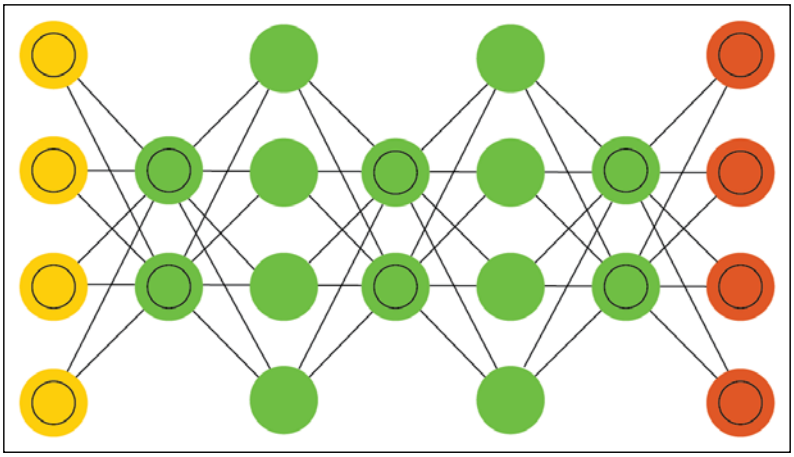


Figure 22: A high-level view of DBNs

The top two layers have undirected, symmetric connections between them and form an associative memory, whereas lower layers receive top-down, directed connections from the preceding layer. The building block of a deep belief network is RBM. Several RBMs are stacked one after another to form deep belief networks.

Restricted Boltzmann Machines

An RBM is an undirected probabilistic graphical model called Markov Random Fields. It consists of two layers. The first layer is composed of visible neurons and second layer consists of hidden neurons. Figure 23 shows the structure of a simple RBM. Visible units accept inputs and hidden units are nonlinear feature detectors. Each visible neuron is connected to all the hidden neurons, but there is no internal connection among neurons in the same layer:

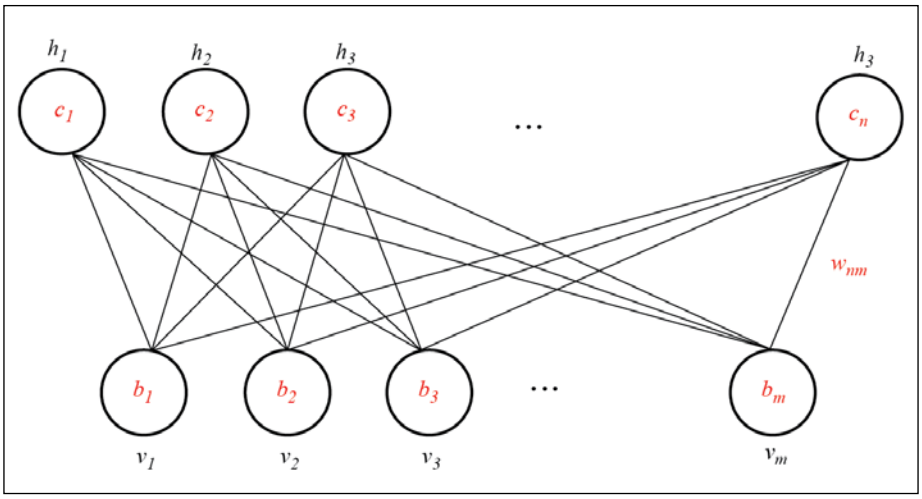


Figure 23: The structure of a simple RBM

The RBM in figure 23 consists of m visible units, $V = (v_1,\ldots,v_m)$ and n hidden units, $H = (h_1,\ldots,h_n)$. Visible units accept values between 0 and 1 and generated values of hidden units are between 0 and 1 as well. The joint probability of the model is an energy function given by the following equation:

$$E(v,h) = -\sum_{i=1}^{m} b_i v_i - \sum_{j=1}^{n} c_j h_j - \sum_{i=1}^{m}\sum_{j=1}^{n} v_i h_j w_i j \qquad (1)$$

In the preceding equation, $i = 1\ldots m$, $j = 1\ldots n$, b_i, and c_j are biases of visible and hidden units respectively, and w_ij is the weight between v_i and h_j. On the other hand, the probability assigned by the model to a visible vector v is s, given by the following equation:

$$p(v) = \frac{1}{Z}\sum_{h} e^{-E(v,h)} \qquad (2)$$

In equation 2, Z is a partition function, defined as follows:

$$Z = \sum_{v,h} e^{-E(v,h)} \qquad (3)$$

The weight can be attained with the following equation:

$$\Delta w_{ij} = \in \left(v_i h_{j_{data}} - v_i h_{j_{model}} \right) \qquad (4)$$

In equation 4, the learning rate is defined by ϵ. In general, a smaller value of ϵ ensures the training is more intensive. However, if you want your network to learn quickly, you can set this value higher.

It is easy to calculate the first term since there are no connections among units in the same layer. Conditional distributions of $p(h \mid v)$ and $p(v \mid h)$ are factorial and given by logistic functions in the following equations:

$$p\left(h_j = 1 \mid v\right) = g\left(c_j + \sum_i v_i w_{ij} \right) \qquad (5)$$

$$p\left(v_j = 1 \mid h\right) = g\left(b_i + \sum_j h_i w_{ij} \right) \qquad (6)$$

$$g\left(x\right) = \frac{1}{1 + \exp\left(-x\right)} \qquad (7)$$

Hence, the sample *vihj* is unbiased. However, calculating the log-likelihood of the second term is exponentially expensive. Although it is possible to get unbiased samples of the second term using Gibbs sampling by running the Markov chain Monte Carlo, this process is not cost effective either. Instead, RBM uses an efficient approximation method called **contrastive divergence**.

In general, the **Markov Chain Monte Carlo** (**MCMC**) requires many sampling steps to reach convergence to stationary. Running Gibbs sampling for a few steps (usually one) is enough to train a model, which is called contrastive divergence learning. The first step of contrastive divergence is to initialize the visible units with a training vector. Then, compute all hidden units using visible units at the same time with equation 5, then reconstruct visible units from hidden units using equation 4. Lastly, the hidden units are updated with the reconstructed visible units. Hence, instead of equation 4, we get the following weight learning model in the end:

$$\Delta w_{ij} = \in \left(v_i h_{j_{data}} - v_i h_{j_{recons}} \right) \qquad (8)$$

In short, this process tries to reduce the reconstruction error between input data and reconstructed data. Several iterations of parameter updates are required for the algorithm to converge, and iterations are called epochs. Input data is divided into mini batches and parameters are updated after each mini batch with the average values of the parameters.

Construction of a simple DBN

A single hidden layer RBM cannot extract all features from the input data due to its inability to model the relationship between variables. Hence, multiple layers of RBMs are used one after another to extract nonlinear features. In deep belief networks, an RBM is trained with input data first, and the hidden layer represents learned features in a greedy learning approach. These learned features of the first RBM, that is, a hidden layer of the first RBM, are used as the input of the second RBM as another layer in the DBN, as shown in figure 6. Similarly, learned features of the second layer are used as input for another layer.

This way, deep belief networks can extract deep and nonlinear features from input data. The hidden layer of the last RBM represents the learned features of the whole network. The process of learning features described earlier for all RBM layers is called pretraining and will be discussed in the next subsection.

Unsupervised Pretraining

Suppose you want to tackle a complex task for which you don't have much-labeled training data; in that case, it would be difficult to find a suitable DNN implementation or architecture to be trained and used for predictive analytics.

Nevertheless, if you have plenty of unlabeled training data, you can try to train the layers one by one, starting with the lowest layer and then going up, using an unsupervised feature detector algorithm. This is how exactly RBMs (figure 31) or autoencoders (figure 30) work.

However, unsupervised pretraining is still a good option when you have a complex task to solve, no similar model you can reuse, and little-labeled training data but plenty of unlabeled training data. However, the current trend is to use autoencoders rather than RBMs. But for the example that will be demonstrated in the next section, RBMs will be used to make the example simpler. Readers can give it a try using autoencoders rather than RBMs too:

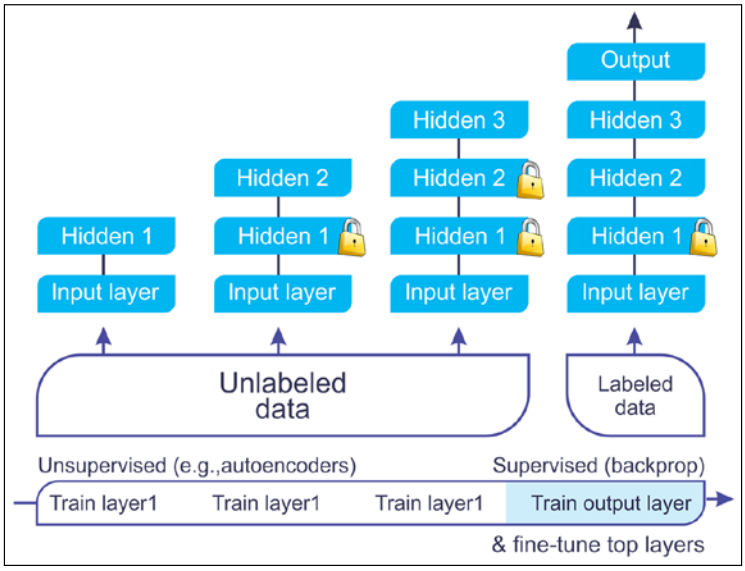


Figure 30: Unsupervised pretraining in DBN using autoencoders

Pretraining is an unsupervised learning process. After pretraining, fine-tuning of the network is carried out by adding a labeled layer at the top of the last RBM layer. This step is a supervised learning process. The unsupervised pretraining step tries to find network weights, which is close to a good solution:

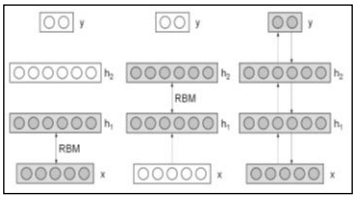


Figure 31: Unsupervised pretraining in DBN by constructing a simple DBN with a stack of RBMs

Then, in the supervised learning, instead of randomly initializing network weights, they are initialized with the weights computed in the pretraining step. This way, the DBN can avoid converging to a local minimum when a supervised gradient descent is used.

As stated earlier, using a stack of RBMs, a DBN can be constructed as follows: at first, we need to train the bottom RBM (first RBM) with the parameter *W*1. Then, we need to initialize the second layer weights to *W*2 = *W*1T, which ensures that the two-hidden layer DBN is at least as good as our base RBM. So, putting these together, figure 32 shows the construction of a simple DBN consisting of three RBMs:

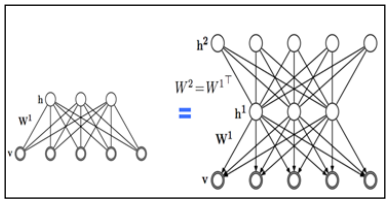


Figure 32: Construction of a simple DBN using several RBMs

Now, when it comes to tuning a DBN for better predictive accuracy, we should tune several hyperparameters so that DBNs fit to the training data by untying and refining *W*2. Putting this all together, here's the conceptual workflow for creating a DBN-based classifier or regressor:

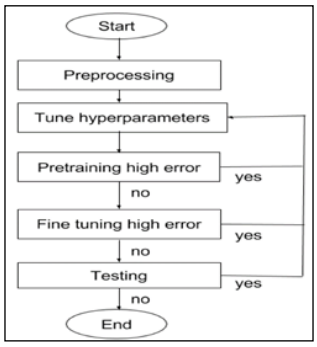


Figure 33: DBN in a nutshell

Now that we have enough theoretical background on how to construct a DBN using several RBMs, it's time to apply our theory in practice. In the next section, we will see how we can develop a supervised DBN classifier for predictive analytics.

Using deep belief networks for predictive analytics

In the previous example on the bank marketing dataset, we observed about 89% classification accuracy using MLP. We also normalized the original dataset before feeding it to the MLP.

In this section, we will see how to use the same datasets for the DBN-based predictive model. We will use the customized and extended version of DBN implantation called deep-belief-network that can be downloaded from GitHub at `https://github.com/albertbup/deep-belief-network`. The deep-belief-network is a simple, clean, fast Python implementation of deep belief networks based on binary Restricted Boltzmann Machines (RBM), built upon NumPy and TensorFlow libraries in order to take advantage of GPU computation. This library is implemented based on the following two research papers:

- Hinton, Geoffrey E., Simon Osindero, and Yee-Whye Teh. *A fast learning algorithm for deep belief nets* Neural Computation 18.7 (2006): 1527-1554.
- Fischer, Asja, and Christian Igel. *Training restricted Boltzmann machines: an introduction* Pattern Recognition 47.1 (2014): 25-39.

We will see how to train the RBMs in an unsupervised way and then we will train the ANNs in a supervised way. In short, there are several steps to be followed:

1. Loading required modules and libraries:

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.metrics.classification import accuracy_score
from sklearn.metrics import precision_recall_fscore_support
from sklearn.metrics import confusion_matrix
import itertools
from tf_models import SupervisedDBNClassification
import matplotlib.pyplot as plt
```

2. Preparing the training and test set.

   We load the already normalized dataset used in the previous MLP example:

```
FILE_PATH = 'input/bank_normalized.csv'
raw_data = pd.read_csv(FILE_PATH)
```

   In the preceding code, we have used the pandas `read_csv()` method and have created a DataFrame. Now, the next task is to spate the features and labels that can be done as follows:

```
Y_LABEL = 'y'
KEYS = [i for i in raw_data.keys().tolist() if i != Y_LABEL]
X = raw_data[KEYS].get_values()
Y = raw_data[Y_LABEL].get_values()
class_names = list(raw_data.columns.values)
print(class_names)
```

In the preceding lines, we have separated the features and labels. The features are stored in `X` and the labels are in `Y`. Now, the next task is to split them into the train (75%) and test sets (25%) as follows:

```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_
size=0.25, random_state=100)
```

3. Model building and training the DBN.

   Now that we have the training and test sets, we can go to the DBN training step directly. However, at first we need to instantiate the DBN, we will do it in a supervised way for classification, but of course, we need to provide the hyperparameters for this DNN architecture:

```
classifier = SupervisedDBNClassification(hidden_layers_
structure=[64, 64],
learning_rate_rbm=0.05,
learning_rate=0.01,
n_epochs_rbm=10,
n_iter_backprop=100,
batch_size=32,
activation_function='relu',
dropout_p=0.2)
```

   This library has an implementation to support Sigmoid, ReLU, and Tanh activation functions. Also, it utilizes the l2 regularization to avoid overfitting. Now we will do the actual fitting as follows:

```
classifier.fit(X_train, Y_train)
```

   If everything goes fine, you should observe the following progress on the console:

```
[START] Pre-training step:
>> Epoch 1 finished     RBM Reconstruction error 1.681226
….
>> Epoch 10 finished     RBM Reconstruction error 4.926415
>> Epoch 1 finished     RBM Reconstruction error 7.185334
…
>> Epoch 10 finished     RBM Reconstruction error 37.734962
>> Epoch 1 finished      RBM Reconstruction error 467.182892
….
>> Epoch 10 finished      RBM Reconstruction error 938.583801
[END] Pre-training step
[START] Fine tuning step:
```

```
>> Epoch 0 finished     ANN training loss 0.316619
>> Epoch 1 finished     ANN training loss 0.311203
>> Epoch 2 finished     ANN training loss 0.308707
….
>> Epoch 98 finished     ANN training loss 0.288299
>>        Epoch        99         finished                    ANN
training         loss        0.288900
```

Now, after 100 iterations, the fine-tuning graph showing the training gloss per epoch is as follows:

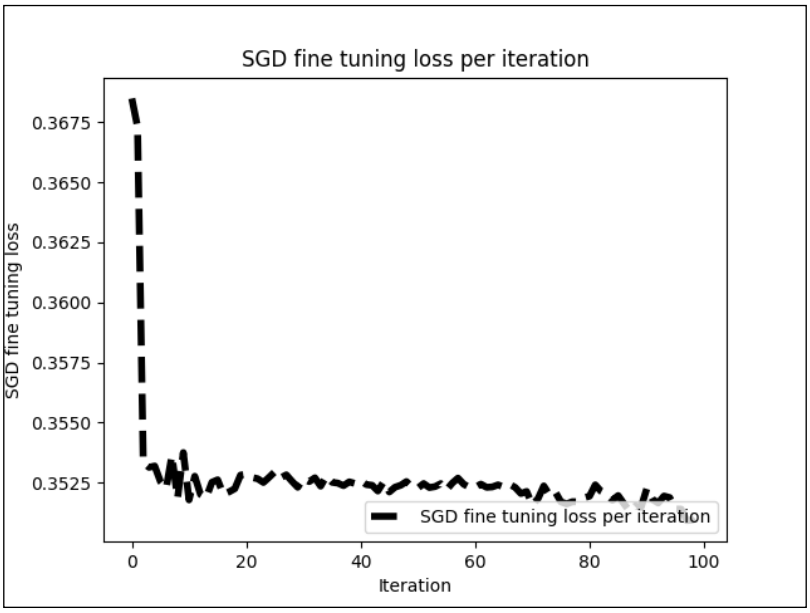


Figure 34: SGD fine-tuning loss per iteration (only 100 iterations)

However, when I iterated the preceding training and fine-tuning up to 1000 epochs, I didn't see any significant improvement in the training loss:

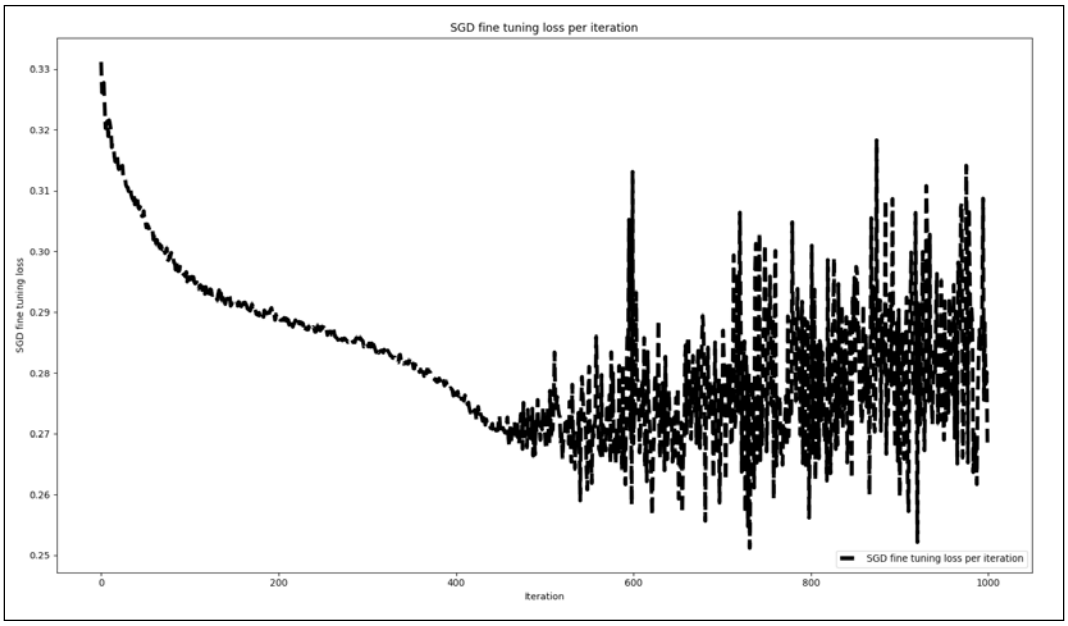


Figure 35: SGD fine-tuning loss per iteration (1000 iterations)

Here's the implementation of supervised DBN classifiers. This class implements a deep belief network for classification problems. It converts network output to original labels. It takes network parameters and returns a list after performing index to label mapping.

It predicts the probability distribution of classes for each sample in the given data and returns a list of dictionaries, one per sample. Finally, it appends a Softmax Linear Classifier as an output layer:

```
class SupervisedDBNClassification(TensorFlowAbstractSupervisedDBN,
ClassifierMixin):
    def _build_model(self, weights=None):
        super(SupervisedDBNClassification, self)._build_
model(weights)
        self.output = tf.nn.softmax(self.y)
        self.cost_function = tf.reduce_mean(tf.nn.softmax_cross_
entropy_with_logits(logits=self.y, labels=self.y_))
        self.train_step = self.optimizer.minimize(self.cost_
function)
    @classmethod
    def _get_param_names(cls):
        return super(SupervisedDBNClassification, cls)._get_param_
names() + ['label_to_idx_map', 'idx_to_label_map']
    @classmethod
```

```
    def from_dict(cls, dct_to_load):
        label_to_idx_map = dct_to_load.pop('label_to_idx_map')
        idx_to_label_map = dct_to_load.pop('idx_to_label_map')
        instance = super(SupervisedDBNClassification, cls).from_
dict(dct_to_load)
        setattr(instance, 'label_to_idx_map', label_to_idx_map)
        setattr(instance, 'idx_to_label_map', idx_to_label_map)
        return instance
    def _transform_labels_to_network_format(self, labels):
        new_labels, label_to_idx_map, idx_to_label_map = to_
categorical(labels, self.num_classes)
        self.label_to_idx_map = label_to_idx_map
        self.idx_to_label_map = idx_to_label_map
        return new_labels
    def _transform_network_format_to_labels(self, indexes):
        return list(map(lambda idx: self.idx_to_label_map[idx],
indexes))
    def predict(self, X):
        probs = self.predict_proba(X)
        indexes = np.argmax(probs, axis=1)
        return self._transform_network_format_to_labels(indexes)
    def predict_proba(self, X):
        return super(SupervisedDBNClassification, self)._compute_
output_units_matrix(X)
    def predict_proba_dict(self, X):
        if len(X.shape) == 1:  # It is a single sample
            X = np.expand_dims(X, 0)
        predicted_probs = self.predict_proba(X)
        result = []
        num_of_data, num_of_labels = predicted_probs.shape
        for i in range(num_of_data):
            # key : label
            # value : predicted probability
            dict_prob = {}
            for j in range(num_of_labels):
                dict_prob[self.idx_to_label_map[j]] = predicted_
probs[i][j]
            result.append(dict_prob)
        return result
    def _determine_num_output_neurons(self, labels):
        return len(np.unique(labels))
```

As we have mentioned in the previous example and the running section, fine-tuning the parameters of a neural network is a tricky process, and there are many different approaches out there, but there is no one size fits all best approach, to my knowledge. With the preceding combination, I have received better classification results. Another important parameter to select is the learning rate. Thus, adapting the learning rate as your model goes is an approach that can be taken in order to reduce training time while avoiding local minimums. Here, I would like to discuss some tips that really helped me to get better predictive accuracy, not only for this application, but for others as well.



For more information about this library, visit `https://github.com/albertbup/deep-belief-network`.

Now that we have our model built, it's time to evaluate its performance.

1. Evaluating the model.

   To evaluate the classification accuracy, we will use several performance metrics, such as `precision`, `recall` and the `F1` score. Moreover, we will draw the confusion matrix to observe the predicted labels against the true labels.

   At first, let's compute the prediction accuracy as follows:

```
Y_pred = classifier.predict(X_test)
print('Accuracy: %f' % accuracy_score(Y_test, Y_pred))
```

   Then, we compute the `precision`, `recall`, and `F1` score of the classification:

```
p, r, f, s = precision_recall_fscore_support(Y_test, Y_pred,
average='weighted')
print('Precision:', p)
print('Recall:', r)
print('F1-score:', f)
>>>
Accuracy: 0.892007
Precision: 0.874006414768
Recall: 0.892007380791
F1-score: 0.848183108881
```

Now, using DBN, we have solved a classification problem. If you want to solve a regression problem where the labels to be predicted are continuous, you will have to use the `SupervisedDBNRegression()` function. You just need to prepare your dataset so that they can be consumed by the TensorFlow-based DBN. That's it. One possible and good datasets is *House Prices: Advanced Regression Techniques*, which can be accessed on the Kaggle website at `https://www.kaggle.com/c/house-prices-advanced-regression-techniques/data`. The target here is to predict sales prices.

Summary

In this chapter, we have seen how DNNs act as the core of Deep Learning. They are versatile, powerful, and scalable, making them ideal to tackle large and highly complex machine learning tasks, such as classifying billions of images, powering speech recognition services, and recommending the best videos to watch to hundreds of millions of users every day by stacking multiple ANNs together in forms called DNNs. We have discussed how to train DNNs and analyze the performance metrics that are needed to evaluate a DNN-predictive model.

Finally, we have discussed how to tune the hyperparameters for DNNs for better and optimized performance. By combining all of these, we have seen how to build very robust and accurate predictive models for predictive analytics as well. In particular, we have provided two practical examples using deep belief networks and multilayer perceptrons on a bank marketing dataset. We have experienced more than 89% prediction accuracy.

In *Chapter 8*, *Using Convolutional Neural Networks for Predictive Analytics*, we will see how we can use **Convolutional Neural Networks** (**CNNs**) to overcome the limitations of DNNs for handling unstructured datasets and digital objects such as images. We will see three examples: emotion prediction, image classification, and sentiment prediction using CNN.

8
Using Convolutional Neural Networks for Predictive Analytics

In this chapter, readers will learn how to develop predictive analytics applications such as emotion recognition, image classification, and text classification using **convolutional neural networks** (**CNN**) on real image/text datasets. Finally, we will provide some pointers on how to tune and debug CNN-based networks for optimized performance.

The following topics will be covered in this chapter:

- The drawbacks of regular deep neural networks
- Convolutional neural network architectures
- The convolution operations and pooling layers
- Tuning CNNs
- Developing a predictive model for text classification
- Developing a predictive model for emotion recognition
- Using CNNs for image classification

CNNs and the drawbacks of regular DNNs

In predictive modeling, CNNs are a type of feedforward neural network in which the connectivity pattern between its neurons is based on the animal visual cortex. In the last few years, CNNs have managed to achieve and demonstrate outstanding performance on some complex visual tasks such as biomedical imaging, image search, self-driving cars, automatic image/video classification systems, and more.

Moreover, CNNs are not restricted to visual perception: they are also successful at other tasks, such as voice recognition and NLP.

You might be wondering why we need a separate chapter for CNN having similar DNN architecture. Why not simply use a regular deep neural network with fully connected layers for image recognition tasks? The thing is that although regular DNNs work fine for small images (for example, MNIST, CIFAR-10), it breaks down for larger images because of the huge number of parameters it requires.

For example, a 100 × 100 image has 10,000 pixels, and if the first layer has just 1,000 neurons (which already severely restricts the amount of information transmitted to the next layer), this means a total of 10 million connections. And that's just for the first layer.

CNNs solve this problem using partially connected layers. Because consecutive layers are only partially connected and because it heavily reuses its weights, a CNN has far fewer parameters than a fully connected DNN, which makes it much faster to train, reduces the risk of overfitting, and requires much less training data. Moreover, when a CNN has learned a kernel that can detect a particular feature, it can detect that feature anywhere on the image. In contrast, when a DNN learns a feature in one location, it can detect it only in that particular location. Since images typically have very repetitive features, CNNs are able to generalize much better than DNNs for image processing tasks such as classification, using fewer training examples.

Importantly, DNN has no prior knowledge of how pixels are organized; it does not know that nearby pixels are close. A CNN's architecture embeds this prior knowledge. Lower layers typically identify features in small areas of the images, while higher layers combine the lower-level features into larger features. This works well with most natural images, giving CNNs a decisive head start compared to DNNs.

For example, in the following figure 1, on the left, you can see a regular three-layer neural network. On the right, a ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. The red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be three (red, green, blue channels):

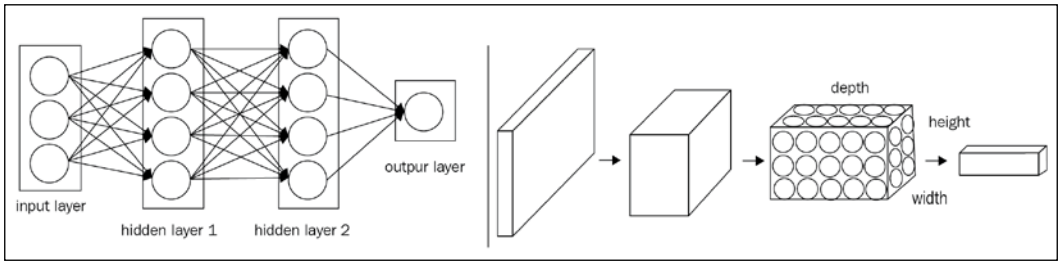


Figure 1: Regular DNN versus CNN

So, all the multilayer neural networks we looked at had layers composed of a long line of neurons, and we had to flatten input images or data to 1D before feeding them to the neural network. However, what happens once you try to feed them a 2D image directly? The answer is that in CNN, each layer is represented in 2D, which makes it easier to match neurons with their corresponding inputs. We will see examples of it in upcoming sections.

Another important fact is all the neurons in a feature map share the same parameters so it dramatically reduces the number of parameters in the model, but more importantly it means that once the CNN has learned to recognize a pattern in one location, it can recognize it in any other location. In contrast, once a regular DNN has learned to recognize a pattern in one location, it can recognize it only in that particular location.

CNN architecture

In multilayer networks such as MLP or DBN, the outputs of all neurons of the input layer would be connected to each neuron in the hidden layer, then the output will again act as the input to the fully-connected layer. In CNN networks, the connection scheme that defines the convolutional layer is significantly different. The convolutional layer is the main type of layer in CNN, where each neuron is connected to a certain region of the input area called the receptive field.

In a typical CNN architecture, a few convolutional layers are connected in a cascade style where each layer is followed by a **rectified linear unit** (**ReLU**) layer, then a pooling layer, then a few more convolutional layers (+ReLU), then another pooling layer, and so on.

The output from each convolution layer is a set of objects called feature maps that are generated by a single kernel filter. Then the feature maps can be used to define a new input to the next layer. Each neuron in a CNN network produces an output followed by an activation threshold which is proportional to the input and not bound:

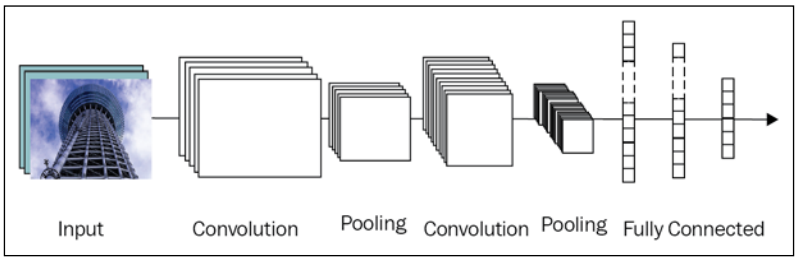


Figure 2: A conceptual architecture of CNN

As you can see in figure 2, the pooling layers are usually placed after the convolutional layers. The convolutional region is then divided by a pooling layer into sub regions. Then, a single representative value is selected using either a max-pooling or average pooling (see in the pooling layer section) technique to reduce the computational time of subsequent layers. This way, the robustness of the feature with respect to its spatial position gets increased too. To be more specific, when the image properties as feature maps pass through the image, it gets smaller and smaller as it progresses through the network, but it also typically gets deeper and deeper since more feature maps will be added.

At the top of the stack, a regular feedforward neural network is added just like an MLP which might compose of a few fully connected layers (+ReLUs), and the final layer outputs the prediction. For example, a softmax layer that outputs estimated class probabilities for a multiclass classification.

Convolutional operations

A convolution is a mathematical operation that slides one function over another and measures the integral of their pointwise multiplication. It has deep connections with the Fourier transform and the Laplace transform and is heavily used in signal processing. Convolutional layers actually use cross-correlations, which are very similar to convolutions.

Thus, the most important building block of a CNN is the convolutional layer: neurons in the first convolutional layer are not connected to every single pixel in the input image (like they were in previous chapters), but only to pixels in their receptive fields – see figure 3. In turn, each neuron in the second convolutional layer is connected only to neurons located within a small rectangle in the first layer:

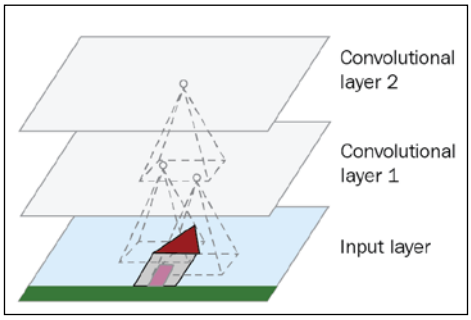


Figure 3: CNN layers with rectangular local receptive fields

This architecture allows the network to concentrate on low-level features in the first hidden layer, then assemble them into higher-level features in the next hidden layer, and so on. This hierarchical structure is common in real-world images, which is one of the reasons why CNNs work so well for image recognition.

Applying convolution operations in TensorFlow

TensorFlow provides a variety of methods for convolution. The canonical form is applied by the conv2d operation. Let's have a look at the usage of this operation:

```
tf.nn.conv2d(input,
filter,
strides,
padding,
use_cudnn_on_gpu,
data_format,
name=None)
```

The parameters we use are as follows:

- **Input**: This is the original tensor to which the operation will be applied. It has a definite format of four dimensions, and the default dimension order is shown below.

- **Filter**: This is a tensor representing a kernel or filter. It has a very generic method: [`filter_height`, `filter_width`, `in_channels`, `out_channels`]
- **Strides**: This is a list of four int tensor datatypes, which indicate the sliding windows for each dimension.
- **Padding**: This can be SAME or VALID: SAME will try to conserve the initial tensor dimension, but VALID will allow it to grow in case the output size and padding are computed. We will see later how to perform the padding along with the pooling layers.
- **use_cudnn_on_gpu**: This indicates whether or not to use the CUDA GPU CNN library to accelerate calculations.
- **data_format**: This specifies the order in which data is organized (NHWC or NCWH).

Following is an example of a convolutional layer, which concatenates a convolution, adds a bias parameter sum, and finally returns the activation function we have chosen for the whole layer (in this case, the ReLU operation, which is a frequently used one):

```
def conv_layer(data, weights, bias, strides=1):
  x = tf.nn.conv2d(x,
      weights,
      strides=[1, strides, strides, 1],
      padding='SAME')
  x = tf.nn.bias_add(x, bias)
  return tf.nn.relu(x)
```

Here, `x` is the 4D tensor input: [batch size, height, width, channel]. TensorFlow also offers a few other kinds of convolutional layers. For example:

- `tf.layers.conv1d()`: Creates a convolutional layer for 1D inputs. This is useful, for example, in natural language processing, where a sentence may be represented as a 1D array of words, and the receptive field covers a few neighboring words.
- `tf.layers.conv3d()`: Creates a convolutional layer for 3D inputs.
- `tf.nn.atrous_conv2d()`: Creates an atrous convolutional layer ("a trous" is French for "with holes"). This is equivalent to using a regular convolutional layer with a filter dilated by inserting rows and columns of zeros. For example, a 1 × 3 filter equal to [[1, 2, 3]] may be dilated with a dilation rate of 4, resulting in a dilated filter [[1, 0, 0, 0, 2, 0, 0, 0, 3]]. This allows the convolutional layer to have a larger receptive field at no computational price and using no extra parameters.

- `tf.layers.conv2d_transpose ()`: Creates a transpose convolutional layer, sometimes called a deconvolutional layer which up-samples an image. It does so by inserting zeros between the inputs, so you can think of this as a regular convolutional layer using a fractional stride. Up-sampling is useful, for example, in image segmentation: in a typical CNN, feature maps get smaller and smaller as you progress through the network, so if you want to output an image of the same size as the input, you need an up-sampling layer.
- `tf.nn.depthwise_conv2d()`: Creates a depth-wise convolutional layer that applies every filter to every individual input channel independently. Thus, if there are *f*n filters and *f*n′ input channels, then this will output *f*n × *f*n′ feature maps.
- `tf.layers.separable_conv2d()`: Creates a separable convolutional layer that first acts like a depth-wise convolutional layer, then applies a 1 × 1 convolutional layer to the resulting feature maps. This makes it possible to apply filters to arbitrary sets of input channels.

Pooling layer and padding operations

Once you understand how convolutional layers work, the pooling layers are quite easy to grasp. A pooling layer typically works on every input channel independently, so the output depth is the same as the input depth. You may alternatively pool over the depth dimension, as we will see next, in which case the image's spatial dimensions (height and width) remain unchanged, but the number of channels is reduced.

According to the TensorFlow website:

> *"The pooling ops sweep a rectangular window over the input tensor, computing a reduction operation for each window (average, max, or max with argmax). Each pooling op uses rectangular windows of size called ksize separated by offset strides. For example, if strides are all ones, every window is used, if strides are all twos, every other window is used in each dimension, and so on".*

So, in summary, the output can be computed as follows:

```
output[i] = reduce(value[strides * i:strides * i + ksize])
```

Just like in convolutional layers, each neuron in a pooling layer is connected to the outputs of a limited number of neurons in the previous layer, located within a small rectangular receptive field. You must define its size, the stride, and the padding type, just like before. However, a pooling neuron has no weights; all it does is aggregate the inputs using an aggregation function such as the max or mean.

Well, the goal of using pooling is to subsample the input image in order to reduce the computational load, the memory usage, and the number of parameters. This helps to avoid overfitting in the training stage. Reducing the input image size also makes the neural network tolerate a little bit of image shift.

In the following example, we use a 2 × 2 pooling kernel, a stride of 2, and no padding. Note that only the max input value in each kernel makes it to the next layer. The other inputs are dropped:

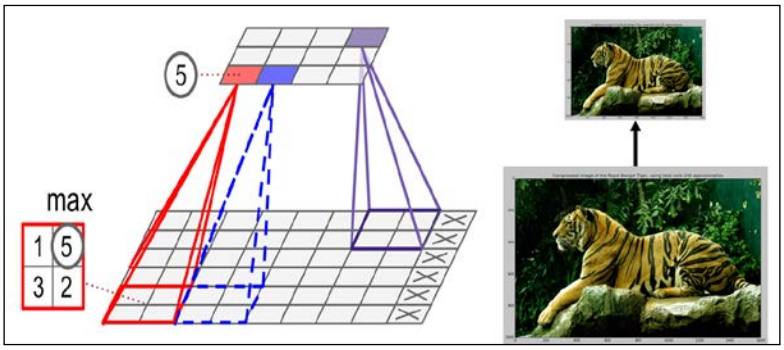


Figure 4: an example using max pooling, that is subsampling

Usually, *(stride_length)* x + filter_size <= input_layer_size* is recommended for most CNN-based network development.

Applying subsampling operations in TensorFlow

Using TensorFlow, a subsampling layer can normally be represented by a `max_pool` operation by maintaining the initial parameters of the layer:

```
import tensorflow as tf
def maxpool2d(x, k=2):
  return tf.nn.max_pool(x,
      ksize=[1, k, k, 1],
      strides=[1, k, k, 1],
      padding='SAME')
```

As stated earlier, a neuron located in a given layer is connected to the outputs of the neurons in the previous layer. Now, in order for a layer to have the same height and width as the previous layer, it is common to add zeros around the inputs, as shown in the diagram. This is called SAME or zero padding.

The term SAME means that the output feature map has the same spatial dimensions as the input feature map. Zero padding is introduced to make the shapes match as needed, equally on every side of the input map. On the other hand, VALID means no padding and only drops the right-most columns (or bottom-most rows):

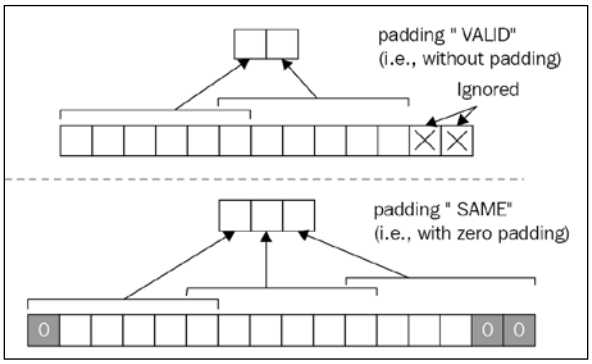


Figure 5: SAME versus VALID padding with CNN

Now, we need to analyze some of the more commonly used pool operations such as max pool and average pool that utilize padding operations. For the `max_pool`, it has the following signature in TensorFlow:

```
tf.nn.max_pool(value, ksize, strides, padding, data_format, name)
```

This method is similar to conv2d, and the parameters are as follows:

- `value`: This is a 4D tensor of float32 elements and shape (batch length, height, width, channels)
- `ksize`: This is a list of ints representing the window size on each dimension
- `strides`: This is the step of the moving windows on each dimension
- `data_format`: This sets the data dimensions
- `ordering`: NHWC or NCHW
- `padding`: VALID or SAME

However, depending upon the layering structures in CNN, there are other pooling operations supported by TensorFlow as follows:

- `tf.nn.avg_pool`: This returns a reduced tensor with the average of each window
- `tf.nn.max_pool_with_argmax`: This returns the `max_pool` tensor and a tensor with the flattened index of the `max_value`

- `tf.nn.avg_pool3d`: This performs an `avg_pool` operation with a cubic-like window; the input has an additional depth
- `tf.nn.max_pool3d`: This performs the same function as (...) but applies the max operation

Let's see an example of how these two work in TensorFlow. Suppose we have an input image of shape [`2`, `4`] which is 1 channel:

```
import tensorflow as tf
x = tf.constant([[2., 4., 6., 8.,],
                 [10., 12., 14., 16.]])
```

Now let's give it a shape accepted by `tf.nn.max_pool`:

```
x = tf.reshape(x, [1, 2, 4, 1])
```

If we want to apply the `VALID` padding with max pool with a 2 x 2 kernel, stride 2:

```
VALID = tf.nn.max_pool(x, [1, 2, 2, 1], [1, 2, 2, 1], padding='VALID')
```

On the other hand, using the max pool with a 2 x 2 kernel, stride 2, and `SAME` padding:

```
SAME = tf.nn.max_pool(x, [1, 2, 2, 1], [1, 2, 2, 1], padding='SAME')
```

For `VALID` padding, since there is no padding, the output shape is [`1`, `1`].

However, for the `SAME` padding, since we pad the image to the shape [`2`, `4`] (with `-inf` and then apply max pool), the output shape is [`1`, `2`]. Let's validate them:

```
print(VALID.get_shape())
print(SAME.get_shape())
>>>
(1, 1, 2, 1)
(1, 1, 2, 1)
```

Tuning CNN hyperparameters

In *Chapter 7, Using Deep Neural Networks for Predictive Analytics*, we have seen some ways to tune your DNN networks. However, since from the layering architecture's perspective CNN is different, it has a different requirement as well as tuning criteria.

Another problem with CNNs is that the convolutional layers require a huge amount of RAM, especially during training, because the reverse pass of back propagation requires all the intermediate values computed during the forward pass. During inference (that is, when making a prediction for a new instance) the RAM occupied by one layer can be released as soon as the next layer has been computed, so you only need as much RAM as required by two consecutive layers.

But during training, everything computed during the forward pass needs to be preserved for the reverse pass, so the amount of RAM needed is (at least) the total amount of RAM required by all layers. If your GPU runs out of memory while training a CNN, here are five things you could try to solve the problem (other than purchasing a GPU with more RAM):

- Reduce the mini-batch size
- Reduce dimensionality using a larger stride in one or more layers
- Remove one or more layers
- Use 16-bit floats instead of 32-bit
- Distribute the CNN across multiple devices (see more at `https://www.tensorflow.org/deploy/distributed`)

Another important question is when do you want to add a max pooling layer rather than a convolutional layer with the same stride? The thing is that a max pooling layer has no parameters at all, whereas a convolutional layer has quite a few.

sometimes, adding a local response normalization layer that makes the neurons that most strongly activate inhibit neurons at the same location but in neighboring feature maps, encourages different feature maps to specialize and pushes them apart, forcing them to explore a wider range of features. It is typically used in the lower layers to have a larger pool of low-level features that the upper layers can build upon.

As mentioned in *Chapter 7*, *Using Deep Neural Networks for Predictive Analytics* one of the main advantages observed during the training of large neural networks is overfitting, that is, generating very good approximations for the training data but emitting noise for the zones between single points.

In the case of overfitting, the model is specifically adjusted to the training dataset, so it will not be used for generalization. Therefore, although it performs well on the training set, its performance on the test dataset and subsequent tests is poor because it lacks the generalization property:

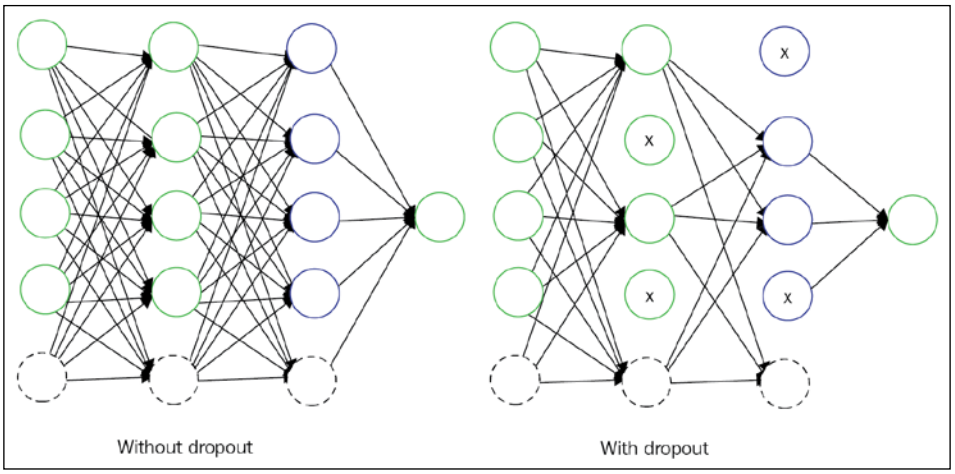


Figure 6: drop out versus without drop out

The main advantage of this method is that it avoids all neurons in a layer to synchronously optimize their weights. This adaptation made in random groups avoids all the neurons converging on the same goals, thus de-correlating the adapted weights. A second property discovered in the dropout application is that the activation of the hidden units becomes sparse, which is also a desirable characteristic.

In the preceding figure, we have a representation of an original, fully connected multilayer neural network and the associated network with the dropout linked. In order to apply the dropout operation, TensorFlow implements the `tf.nn.dropout` method, which works as follows:

```
tf.nn.dropout (x, keep_prob, noise_shape, seed, name)
```

Here, `x` is the original tensor. The `keep_prob` means the probability of keeping a neuron and the factor by which the remaining nodes are multiplied. The `noise_shape` signifies a four-element list that determines whether a dimension will apply zeroing independently or not.

Let's have a look at the following code segment:

```
import tensorflow as tf
X = [1.5, 0.5, 0.75, 1.0, 0.75, 0.6, 0.4, 0.9]
drop_out = tf.nn.dropout(X, 0.5)
```

```
sess = tf.Session()
with sess.as_default():
  print(drop_out.eval())
sess.close()

>>>
[ 3.          0.                  1.5         0.
  0.          1.20000005   0.          1.79999995]
```

In the preceding code segment, we will apply the dropout operation to a sample vector. Dropout will also work on transmitting the dropout to all the architecture-dependent units.

In the following example, you can see the results of applying dropout to the `x` variable, with a `0.5` probability of zeroing, and in the cases in which it didn't occur, the values were doubled (multiplied by 1/1.5, the dropout probability).

As a result, approximately half of the input was zeroed (this example was chosen to show that probabilities will not always give the expected four zeroes). One factor that could have surprised you is the scale factor applied to the non-dropped elements. This technique is used to maintain the same network, and restore it to the original architecture when training, using `dropout_keep_prob` as `1`.

Since in CNN, one of the objective functions is to minimize the evaluated cost, we must define an optimizer. In this case, we can adopt the implemented `RMSPropOptimizer` function which is an advanced form of gradient descent. The `RMSPropOptimizer` function implements the RMSProp algorithm. The `RMSPropOptimizer` function also divides the learning rate by an exponentially decaying average of squared gradients. The suggested setting value of the decay parameter is `0.9`, while a good default value for the learning rate is `0.001`:

```
optimizer = tf.train.RMSPropOptimizer(0.001, 0.9).minimize(cost_op)
```

Using the most common optimizer like **Stochastic Gradient Descent** (**SGD**), the learning rates must scale with 1/T to get convergence, where T is the number of iterations. RMSProp tries to overcome this limitation automatically by adjusting the step size so that the step is on the same scale as the gradients.

So if you're training a neural network but computing the gradients is mandatory, using `tf.train.RMSPropOptimizer()` would be the faster way of learning in a mini-batch setting. Researchers also recommend using Momentum optimizer while training a deep CNN or DNN.

Now, we have minimal theoretical knowledge on how to build your first CNN network for making a prediction.

In the next section, we will see how to develop a text classification predictive model using CNN. The task we will be seeing is to predict sentiments in movie reviews, negative or positive reviews, that is a binary classification problem.

CNN-based predictive model for sentiment analysis

In *Chapter 6*, *Predictive Analytics Pipelines for NLP*, we have seen how to develop a predictive analytics application for sentiment prediction. We have seen how to use the CBOW concept for word embedding and model building where a CBOW is a binary or frequency vector of all words in a document. Unfortunately, it loses word order. One potential limitation of this approach is that the feature size has to be equal to vocabulary size. The other popular method of doing this kind of predictive analytics is as follows:

- **Support Vector Machine** (**SVM**): Finds an optimal hyperplane to linearly separate data. Can work poorly if classes are not linearly separable.
- **Clustering with mean-NN classification**: Combine features into similar clusters by choosing the most popular class in the nearest cluster. It requires manually choosing cluster count. Also, the data may be difficult to separate into distinct clusters.
- **Naive Bayes**: Create probabilities of each word given a class and each class. Combine all probabilities of words in a document for each class. It then chooses a class with the highest probability. Nevertheless, it also assumes all words are independent, which is rarely true.
- **One-hot (sequence) vectors**: Single binary vector per word in a document. Advantageously, can keep the word order. But unfortunately, if the feature matrix is very sparse, feature size quickly becomes unmanageable as document size increases.
- **N-grams**: Is used with BOW or one-hot technique. It also uses the neighboring pairs of words as vocabulary instead of an individual.

However, none of these can perform satisfying accuracy but we have already observed only 50.6% prediction accuracy using a CBOW-based approach which is not good but disappointing.

In this section, therefore, we will try to see if we can use CNN for such a use case and experience much better accuracy. Well, the motivation here is that we know CNN is mostly suitable for handling image recognition, classification, or pattern recognition. However, classifying a movie review or a product review, that is sentiment, can also be conducted using a CNN-based predictive model that involves some sort of pattern mining too.



Sentiment analysis is about determining if a document is positive/negative towards a specific subject.

Exploring movie and product review datasets

More specifically, we will implement a CNN-based predictive model for movie review sentiment analysis and prediction for movie or product review. The movie review data that I'm going to use is downloaded from the **Cornell University** (**CU**) movie review dataset from: `http://www.cs.cornell.edu/people/pabo/movie-review-data/`. The second source for the movie review dataset and product reviews from Amazon is the UCI ML repository at: `https://archive.ics.uci.edu/ml/machine-learning-databases/00331/`.

The movie review dataset from CU has 5331 positive and 5331 negative processed sentences/snippets donated by Introduced in Pang/Lee ACL 2005. More specifically, there are two files namely rt-polarity.pos and rt-polarity.neg. The rt-polarity.pos contains 5331 positive snippets and rt-polarity.neg contains 5331 negative snippets.

The UCI ML repo contains the review dataset about movies from IMDb and the product review dataset from Amazon. In both datasets, there are negative and positive reviews for products or movies, however, later on I have processed them to separate the negative and positive respectively.

Note that for all of these datasets, each line in these two files corresponds to a single snippet usually containing roughly one single sentence. Also, it is noted that all snippets are lowercase. The snippets were labelled automatically but before labelling, the authors assumed the snippets were from the Rotten Tomatoes web pages because the reviews marked with 'fresh' are positive, and reviews marked with 'rotten' are negative.

Using CNN for predictive analytics about movie reviews

At first, we need to design the CNN-based classifier so that it can predict the sentiment about a movie, that is if a review is positive or negative:

1. Load required library and packages:

```
import tensorflow as tf
import numpy as np
import os
import time
import data_helpers
from text_cnn import TextCNN
from tensorflow.contrib import learn
from sklearn.metrics.classification import accuracy_score
from sklearn.metrics import precision_recall_fscore_support
from sklearn.metrics import confusion_matrix
import csv
```

2. Implement a CNN for text pipeline.

   In theory, we have seen that a CNN is a stacked layer which consists of an input layer, followed by a convolutional layer, max-pooling layer, a softmax layer, and output layers.

   However, this does not work for the text dataset since text cannot be convoluted. Therefore, we need an additional layer by means of words and text embedding. Let's name it the embedding layer. The goal is to learn a region-based text embedding as shown in figure 7:

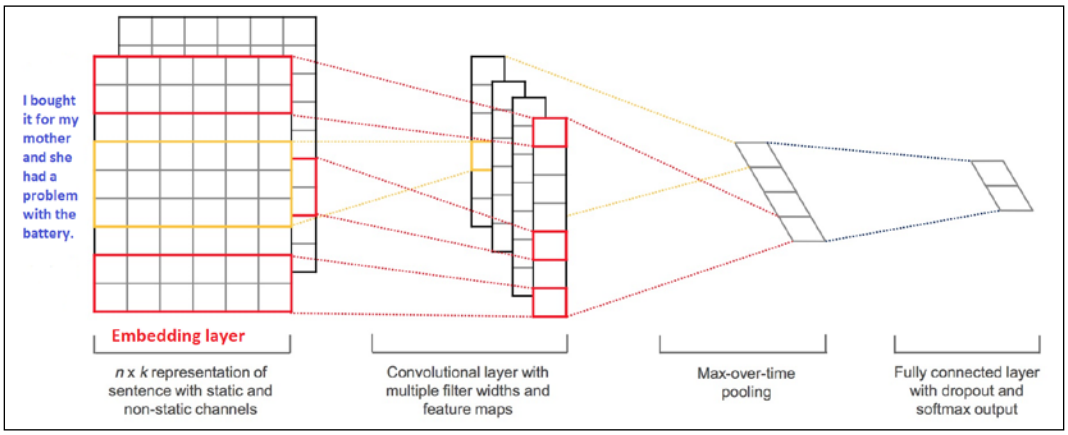


Figure 7: CNN-based text classification pipeline

Now the whole pipeline presented in figure 7 can be described as follows:

1. Initialize the constructor.
2. Create the placeholders for input, output, and the dropout => input layer.
3. Init the L2 regularization to avoid overfitting.
4. Add an embedding layering.
5. Adding convolutional and max-pooling layering.
6. Combine all the pooled features.
7. Add the drop-out.
8. Compute the prediction score and the accuracy.

Now, the following code exactly follows the preceding steps to implement a CNN-based text classification pipeline:

```
class TextCNN(object):
      # define the constructor
      def __init__(
      self, sequence_length, num_classes, vocab_size,
      embedding_size, filter_sizes, num_filters, l2_reg_
lambda=0.0):
      # Placeholders for input, output and dropout
        self.input_x = tf.placeholder(tf.int32, [None, sequence_
length], name="input_x")
        self.input_y = tf.placeholder(tf.float32, [None, num_
classes], name="input_y")
        self.dropout_keep_prob = tf.placeholder(tf.float32,
name="dropout_keep_prob")
        # Keeping track of l2 regularization loss (optional) to
avoid overfitting
        l2_loss = tf.constant(0.0)
        # Add an embedding layer
        with tf.device('/cpu:0'), tf.name_scope("embedding"):
            self.W = tf.Variable(
                tf.random_uniform([vocab_size, embedding_size],
-1.0, 1.0),
                name="W")
            self.embedded_chars = tf.nn.embedding_lookup(self.W,
self.input_x)
            self.embedded_chars_expanded = tf.expand_dims(self.
embedded_chars, -1)
        # Create a convolution + maxpool layer for each filter
size
        pooled_outputs = []
```

```
        for i, filter_size in enumerate(filter_sizes):
            with tf.name_scope("conv-maxpool-%s" % filter_size):
                # Convolution Layer
                filter_shape = [filter_size, embedding_size, 1,
num_filters]
                W = tf.Variable(tf.truncated_normal(filter_shape,
stddev=0.1), name="W")
                b = tf.Variable(tf.constant(0.1, shape=[num_
filters]), name="b")
                conv = tf.nn.conv2d(
                    self.embedded_chars_expanded,
                    W,
                    strides=[1, 1, 1, 1],
                    padding="VALID",
                    name="conv")
                # Apply nonlinearity
                h = tf.nn.relu(tf.nn.bias_add(conv, b),
name="relu")
                # Maxpooling over the outputs
                pooled = tf.nn.max_pool(
                    h,
                    ksize=[1, sequence_length - filter_size + 1,
1, 1],
                    strides=[1, 1, 1, 1],
                    padding='VALID',
                    name="pool")
                pooled_outputs.append(pooled)
        # Combine all the pooled features
        num_filters_total = num_filters * len(filter_sizes)
        self.h_pool = tf.concat(pooled_outputs, 3)
        self.h_pool_flat = tf.reshape(self.h_pool, [-1, num_
filters_total])
        # Add dropout
        with tf.name_scope("dropout"):
            self.h_drop = tf.nn.dropout(self.h_pool_flat, self.
dropout_keep_prob)
        # Final (unnormalized) scores and predictions
        with tf.name_scope("output"):
            W = tf.get_variable(
                "W",
                shape=[num_filters_total, num_classes],
                initializer=tf.contrib.layers.xavier_
initializer())
            b = tf.Variable(tf.constant(0.1, shape=[num_classes]),
name="b")
```

```
            l2_loss += tf.nn.l2_loss(W)
            l2_loss += tf.nn.l2_loss(b)
            self.scores = tf.nn.xw_plus_b(self.h_drop, W, b,
name="scores")
            self.predictions = tf.argmax(self.scores, 1,
name="predictions")
        # CalculateMean cross-entropy loss
        with tf.name_scope("loss"):
            losses = tf.nn.softmax_cross_entropy_with_
logits(logits=self.scores, labels=self.input_y)
            self.loss = tf.reduce_mean(losses) + l2_reg_lambda *
l2_loss
        # Accuracy
        with tf.name_scope("accuracy"):
            correct_predictions = tf.equal(self.predictions,
tf.argmax(self.input_y, 1))
            self.accuracy = tf.reduce_mean(tf.cast(correct_
predictions, "float"), name="accuracy")
```

In the preceding code, we have used the VALID padding since no padding is required.

3. Preparing/pre-processing the dataset.

   At first, we need to do some pre-processing, for example, clean up the strings about the review since the review text usually has a number, special characters, blank space, double quotations, and so on. Then it converts all the strings into lowercase. The following `clean_str()` function does all of this:

```
def clean_str(string):
    string = re.sub(r"[^A-Za-z0-9(),!?\'\`]", " ", string)
    string = re.sub(r"\'s", " \'s", string)
    string = re.sub(r"\'ve", " \'ve", string)
    string = re.sub(r"n\'t", " n\'t", string)
    string = re.sub(r"\'re", " \'re", string)
    string = re.sub(r"\'d", " \'d", string)
    string = re.sub(r"\'ll", " \'ll", string)
    string = re.sub(r",", " , ", string)
    string = re.sub(r"!", " ! ", string)
    string = re.sub(r"\(", " \( ", string)
    string = re.sub(r"\)", " \) ", string)
    string = re.sub(r"\?", " \? ", string)
    string = re.sub(r"\s{2,}", " ", string)
    return string.strip().lower()
```

Then we need to perform another typical step, that is to load the raw dataset about the positive and negative reviews dataset but add the processes described by the preceding `clean_str()` function. To be more specific, the following function (that is `load_data_and_labels()`), loads MR polarity data from files, splits the data into words, and generates labels. Finally, it returns split sentences and labels:

```
def load_data_and_labels(positive_data_file, negative_data_file):
    # Load data from files
    positive_examples = list(open(positive_data_file, "r").
readlines())
    positive_examples = [s.strip() for s in positive_examples]
    negative_examples = list(open(negative_data_file, "r").
readlines())
    negative_examples = [s.strip() for s in negative_examples]
    # Split by words
    x_text = positive_examples + negative_examples
    x_text = [clean_str(sent) for sent in x_text]
    # Generate labels
    positive_labels = [[0, 1] for _ in positive_examples]
    negative_labels = [[1, 0] for _ in negative_examples]
    y = np.concatenate([positive_labels, negative_labels], 0)
    return [x_text, y]
```

I believe the preceding code is pretty easy to understand. At first, we load the data files (that is negative and positive review files), then I have to tokenize them. Then I have extracted the corresponding labels. Finally, I have returned the tokens and the labels.

Well, now the thing is that we have the tokens and the labels, however the preceding tokens and the labels cannot be fed to the convolutional layers through placeholders and the tensors but we do need to generate the batch data that we feed. The following method `batch_iter()` generates a batch iterator for a dataset:

```
def batch_iter(data, batch_size, num_epochs, shuffle=True):
    data = np.array(data)
    data_size = len(data)
    num_batches_per_epoch = int((len(data)-1)/batch_size) + 1
    for epoch in range(num_epochs):
        # Shuffle the data at each epoch
        if shuffle:
            shuffle_indices = np.random.permutation(np.
arange(data_size))
            shuffled_data = data[shuffle_indices]
        else:
```

```
            shuffled_data = data
        for batch_num in range(num_batches_per_epoch):
            start_index = batch_num * batch_size
            end_index = min((batch_num + 1) * batch_size, data_
size)
            yield shuffled_data[start_index:end_index]
```

The preceding code generates the batch data by shuffling the data in each epoch. We will get to know about the training and hyper parameters during the model training.

4. Data loading and hyper parameters.

   At first, let's define some data loading params for example, validation sample percentage, that is dev and location of the data file containing the positive and negative review related texts. Here at first, I have used the Amazon product review dataset. If you want to do the prediction on the IMDb or CU dataset, just change the path:

```
tf.flags.DEFINE_float("dev_sample_percentage", .1, "Percentage of
the training data to use for validation")
tf.flags.DEFINE_string("positive_data_file", "./data/amazon/
amazon.pos", "Data source for the positive data.")
tf.flags.DEFINE_string("negative_data_file", "./data/amazon/
amazon.neg", "Data source for the negative data.")
```

   Now, we need to define the model hyperparameters such as the dimension of the embedding, number of filter sizes for the convolutional and the pooling layer, drop out probability, and the regularization lambda:

```
tf.flags.DEFINE_integer("embedding_dim", 128, "Dimensionality of
character embedding (default: 128)")
tf.flags.DEFINE_string("filter_sizes", "3,4,5", "Comma-separated
filter sizes (default: '3,4,5')")
tf.flags.DEFINE_integer("num_filters", 128, "Number of filters per
filter size (default: 128)")
tf.flags.DEFINE_float("dropout_keep_prob", 0.5, "Dropout keep
probability (default: 0.5)")
tf.flags.DEFINE_float("l2_reg_lambda", 0.0, "L2 regularization
lambda (default: 0.0)")
```

   Finally, we also need to define the training parameters such as size of the batch, number of epochs, when to cross validate and evaluate, when to check point, number of check points, and so on:

```
tf.flags.DEFINE_integer("batch_size", 64, "Batch Size (default:
64)")
tf.flags.DEFINE_integer("num_epochs", 200, "Number of training
epochs (default: 200)")
```

```
tf.flags.DEFINE_integer("evaluate_every", 100, "Evaluate model on
dev set after this many steps (default: 100)")
tf.flags.DEFINE_integer("checkpoint_every", 100, "Save model after
this many steps (default: 100)")
tf.flags.DEFINE_integer("num_checkpoints", 5, "Number of
checkpoints to store (default: 5)")
# Misc Parameters
tf.flags.DEFINE_boolean("allow_soft_placement", True, "Allow
device soft device placement")
tf.flags.DEFINE_boolean("log_device_placement", False, "Log
placement of ops on devices")
```

Additionally, if you want you can print those parameters as follows:

```
FLAGS = tf.flags.FLAGS
FLAGS._parse_flags()
print("\nParameters:")
for attr, value in sorted(FLAGS.__flags.items()):
    print("{}={}".format(attr.upper(), value))
print("")
```

5. Preparing training and test set.

   At first, we load the dataset. For this we use the `load_data_and_labels()` method:

   ```
   print("Loading data...")
   x_text, y = data_helpers.load_data_and_labels(FLAGS.positive_data_
   file, FLAGS.negative_data_file)
   ```

   Then we build the vocabulary. For this we use the `VocabularyProcessor()` method from the TensorFlow contrib package learn. Finally, we create the vocabulary:

   ```
   max_document_length = max([len(x.split(" ")) for x in x_text])
   vocab_processor = learn.preprocessing.VocabularyProcessor(max_
   document_length)
   x = np.array(list(vocab_processor.fit_transform(x_text)))
   ```

   We now have the dataset in numeric vectors. Still we need to randomly shuffle them to avoid imbalance as far as possible:

   ```
   np.random.seed(10)
   shuffle_indices = np.random.permutation(np.arange(len(y)))
   x_shuffled = x[shuffle_indices]
   y_shuffled = y[shuffle_indices]
   Then, we can split train/test set
   ```

Now we need to split the training and the test set as follows:

```
dev_sample_index = -1 * int(FLAGS.dev_sample_percentage *
float(len(y)))
x_train, x_dev = x_shuffled[:dev_sample_index], x_shuffled[dev_
sample_index:]
y_train, y_dev = y_shuffled[:dev_sample_index], y_shuffled[dev_
sample_index:]
print("Vocabulary Size: {:d}".format(len(vocab_processor.
vocabulary_)))
print("Train/Dev split: {:d}/{:d}".format(len(y_train), len(y_
dev)))
```

6. Training the model.

   This step is very important since our predictive analytics result is fully dependent on this and utilizes the CNN implementation shown in the preceding step. The following code does the training process based on the training and hyper parameters defined in step 4. Here's the workflow of the training:

   1. At first, we create the TensorFlow session and then inside an active session, invoke the `TextCNN()` class to instantiate the classifier.
   2. After this, we define the training procedure about the cost optimization using Adam optimizer.

      The target of creating the placeholders is so that batches of data in the placeholders can be passed through the tensors.

   3. Summaries for loss, accuracy, write, the train, and dev summaries.
   4. Checkpoint directory. TensorFlow assumes this directory already exists so we need to create it.
   5. Generate the vocabulary.
   6. Evaluate the training on the dev set.
   7. Finally, write the model in your preferred storage:

```
with tf.Graph().as_default():
    session_conf = tf.ConfigProto(
      allow_soft_placement=FLAGS.allow_soft_placement,
      log_device_placement=FLAGS.log_device_placement)
    sess = tf.Session(config=session_conf)
    with sess.as_default():
        cnn = TextCNN(
            sequence_length=x_train.shape[1],
            num_classes=y_train.shape[1],
```

```
            vocab_size=len(vocab_processor.vocabulary_),
            embedding_size=FLAGS.embedding_dim,
            filter_sizes=list(map(int, FLAGS.filter_sizes.
split(","))),
            num_filters=FLAGS.num_filters,
            l2_reg_lambda=FLAGS.l2_reg_lambda)
        # Define Training procedure
        global_step = tf.Variable(0, name="global_step",
trainable=False)
        optimizer = tf.train.AdamOptimizer(1e-3)
        grads_and_vars = optimizer.compute_gradients(cnn.loss)
        train_op = optimizer.apply_gradients(grads_and_vars,
global_step=global_step)
        # Keep track of gradient values and sparsity (optional)
        grad_summaries = []
        for g, v in grads_and_vars:
            if g is not None:
                grad_hist_summary = tf.summary.histogram("{}/grad/
hist".format(v.name), g)
                sparsity_summary = tf.summary.scalar("{}/grad/
sparsity".format(v.name), tf.nn.zero_fraction(g))
                grad_summaries.append(grad_hist_summary)
                grad_summaries.append(sparsity_summary)
        grad_summaries_merged = tf.summary.merge(grad_summaries)
        # Output directory for models and summaries
        timestamp = str(int(time.time()))
        out_dir = os.path.abspath(os.path.join(os.path.curdir,
"runs", timestamp))
        print("Writing to {}\n".format(out_dir))
        # Summaries for loss and accuracy
        loss_summary = tf.summary.scalar("loss", cnn.loss)
        acc_summary = tf.summary.scalar("accuracy", cnn.accuracy)
        # Write and the train  and dev summaries
        train_summary_op = tf.summary.merge([loss_summary, acc_
summary, grad_summaries_merged])
        train_summary_dir = os.path.join(out_dir, "summaries",
"train")
        train_summary_writer = tf.summary.FileWriter(train_
summary_dir, sess.graph)
        # Dev summaries
        dev_summary_op = tf.summary.merge([loss_summary, acc_
summary])
        dev_summary_dir = os.path.join(out_dir, "summaries",
"dev")
        dev_summary_writer = tf.summary.FileWriter(dev_summary_
dir, sess.graph)
```

```
        # Checkpoint directory. Tensorflow assumes this directory
already exists so we need to create it
        checkpoint_dir = os.path.abspath(os.path.join(out_dir,
"checkpoints"))
        checkpoint_prefix = os.path.join(checkpoint_dir, "model")
        if not os.path.exists(checkpoint_dir):
            os.makedirs(checkpoint_dir)
        saver = tf.train.Saver(tf.global_variables(), max_to_
keep=FLAGS.num_checkpoints)
        # Write vocabulary
        vocab_processor.save(os.path.join(out_dir, "vocab"))
        # Initialize all variables
        sess.run(tf.global_variables_initializer())
        def train_step(x_batch, y_batch):
            feed_dict = {
              cnn.input_x: x_batch,
              cnn.input_y: y_batch,
              cnn.dropout_keep_prob: FLAGS.dropout_keep_prob}
            _, step, summaries, loss, accuracy = sess.run(
                [train_op, global_step, train_summary_op, cnn.
loss, cnn.accuracy],
                feed_dict)
            time_str = datetime.datetime.now().isoformat()
            print("{}: step {}, loss {:g}, acc {:g}".format(time_
str, step, loss, accuracy))
            train_summary_writer.add_summary(summaries, step)

        def dev_step(x_batch, y_batch, writer=None):
            feed_dict = {
              cnn.input_x: x_batch,
              cnn.input_y: y_batch,
              cnn.dropout_keep_prob: 1.0
            }
            step, summaries, loss, accuracy = sess.run(
                [global_step, dev_summary_op, cnn.loss, cnn.
accuracy],
                feed_dict)
            time_str = datetime.datetime.now().isoformat()
            print("{}: step {}, loss {:g}, acc {:g}".format(time_
str, step, loss, accuracy))
            if writer:
                writer.add_summary(summaries, step)
        # Generate batches
        batches = data_helpers.batch_iter(list(zip(x_train, y_
train)), FLAGS.batch_size, FLAGS.num_epochs)
```

```
            # Training loop. For each batch...
            for batch in batches:
                x_batch, y_batch = zip(*batch)
                train_step(x_batch, y_batch)
                current_step = tf.train.global_step(sess, global_step)
                if current_step % FLAGS.evaluate_every == 0:
                    print("\nEvaluation:")
                    dev_step(x_dev, y_dev, writer=dev_summary_writer)
                    print("")
                if current_step % FLAGS.checkpoint_every == 0:
                    path = saver.save(sess, checkpoint_prefix, global_
step=current_step)
                    print("Saved model checkpoint to {}\n".
format(path))
```

7. Evaluating the model.

   We will evaluate our CNN model on the Amazon product, IMDb movie review, and CU movie review dataset. At first, we need to define the training, miscellaneous, and evaluation parameters for the training parameters required to train the CNN model:

   ```
   f.flags.DEFINE_string("positive_data_file", "./data/amazon/amazon.
   pos", "Data source for the positive data.")
   tf.flags.DEFINE_string("negative_data_file", "./data/amazon/
   amazon.neg", "Data source for the positive data.")
   ```

   Now, let's define all the evaluation parameters required to evaluate the CNN model:

   ```
   tf.flags.DEFINE_integer("batch_size", 64, "Batch Size (default:
   64)")
   tf.flags.DEFINE_string("checkpoint_dir", "./runs/1501543555/
   checkpoints/", "Checkpoint directory from training run")
   tf.flags.DEFINE_boolean("eval_train", True, "Evaluate on all
   training data")
   ```

   Here `1501543555` is a randomly generated directory containing the checkpoints and the final model. So please replace it with yours. Also set the check point directory accordingly.

   Then, we need to define some additional parameters for tuning, and so on. For example, whether you would like to allow the soft placement, logging the device placement, and so on:

   ```
   tf.flags.DEFINE_boolean("allow_soft_placement", True, "Allow
   device soft device placement")
   tf.flags.DEFINE_boolean("log_device_placement", False, "Log
   placement of ops on devices")
   ```

Finally, let's print the preceding parameters:

```
FLAGS = tf.flags.FLAGS
FLAGS._parse_flags()
print("\nParameters:")
for attr, value in sorted(FLAGS.__flags.items()):
    print("{}={}".format(attr.upper(), value))
print("")
>>>
ALLOW_SOFT_PLACEMENT=True
BATCH_SIZE=64
CHECKPOINT_DIR=./runs/1501541757/checkpoints/
EVAL_TRAIN=True
LOG_DEVICE_PLACEMENT=False
NEGATIVE_DATA_FILE=./data/imdb/IMDb.neg
POSITIVE_DATA_FILE=./data/imdb/IMDb.pos
```



Make sure that you have followed the data file path correctly. Negative reviews and positive reviews should be placed in separate folders.

Now it's time to load the new dataset. For simplicity, let's use the test set or your own generic test set containing a few review texts:

```
x_raw, y_test = data_helpers.load_data_and_labels(FLAGS.positive_
data_file, FLAGS.negative_data_file)
y_test = np.argmax(y_test, axis=1)
```

Then you have to map this data with the vocabulary we built in the previous step. Make sure that you have used the right path:

```
vocab_path = os.path.join("./runs/1501796488", "vocab")
vocab_processor = learn.preprocessing.VocabularyProcessor.
restore(vocab_path)
x_test = np.array(list(vocab_processor.transform(x_raw)))
```

Then, let's stat the evaluation. At first, we need to load the saved meta graph and restore variables from the check point directory. Then prepare the tensors we want to evaluate to get the prediction score. But we also need to generate batches for each epoch so that it can be evaluated during the TensorFlow session. Finally, we compute the raw prediction score:

```
print("\nEvaluating...\n")
checkpoint_file = tf.train.latest_checkpoint(FLAGS.checkpoint_dir)
graph = tf.Graph()
with graph.as_default():
    session_conf = tf.ConfigProto(
```

```
      allow_soft_placement=FLAGS.allow_soft_placement,
      log_device_placement=FLAGS.log_device_placement)
    sess = tf.Session(config=session_conf)
    with sess.as_default():
        # Load the saved meta graph and restore variables
        saver = tf.train.import_meta_graph("{}.meta".
format(checkpoint_file))
        saver.restore(sess, checkpoint_file)
        # Get the placeholders from the graph by name
        input_x = graph.get_operation_by_name("input_x").
outputs[0]
        # input_y = graph.get_operation_by_name("input_y").
outputs[0]
        dropout_keep_prob = graph.get_operation_by_name("dropout_
keep_prob").outputs[0]
        # Tensors we want to evaluate
        predictions = graph.get_operation_by_name("output/
predictions").outputs[0]
        # Generate batches for one epoch
        batches = data_helpers.batch_iter(list(x_test), FLAGS.
batch_size, 1, shuffle=False)
        # Collect the predictions here
        all_predictions = []
        for x_test_batch in batches:
            batch_predictions = sess.run(predictions, {input_x: x_
test_batch, dropout_keep_prob: 1.0})
            all_predictions = np.concatenate([all_predictions,
batch_predictions])
```

Well done! We have been able to compute the raw prediction. Now it's time to evaluate the model. We will use several performance metrics such as accuracy, precision, recall, f1-score, and confusion matrix:

```
correct_predictions = float(sum(all_predictions == y_test))
print("Total number of test examples: {}".format(len(y_test)))
print("Accuracy: {:g}".format(correct_predictions/float(len(y_
test))))
>>>
Accuracy: 0.978
```

Then, let's compute the precision, recall, and f1 score of the classification. For these, we will use the `precision_recall_fscore_support()` method from the sklearn Python module:

```
p, r, f, s = precision_recall_fscore_support(y_test, all_
predictions, average='weighted')
print('Precision:', p)
print('Recall:', r)
print('F1-score:', f)
print('Support:', s)
>>>
Precision: 0.978122399334
Recall: 0.978
F1-score: 0.97799859191
Support: None
```

Finally, let's compute, print, and plot the confusion matrix using the `confusion_matrix()` method from the sklearn Python module:

```
cnf_matrix = confusion_matrix(y_test, all_predictions)
np.set_printoptions(precision=2)
print(cnf_matrix)
>>>
[[493   7]
 [ 15 485]]
```

Fantastic! We have achieved about 98% accuracy which is outstanding. Now let's check with a sample review.

We will provide a small test set of review texts along with their labels but for this we need to replace these two lines with the customized and small test set as follows:

```
x_raw, y_test = data_helpers.load_data_and_labels(FLAGS.positive_
data_file, FLAGS.negative_data_file)
y_test = np.argmax(y_test, axis=1)
x_raw = ["I bought it for my mother and she had a problem with the
battery.", "I have yet to run this new battery below two bars and
that's three days without charging."]
y_test = [0, 1]
```

From the preceding text, you can see that the first review is negative since it contains some negative emotions regarding a problem. On the other hand, the second review is a positive review about a product. Now if you repeat the same step again, you should observe the following result:

```
Total number of test examples: 2
Accuracy: 1
Precision: 1.0
Recall: 1.0
F1-score: 1.0
Support: None
[[1 0]
 [0 1]]
```

That means 100% accuracy. You may still have doubts about the high predictive accuracy, this is normal. But we have more options to further validate the classifier's performance.

Let's use the IMDb movie review dataset. Just change the data file path and then show the right path of the vocabulary file, train the CNN model, and finally show the check point directory and the test set, that's it:

```
>>>
Total number of test examples: 995
Accuracy: 0.966834
Precision: 0.966835722082
Recall: 0.966834170854
F1-score: 0.966833969847
Support: None
[[484  16]
 [ 17 478]]
```

Again, above 96% accuracy. Still not convinced? Let's try with a larger dataset having, say, 21000 movie reviews, validate the training with 4000 reviews, and finally, evaluate the CNN model with 12500 reviews:

```
>>>
Total number of test examples: 10662
Accuracy: 0.971112
Precision: 0.971117732682
Recall: 0.971112361658
F1-score: 0.971112279324
Support: None
[[5168  163]
 [ 145 5186]]
```

Now it's above 97%. Simply outstanding, don't you think? Well, make sure that you have placed the data files, checkpoint directory, and vocabulary path correctly. For convenience, you can follow the path structure in my code.

We will move on to more general use cases that can be solved using the usual CNN architecture, for example, emotion recognition based on the given features or animal classification based on the given images. The very next section is about developing an emotion predictor for a set of given human faces.

CNN model for emotion recognition

One of the hardest problems to solve in deep learning has nothing to do with neural nets, it's the problem of getting the right data in the right format. In this section, we will see how to develop a CNN classifier that accurately predicts emotion from a set of given human faces.

Dataset description

More specifically, we will first train the CNN model based on the dataset from Kaggle and then we will test that model to test a human face to predict one of the given emotions from:

- Anger
- Disgust
- Fear
- Happy
- Sad
- Surprise
- Neutral

In this section, we show how to develop a CNN for emotion prediction from facial images. The train and test set of this example can be downloaded from: `https://inclass.kaggle.com/c/facial-keypoints-detector/data`.

There are 3761 grayscale images of 48x48 pixels in the training set. The shape of the train images is (3761, 48, 48, 1). The dataset was prepared for a supervised learning task -labels are associated with each image. Altogether, there are 3761 class labels, each class contains 7 elements, that is label set size. That is each element encodes an emotional stretch: 0 = anger, 1 = disgust, 2 = fear, 3 = happy, 4 = sad, 5 = surprise, 6 = neutral. Thus, the shape of the train labels is (3761, 7).

For example, the label set for the first image is [0. 0. 0. 1. 0. 0. 0.]. Now, if you map it to the list of available emotions, you will see that it is an image which is classified/ labeled as a happy face. More specifically, it corresponds to a happy emotional stretch that we visualize in the following:

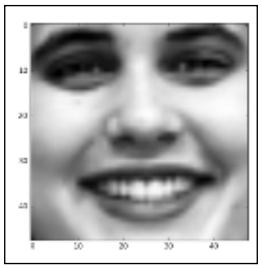


Figure 8: The first image from the emotion detection face dataset

On the other hand, the test set is formed by 1312 grayscale images of 48x48 pixel size which means the shape of the test labels is (1312, 48, 48, 1). Finally, a single image has the shape of (48, 48, 1).

CNN architecture design

We shall now proceed to study the CNN architecture. The following figure shows how the data flows in the CNN that will be implemented:

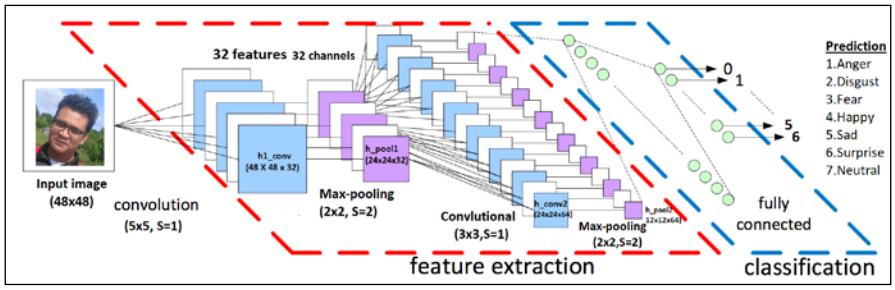


Figure 9: The first two convolutional layers of the implemented CNN

So, our network has two convolutional layers, two fully-connected layers (FCL), and finally a softmax classification layer. Each input image (48 pixels) is processed in the first convolutional layer with 5x5 convolutional kernels. This results in 32 features, one for each filter that we used. All the images are then down-sampled (from 48x48 to 24x24 pixels) by a max-pooling operation to decrease the images. These 32 down-sampled images are then processed by a second convolutional layer. The output of the second convolutional layer produces 64 new features (see figure 9). Then the second pooling operation then down-sampled them into 12x12 pixels.

The output of the second pooling layer has 64 images, 12x12 pixels each. These are then flattened to a single vector of length 12x12x64 = 9126, which is used as the input to the FCL. The FCL has 256 neurons. The output of this FCL is then fed into another FCL with 6 neurons. These 6 neurons are used to determine the class of the image. This way, the emotions are decoded and depicted in the image.

Now let's get on with the weights and bias definition. The following data structure represents the definition of the network's weights and summarizes what we have previously described:

```
weights = {
    'wc1': weight_variable([5, 5, 1, 32], name="W_conv1"),
    'wc2': weight_variable([3, 3, 32, 64],name="W_conv2"),
    'wf1': weight_variable\
           ([(IMAGE_SIZE // 4) * (IMAGE_SIZE // 4) * 64,\
                                              256],name="W_fc1"),
    'wf2': weight_variable([256, NUM_LABELS], name="W_fc2")
}
```

Note again, that the convolutional filters are randomly chosen, so the classification is done randomly:

```
def weight_variable(shape, stddev=0.02, name=None):
    initial = tf.truncated_normal(shape, stddev=stddev)
    if name is None:
        return tf.Variable(initial)
    else:
        return tf.get_variable(name, initializer=initial)
```

In a similar way, we have defined the bias:

```
biases = {
    'bc1': bias_variable([32], name="b_conv1"),
    'bc2': bias_variable([64], name="b_conv2"),
    'bf1': bias_variable([256], name="b_fc1"),
    'bf2': bias_variable([NUM_LABELS], name="b_fc2")
}

def bias_variable(shape, name=None):
    initial = tf.constant(0.0, shape=shape)
    if name is None:
        return tf.Variable(initial)
    else:
        return tf.get_variable(name, initializer=initial)
```

We then compute the error between the predicted and true class of the input image in terms of a `loss function`. It takes input and computes the predicted output of the model `pred` to the desired output `label`:

```
def loss(pred, label):
    cross_entropy_loss=
                tf.nn.softmax_cross_entropy_with_logits(pred, label)
    cross_entropy_loss= tf.reduce_mean(cross_entropy_loss)
    reg_losses = tf.add_n(tf.get_collection("losses"))
    return cross_entropy_loss + REGULARIZATION * reg_losses
```

The `tf.nn.softmax_cross_entropy_with_logits`(pred, label) function computes the `cross_entropy_loss` of the result after applying the softmax function (but it does it in a more considered, mathematical way). It's like the result of:

```
a = tf.nn.softmax(x)
b = cross_entropy(a)
```

We calculate the `cross_entropy_loss` for each of the classified images so we'll measure how well the model performs on each image individually. We take the cross-entropy's average for the classified images:

```
cross_entropy_loss= tf.reduce_mean(cross_entropy_loss)
```

To prevent overfitting, we use L2 regularization that consists of inserting an additional term to the `cross_entropy_loss`:

```
reg_losses = tf.add_n(tf.get_collection("losses"))
return cross_entropy_loss + REGULARIZATION * reg_losses
def add_to_regularization_loss(W, b):
    tf.add_to_collection("losses", tf.nn.l2_loss(W))
    tf.add_to_collection("losses", tf.nn.l2_loss(b))
```

We built the network's weights and bias and their optimization procedure. However, like all the implemented networks, we must start the implementation importing all necessary libraries:

```
import tensorflow as tf
import numpy as np
import os, sys, inspect
from datetime import datetime
import Utility
```

We set the paths for storing the dataset on your computer, and the network parameters:

```
FLAGS = tf.flags.FLAGS
tf.flags.DEFINE_string("data_dir", "Dataset/", "Path to data files
(train and test)")
tf.flags.DEFINE_string("logs_dir", "Logs/CNN_logs/", "Logging path")
tf.flags.DEFINE_string("mode", "train", "mode: train (Default)/ test")

BATCH_SIZE = 128
LEARNING_RATE = 1e-3
MAX_ITERATIONS = 30000
REGULARIZATION = 1e-3
IMAGE_SIZE = 48
NUM_LABELS = 7
VALIDATION_PERCENT = 0.1
```

The `emotionCNN()` function implements our model:

```
def emotionCNN(dataset):
    with tf.name_scope("conv1") as scope:
        tf.summary.histogram("W_conv1", weights['wc1'])
        tf.summary.histogram("b_conv1", biases['bc1'])
        conv_1 = tf.nn.conv2d(dataset, weights['wc1'], strides=[1, 1,
1, 1], padding="SAME")
        h_conv1 = tf.nn.bias_add(conv_1, biases['bc1'])
        h_1 = tf.nn.relu(h_conv1)
        h_pool1 = max_pool_2x2(h_1)
        add_to_regularization_loss(weights['wc1'], biases['bc1'])

    with tf.name_scope("conv2") as scope:
        tf.summary.histogram("W_conv2", weights['wc2'])
        tf.summary.histogram("b_conv2", biases['bc2'])
        conv_2 = tf.nn.conv2d(h_pool1, weights['wc2'], strides=[1, 1,
1, 1], padding="SAME")
        h_conv2 = tf.nn.bias_add(conv_2, biases['bc2'])
        h_2 = tf.nn.relu(h_conv2)
        h_pool2 = max_pool_2x2(h_2)
        add_to_regularization_loss(weights['wc2'], biases['bc2'])

    with tf.name_scope("fc_1") as scope:
        prob = 0.5
        image_size = IMAGE_SIZE // 4
        h_flat = tf.reshape(h_pool2, [-1, image_size * image_size *
64])
```

```
            tf.summary.histogram("W_fc1", weights['wf1'])
            tf.summary.histogram("b_fc1", biases['bf1'])
            h_fc1 = tf.nn.relu(tf.matmul(h_flat, weights['wf1']) +
biases['bf1'])
            h_fc1_dropout = tf.nn.dropout(h_fc1, prob)

        with tf.name_scope("fc_2") as scope:
            tf.summary.histogram("W_fc2", weights['wf2'])
            tf.summary.histogram("b_fc2", biases['bf2'])
            pred = tf.matmul(h_fc1_dropout, weights['wf2']) +
biases['bf2']
        return pred
```

We defined the `main` function where we'll define the dataset, the input, and output placeholder variables and the main session to start the training procedure:

```
def main(argv=None):
```

The first operation in this function is to load the dataset for training and validation phase. We'll use the training set to teach the classifier to recognize the to-be-predicted labels, and we'll use the validation set to estimate the classifier performance:

```
train_images, train_labels, valid_images, valid_labels, test_images =
Utility.read_data(FLAGS.data_dir)
    print("Training set size: %s" % train_images.shape[0])
    print('Validation set size: %s' % valid_images.shape[0])
    print("Test set size: %s" % test_images.shape[0])
```

In the preceding code segment, the `read_data()` method from the `Utility.py` file actually read the train and test set and returns the contents that are required to build our CNN such as train images, train labels, validation images, validation labels, and finally test images. The function goes as follows:

```
def read_data(data_dir, force=False):
    def create_onehot_label(x):
        label = np.zeros((1, NUM_LABELS), dtype=np.float32)
        label[:, int(x)] = 1
        return label
    pickle_file = os.path.join(data_dir, "EmotionDetectorData.pickle")
    if force or not os.path.exists(pickle_file):
        train_filename = os.path.join(data_dir, "train.csv")
        data_frame = pd.read_csv(train_filename)
        data_frame['Pixels'] = data_frame['Pixels'].apply(lambda x:
np.fromstring(x, sep=" ") / 255.0)
        data_frame = data_frame.dropna()
        print("Reading train.csv ...")
```

```
        train_images = np.vstack(data_frame['Pixels']).reshape(-1,
IMAGE_SIZE, IMAGE_SIZE, 1)
        print(train_images.shape)
        train_labels = np.array(list(map(create_onehot_label, data_
frame['Emotion'].values))).reshape(-1, NUM_LABELS)
        print(train_labels.shape)
        permutations = np.random.permutation(train_images.shape[0])
        train_images = train_images[permutations]
        train_labels = train_labels[permutations]
        validation_percent = int(train_images.shape[0] * VALIDATION_
PERCENT)
        validation_images = train_images[:validation_percent]
        validation_labels = train_labels[:validation_percent]
        train_images = train_images[validation_percent:]
        train_labels = train_labels[validation_percent:]
        print("Reading test.csv ...")
        test_filename = os.path.join(data_dir, "test.csv")
        data_frame = pd.read_csv(test_filename)
        data_frame['Pixels'] = data_frame['Pixels'].apply(lambda x:
np.fromstring(x, sep=" ") / 255.0)
        data_frame = data_frame.dropna()
        test_images = np.vstack(data_frame['Pixels']).reshape(-1,
IMAGE_SIZE, IMAGE_SIZE, 1)
        with open(pickle_file, "wb") as file:
            try:
                print('Picking ...')
                save = {
                    "train_images": train_images,
                    "train_labels": train_labels,
                    "validation_images": validation_images,
                    "validation_labels": validation_labels,
                    "test_images": test_images,
                }
                pickle.dump(save, file, pickle.HIGHEST_PROTOCOL)
            except:
                print("Unable to pickle file :/")
    with open(pickle_file, "rb") as file:
        save = pickle.load(file)
        train_images = save["train_images"]
        train_labels = save["train_labels"]
        validation_images = save["validation_images"]
        validation_labels = save["validation_labels"]
        test_images = save["test_images"]
    return train_images, train_labels, validation_images, validation_
labels, test_images
```

We then define the placeholder variable for the input images. This allows us to change the images that are input to the TensorFlow graph. The data type is set to `float32` and the shape is set to `[None, img_size, img_size, 1]`, where `None` means that the tensor may hold an arbitrary number of images with each image being `img_size` pixels high and `img_size` pixels wide and `1` is the number of color channels:

```
  dropout_prob = tf.placeholder(tf.float32)
    input_dataset = tf.placeholder(tf.float32, [None, IMAGE_SIZE,
IMAGE_SIZE, 1],name="input")
```

Next, we have the placeholder variable for the true labels associated with the images that were input in the placeholder variable `input_dataset`. The shape of this placeholder variable is [`None, NUM_LABELS`] which means it may hold an arbitrary number of labels and each label is a vector of length `NUM_LABELS`, which is 7 in this case:

```
input_labels = tf.placeholder(tf.float32,
[None, NUM_LABELS])
```

`The variable global_step` keeps track of the number of optimization iterations performed so far, we want to save this variable with all the other TensorFlow variables in the checkpoints.

Note that `trainable=False`, which means that TensorFlow will not try to optimize this variable:

```
global_step = tf.Variable(0, trainable=False)
```

And the following variable, `dropout_prob,` for dropout optimization:

```
    dropout_prob = tf.placeholder(tf.float32)
```

Now create the neural network for the test-phase. The `emotionCNN()` function returns the predicted class-labels `pred` for the `input_dataset`:

```
    pred = emotionCNN(input_dataset)
```

`The output_pred` are the predictions for the test and validation, which we'll compute in the running session:

```
    output_pred = tf.nn.softmax(pred,name="output")
```

`The variable loss_val` contains the error between the predicted class (`pred`) and the true class of the input image (`input_labels`):

```
    loss_val = loss(pred, input_labels)
```

`The train_op` defines the optimizer used to minimize the cost function. Since there is a lot of labeled training data, for faster and optimized training we use the momentum optimizer:

```
train_op = tf.train.MomentumOptimizer                (LEARNING_RATE).
minimize(loss_val, global_step)
```

And a `summary_op` for Tensorboard visualizations:

```
summary_op = tf.summary.merge_all()
```

Once the graph has been created, we have to create a TensorFlow session which is used to execute the graph:

```
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
summary_writer = tf.summary.FileWriter(FLAGS.logs_dir, sess.graph_def)
We then define a saver to restore the model:
        saver = tf.train.Saver()
        ckpt = tf.train.get_checkpoint_state(FLAGS.logs_dir)
        if ckpt and ckpt.model_checkpoint_path:
            saver.restore(sess, ckpt.model_checkpoint_path)
            print "Model Restored!"
```

Now we get a batch of training examples, `batch_image` now holds a batch of images and `batch_label` are the true labels for those images:

```
         for step in xrange(MAX_ITERATIONS):
            batch_image, batch_label = get_next_batch(train_images,
train_labels, step)
```

We put the batch into a dict with the proper names for placeholder variables in the TensorFlow graph:

```
            feed_dict = {input_dataset: batch_image, \
                         input_labels: batch_label}
```

We run the optimizer using this batch of training data; TensorFlow assigns the variables in `feed_dict_train`, to the placeholder variables and then runs the optimizer. We can also keep a separate list to hold the training error in each iteration for better visualization:

```
            if step % 10 == 0:
                train_loss, summary_str = sess.run([loss_val, summary_
op], feed_dict=feed_dict)
                summary_writer.add_summary(summary_str, global_
step=step)
                train_error_list.append(train_loss)
                train_step_list.append(step)
```

When the running step is a multiple of `100` we run the trained model on the validation set. We can also keep a separate list to hold the validation error in each iteration for better interpretation:

```
            if step % 100 == 0:
                valid_loss = sess.run(loss_val, feed_dict={input_
dataset: valid_images, input_labels: valid_labels})
                valid_error_list.append(valid_loss)
                valid_step_list.append(step)
                saver.save(sess, FLAGS.logs_dir + 'model.ckpt',
global_step=step)
```

At the end of the training session the model is saved:

```
saver.save(sess, FLAGS.logs_dir + 'model.ckpt', global_step=step)
```

Now, we can plot training loss over time, that is in each iteration:

```
        plt.plot(train_step_list, train_error_list, 'r--', label='CNN
training loss', linewidth=4)
        plt.title('CNN training loss per iteration')
        plt.xlabel('Iteration')
        plt.ylabel('CNN training loss')
        plt.legend(loc='upper right')
        plt.show()
```

The preceding code generates the following loss per iteration for the CNN training:

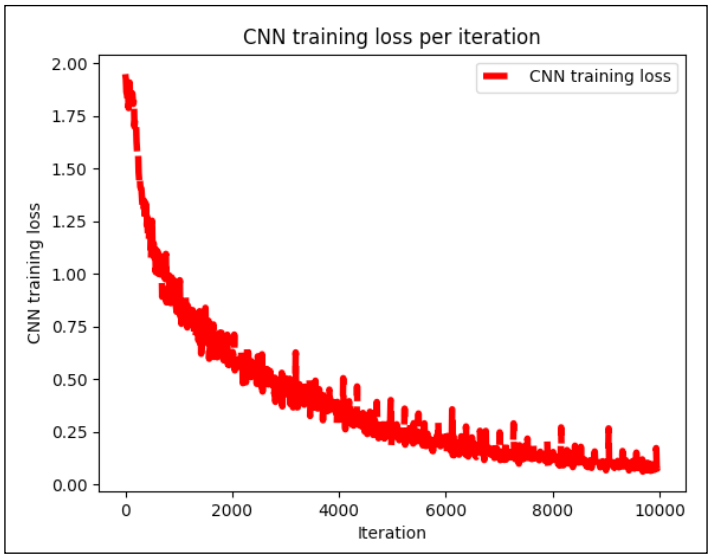


Figure 10: CNN training loss per iteration for emotion prediction

So, from the preceding figure, we can see that the loss function decreases during the simulation. Now we can plot validation loss over time, that is in each iteration:

```
        plt.plot(valid_step_list, valid_error_list, 'r--', label='CNN
validation loss', linewidth=4)
        plt.title('CNN validation loss per iteration')
        plt.xlabel('Iteration')
        plt.ylabel('CNN validation loss')
        plt.legend(loc='upper right')
        plt.show()
```

The preceding code generates the following loss per iteration for the CNN validation:

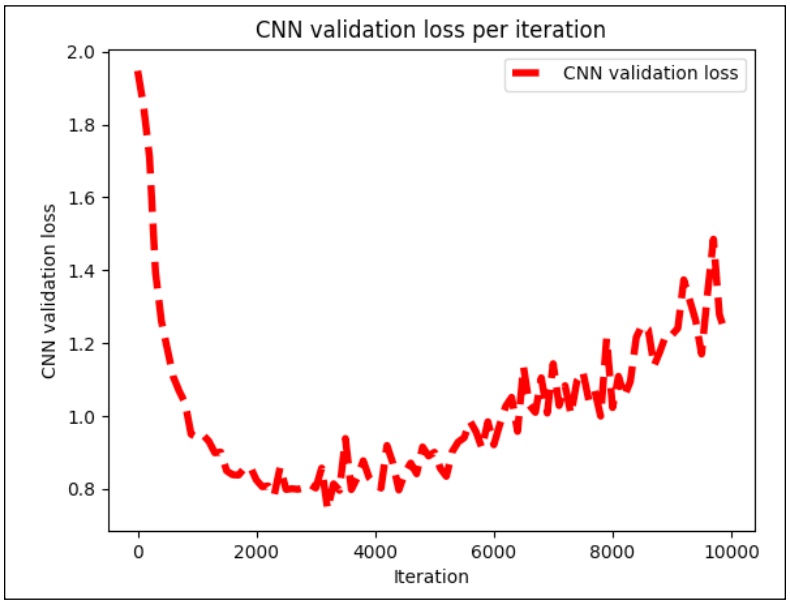


Figure 11: CNN validation loss per iteration for emotion prediction

Finally, we invoke the main method as follows:

```
if __name__ == "__main__":
    tf.app.run()
```

However, the model can be improved by acting on hyperparameters or changing its architecture. In the next section, we will see how to effectively test the model on your own images.

Testing the model on your own image

The dataset we use is standardized. All faces are exactly pointed at the camera and the emotional expressions are exaggerated and even comical in some situations. Let's see now what happens if we use a more natural image.

Make sure that there is no text overlaid on the face, the emotion is recognizable, and the face is pointed mostly at the camera. I will try to evaluate our model's prediction power with four images. The first one is my all-time favorite Bollywood actress Madhuri Dixit, the second one is Jennifer Lopez, the third one is Macaulay Culkin (from Home Alone, 1990) and the last one is Hugh Jackman from Hollywood:



Figure 12: The input images for our test (happy, neutral, fear, and angry)

We actually need to feed the network with the corresponding grayscale images. Using the numpy Python library, we convert the input color image in a valid input for the network, that is a grayscale image. Note that the number of color channels for the images is `3` but for the grayscale image it should be `1`. For example:

```
img = mpimg.imread('madhuri_dixit.jpg')
gray = rgb2gray(img)
```

The conversion function is:

```
def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])
```

At first, we must define a TensorFlow running session:

```
sess = tf.InteractiveSession()
```

Then we can recall the previously saved model:

```
new_saver = tf.train.import_meta_graph('Logs/CNN_logs/model.ckpt-
27500.meta')
new_saver.restore(sess, 'Logs/CNN_logs/model.ckpt-27500')
tf.get_default_graph().as_graph_def()
x = sess.graph.get_tensor_by_name("input:0")
y_conv = sess.graph.get_tensor_by_name("output:0")
```

To test an image, we must reshape it into a `48×48×1` valid format for the network that we constructed in the preceding section:

```
image_0 = np.resize(gray,(1,48,48,1))
```

We evaluated the same picture several times (`10,000`) in order to build a percentage of possible emotional stretches of the input images:

```
tResult = testResult()
num_evaluations = 10000
for i in range(0,num_evaluations):
    result = sess.run(y_conv, feed_dict={x:image_test})
    label = sess.run(tf.argmax(result, 1))
    label = label[0]
    label = int(label)
    tResult.evaluate(label)
tResult.display_result(num_evaluations)
```

In the preceding code segment, there are some utility functions in the `testResult` class in `Utility.py`. The construction that initializes all the emotions is as follows:

```
    def __init__(self):
        self.anger = 0
        self.disgust = 0
        self.fear = 0
        self.happy = 0
        self.sad = 0
        self.surprise = 0
        self.neutral = 0
```

We evaluate the emotion prediction using the `evaluate()` function that can be seen as follows:

```
    def evaluate(self,label):
        if (0 == label):
            self.anger = self.anger+1
        if (1 == label):
            self.disgust = self.disgust+1
        if (2 == label):
            self.fear = self.fear+1
        if (3 == label):
            self.happy = self.happy+1
        if (4 == label):
            self.sad = self.sad+1
        if (5 == label):
            self.surprise = self.surprise+1
        if (6 == label):
            self.neutral = self.neutral+1
```

Finally, the `display_result()` function that prints the result is given as follows:

```
    def display_result(self,evaluations):
        print("anger = "    + str((self.anger/float(evaluations))*100)
+ "%")
        print("disgust = "  + str((self.disgust/
float(evaluations))*100)  + "%")
        print("fear = "     + str((self.fear/float(evaluations))*100)
+ "%")
        print("happy = "    + str((self.happy/float(evaluations))*100)
+ "%")
        print("sad = "      + str((self.sad/float(evaluations))*100)
+ "%")
        print("surprise = " + str((self.surprise/
float(evaluations))*100) + "%")
        print("neutral = "  + str((self.neutral/
float(evaluations))*100)  + "%")
```

If everything went fine, after a few seconds, a result like this should appear:

```
# Madhuri Dixit
>>>
anger = 15.5%
disgust = 18.5%
fear = 0.4%
happy = 45.1%
sad = 0.0%
surprise = 16.6%
neutral = 3.9%
>>>
```

So, our CNN model has predicted Madhuri Dixit's emotion as `happy with the highest percentage, that is 45.1%`. Now let's see the other images. To test other images, just change the image name and repeated the preceding steps, that is something like as follows:

```
img = mpimg.imread(JLO.jpg')
img = mpimg.imread(macaulay_culkin.jpg')
img = mpimg.imread(hugh_jackman.jpg')

# JLO
>>>
anger = 6.84%
disgust = 0.15%
fear = 70.26%
happy = 1.02%
```

```
sad = 0.02%
surprise = 21.15%
neutral = 0.5599999999999999%
>>>

# Fear
>>>
anger = 0.0%
disgust = 0.0%
fear = 7.6%
happy = 14.299999999999999%
sad = 0.1%
surprise = 76.8%
neutral = 1.2%
>>>

# Angry
anger = 5.1%
disgust = 0.0%
fear = 93.30000000000001%
happy = 0.6%
sad = 0.2%
surprise = 0.3%
neutral = 0.5%
>>>
```

So our CNN model has predicted fear as surprise, angry as fear, and sad as fear. Of course, this doesn't mean that our model is performing very badly. Possible improvements can result from a greater and more diverse training set, or intervening on the network's parameters, or modifying the network architecture.

Later on, I iterated the model up to 30,000 times and found the following result (I tried to iterate it 100,000 times but did not have that much time to be frank, but you can try of course):

```
#Madhuri
>>>
anger = 15.65%
disgust = 16.99%
fear = 0.37%
happy = 45.97%
sad = 0.0%
surprise = 16.39%
neutral = 4.63%
>>>
```

```
# JLO
>>>
anger = 6.84%
disgust = 0.15%
fear = 0.02%
happy = 1.02%
sad = 70.26%
surprise = 21.15%
neutral = 0.5599999999999999%
>>>

# Fear
>>>
anger = 0.0%
disgust = 0.05%
fear = 7.84%
happy = 13.83%
sad = 0.25%
surprise = 76.36%
neutral = 1.67%
>>>

# Angry
>>>
anger = 2.4%
disgust = 0.8999999999999999%
fear = 0.3%
happy = 8.3%
sad = 0.2%
surprise = 75.2%
neutral = 12.7%
>>>
```

In short, after 30,000 iterations our CNN model predicted our four t test images:



Figure 13: CNN emotion prediction

Now we know how to build a CNN classifier that classifies emotion based on the grayscale image features. Although we did not observe more than 50% accuracy, we can still use the raw image directly. In the next section, we will see how to classify animals based on their raw images.

Using complex CNN for predictive analytics

In the next section, we will see how to classify and distinguish dogs from cats based on their raw images. We will also see how to implement more complex CNN models to deal with the raw and color image having three channels whereas in the previous example we have dealt with the grayscale image having just 1 channel. However, first we will look at a short description.

Dataset description

For this example, we'll use the dog versus cat dataset from Kaggle that was provided for the infamous dogs versus cats classification problem as a playground competition with kernels enabled. The dataset can be downloaded from: `https://www.kaggle.com/c/dogs-vs-cats-redux-kernels-edition/data`:



Figure 14: Dog versus cat (I'm kidding! They're not fighting, but we will be classifying their images using CNN)

The train folder contains 25,000 images of dogs and cats. Each image in this folder has the label as part of the filename. The test folder contains 12,500 images, named according to a numeric ID. For each image in the test set, you should predict a probability that the image is a dog (1 = dog, 0 = cat), that is a binary classification problem.

CNN predictive model for image classification

Now we start developing our predictive model. For this example, there are three Python scripts:

1. Load required packages.

   Here we import required packages and libraries. Note that depending upon the platform, your imports might be different:

   ```
   import time
   import math
   import random
   import os
   import pandas as pd
   import numpy as np
   import matplotlib.pyplot as plt
   import tensorflow as tf
   import Preprocessor
   import cv2
   import LayersConstructor
   from sklearn.metrics import confusion_matrix
   from datetime import timedelta
   from tensorflow.python.framework import ops
   from sklearn.metrics.classification import accuracy_score
   from sklearn.metrics import precision_recall_fscore_support
   import warnings
   ```

   We could disable the warning as well. To do so use the following lines in your Python Script:

   ```
   warnings.filterwarnings("ignore")
   os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
   ops.reset_default_graph()
   ```

2. Load the training/test images to generate a train/test set.

   We set the number of color channels to `3` for the images. In the previous section, we have seen that it should be 1 for the grayscale images:

   ```
   num_channels = 3
   ```

   For simplicity, we assume the image dimensions to be only squares only. Let set the size to be `128`:

   ```
   img_size = 128
   ```

Now that we have the image size (that is `128` and number of channels, that is `3`), the size of image when flattened to a single dimension would be the multiplication of the image dimension and the number of channels as follows:

```
img_size_flat = img_size * img_size * num_channels
```

Note that at a later stage, we might need to reshape the image for the max pooling and convolutional layers, so we need to reshape the image. For our case, it would be the tuple with height and width of images used to reshape arrays:

```
img_shape = (img_size, img_size)
```

We should have explicitly defined the labels, that is classes, since we only have the raw color image so the images do not have the labels like other numeric machine learning datasets. Let's explicitly define the class info as follows:

```
classes = ['dogs', 'cats']
num_classes = len(classes)
```

We need to define the batch size that needs to be trained on our CNN model later on:

```
batch_size = 14
```

Now, we can also define what portion of the training set will be used as the validation split. Let's assume that 16% will be used for simplicity:

```
validation_size = 0.16
```

One important thing is to set how long to wait after the validation loss stops improving before terminating the training. We should use none if you don't want to implement early stopping:

```
early_stopping = None
```

Now, download the dataset and manually separate the images of dogs and cats and place them in two separate folders. To be more specific, suppose you put your training set under the path `/home/DoG_CaT/data/train/`, in the `train` folder create two separate folders, `dogs` and `cats` but only show the path up to `DoG_CaT/data/train/`.

We also assume that our test set is in the `/home/DoG_CaT/data/test/ directory`. Also, you can define the check point directory where the logs and model check point files will be written:

```
 train_path = '/home/DoG_CaT/data/train/'
test_path = '/home/DoG_CaT/data/test/'
checkpoint_dir = "models/"
```

Then we start reading the training set and suitably prepare them for the CNN model. For processing the test and train set, we have another script, `Preprocessor.py`. But it would be better to prepare the test set as well:

```
data = Preprocessor.read_train_sets(train_path, img_size, classes,
validation_size=validation_size)
```

The preceding line of code reads the raw images of cats and dogs and creates the training set. The `read_train_sets()` function goes as follows:

```
def read_train_sets(train_path, image_size, classes, validation_
size=0):
  class DataSets(object):
    pass
  data_sets = DataSets()
  images, labels, ids, cls = load_train(train_path, image_size,
classes)
  images, labels, ids, cls = shuffle(images, labels, ids, cls)  #
shuffle the data
  if isinstance(validation_size, float):
    validation_size = int(validation_size * images.shape[0])
  validation_images = images[:validation_size]
  validation_labels = labels[:validation_size]
  validation_ids = ids[:validation_size]
  validation_cls = cls[:validation_size]
  train_images = images[validation_size:]
  train_labels = labels[validation_size:]
  train_ids = ids[validation_size:]
  train_cls = cls[validation_size:]
  data_sets.train = DataSet(train_images, train_labels, train_ids,
train_cls)
  data_sets.valid = DataSet(validation_images, validation_labels,
validation_ids, validation_cls)
  return data_sets
```

In the preceding code segment, we have used the method `load_train()` to load the images which is an instance of the class called Dataset:

```
def load_train(train_path, image_size, classes):
    images = []
    labels = []
    ids = []
    cls = []

    print('Reading training images')
    for fld in classes:
        index = classes.index(fld)
```

```
        print('Loading {} files (Index: {})'.format(fld, index))
        path = os.path.join(train_path, fld, '*g')
        files = glob.glob(path)
        for fl in files:
            image = cv2.imread(fl)
            image = cv2.resize(image, (image_size, image_size),
cv2.INTER_LINEAR)
            images.append(image)
            label = np.zeros(len(classes))
            label[index] = 1.0
            labels.append(label)
            flbase = os.path.basename(fl)
            ids.append(flbase)
            cls.append(fld)
    images = np.array(images)
    labels = np.array(labels)
    ids = np.array(ids)
    cls = np.array(cls)
    return images, labels, ids, cls
```

The Dataset class which is used to generate the batches of training set can be seen as follows:

```
class DataSet(object):
  def __init__(self, images, labels, ids, cls):
    self._num_examples = images.shape[0]
    images = images.astype(np.float32)
    images = np.multiply(images, 1.0 / 255.0)

    self._images = images
    self._labels = labels
    self._ids = ids
    self._cls = cls
    self._epochs_completed = 0
    self._index_in_epoch = 0

  @property
  def images(self):
    return self._images

  @property
  def labels(self):
    return self._labels

  @property
```

```
  def ids(self):
    return self._ids

  @property
  def cls(self):
    return self._cls

  @property
  def num_examples(self):
    return self._num_examples

  @property
  def epochs_completed(self):
    return self._epochs_completed

  def next_batch(self, batch_size):
    """Return the next `batch_size` examples from this data
set."""
    start = self._index_in_epoch
    self._index_in_epoch += batch_size
    if self._index_in_epoch > self._num_examples:
      # Finished epoch
      self._epochs_completed += 1
      start = 0
      self._index_in_epoch = batch_size
      assert batch_size <= self._num_examples
    end = self._index_in_epoch
    return self._images[start:end], self._labels[start:end],
self._ids[start:end], self._cls[start:end]
```

Then similarly, we prepare the test set from the test images that are a mixture of dog and cat images.

```
test_images, test_ids = Preprocessor.read_test_set(test_path, img_
size)
```

We have the `read_test_set()` function for ease, which goes as follows:

```
def read_test_set(test_path, image_size):
  images, ids  = load_test(test_path, image_size)
  return images, ids
```

Now similar to the training set, we have a dedicated function `load_test ()` for loading the test set that goes as follows:

```
def load_test(test_path, image_size):
  path = os.path.join(test_path, '*g')
  files = sorted(glob.glob(path))

  X_test = []
  X_test_id = []
  print("Reading test images")
  for fl in files:
      flbase = os.path.basename(fl)
      img = cv2.imread(fl)
      img = cv2.resize(img, (image_size, image_size), cv2.INTER_
LINEAR)
      X_test.append(img)
      X_test_id.append(flbase)
  X_test = np.array(X_test, dtype=np.uint8)
  X_test = X_test.astype('float32')
  X_test = X_test / 255
  return X_test, X_test_id
```

Well done! We can now see some randomly selected images. For this, we have the helper function called `plot_images()` that creates a figure with 3x3 sub-plots so altogether nine images will be plotted along with their true label. It goes as follows:

```
def plot_images(images, cls_true, cls_pred=None):
    if len(images) == 0:
        print("no images to show")
        return
    else:
        random_indices = random.sample(range(len(images)),
min(len(images), 9))
        images, cls_true  = zip(*[(images[i], cls_true[i]) for i
in random_indices])
    fig, axes = plt.subplots(3, 3)
    fig.subplots_adjust(hspace=0.3, wspace=0.3)
    for i, ax in enumerate(axes.flat):
        # Plot image.
        ax.imshow(images[i].reshape(img_size, img_size, num_
channels))
        if cls_pred is None:
            xlabel = "True: {0}".format(cls_true[i])
        else:
```

```
            xlabel = "True: {0}, Pred: {1}".format(cls_true[i],
cls_pred[i])
        ax.set_xlabel(xlabel)
        ax.set_xticks([])
        ax.set_yticks([])
    plt.show()
```

Let's get some random images and their labels from the train set:

```
images, cls_true  = data.train.images, data.train.cls
```

Finally, we plot the images and labels using our `helper-function` above:

```
plot_images(images=images, cls_true=cls_true)
```

The preceding line of code generates the true labels of the images that are randomly selected:



Figure 15: Showing the true labels of the images that are randomly selected

Finally, we can print the dataset statistics:

```
print("Size of:")
print("  - Training-set:\t\t{}".format(len(data.train.labels)))
print("  - Test-set:\t\t{}".format(len(test_images)))
print("  - Validation-set:\t{}".format(len(data.valid.labels)))
>>>
Reading training images
Loading dogs files (Index: 0)
```

```
Loading cats files (Index: 1)
Reading test images
Size of:
- Training-set: 21000
- Test-set: 12500
- Validation-set:    4000
```

3. Define CNN hyperparameters.

   Now that we have the training and test set. It's time to define the hyper parameters for the CNN model before we start constructing. In the first and the second convolutional layers we define the width and height of each filter, that is. `3` where number of filters is `32`:

   ```
   filter_size1 = 3
   num_filters1 = 32
   filter_size2 = 3
   num_filters2 = 32
   ```

   The third convolutional layer has equal dimensions but twice the filters, that is. `64` filters: `filter_size3 = 3:`

   ```
   num_filters3 = 64
   ```

   The last two layers are fully-connected layers specifying the number of neurons:

   ```
   fc_size = 128
   ```

   Now, let's make the training slower for more intensive training by setting a lower value of learning rate as follows:

   ```
   learning_rate=1e-4
   ```

4. Construct the CNN layers.

   Once we have defined the CNN hyperparameters, the next task is to implement the CNN network. As you can guess, for our task, we will construct a CNN network having three convolutional layers, a flatten layer and two fully connected layers (refer to `LayersConstructor.py`). Moreover, we need to define the weight and the bias as well. Furthermore, we will have implicit max-pooling layers too. At first, let's define the weight. In the following, we have the `new_weights()` method that asks for the image shape and returns the truncated normal shapes:

   ```
   def new_weights(shape):
       return tf.Variable(tf.truncated_normal(shape, stddev=0.05))
   ```

Then we define the biases using the `new_biases()` method:

```
def new_biases(length):
    return tf.Variable(tf.constant(0.05, shape=[length]))
```

Now let's define a method `new_conv_layer()` for constructing a convolutional layer. The method takes the input batch, number of input channels, filter size, number of filters and it also uses the max pooling (if true, we use a 2x2 max-pooling) to construct a new convolutional layer. The workflow of the method is as follows:

1. Define the shape of the filter-weights for the convolution which is determined by the TensorFlow API.
2. Create the new weights (that is, filters) with the given shape and new biases, one for each filter.
3. Create the TensorFlow operation for convolution where the strides are set to `1` in all dimensions. The first and last stride must always be `1`, because the first is for the image-number and the last is for the input-channel. For example, `strides=[1, 2, 2, 1]` would mean that the filter is moved 2 pixels across the x- and y-axis of the image. The padding is set to `SAME` which means the input image is padded with zeroes so the size of the output is the same.
4. Add the biases to the results of the convolution. Then a bias-value is added to each filter-channel.
5. It then uses the pooling to down-sample the image resolution. This is 2x2 max-pooling, which means that we consider 2x2 windows and select the largest value in each window. Then we move 2 pixels to the next window.
6. ReLU is then used to calculate the `max(x, 0)` for each input pixel `x`. This adds some non-linearity to the formula and allows us to learn more complicated functions. It should be noted that an ReLU is normally executed before the pooling, but since relu(`max_pool(x)) == max_pool(relu(x)`) we can save 75% of the relu-operations by max-pooling first.
7. Finally, it returns both the resulting layer and the filter-weights because we will plot the weights later.

Now we define a function to construct a convolutional layer:

```
def new_conv_layer(input,  num_input_channels, filter_size, num_
filters,                    use_pooling=True):
    shape = [filter_size, filter_size, num_input_channels, num_
filters]
    weights = new_weights(shape=shape)
    biases = new_biases(length=num_filters)
    layer = tf.nn.conv2d(input=input,
                         filter=weights,
                         strides=[1, 1, 1, 1],
                         padding='SAME')
    layer += biases
    if use_pooling:
        layer = tf.nn.max_pool(value=layer,
                               ksize=[1, 2, 2, 1],
                               strides=[1, 2, 2, 1],
                               padding='SAME')
    layer = tf.nn.relu(layer)
    return layer, weights
```

The next task is to define the flatten layer that does the following:

1. Gets the shape of the input layer.
2. The number of features is: `img_height * img_width * num_channels`. The `get_shape()` function TensorFlow is used to calculate this.
3. It will then reshape the layer to [`num_images`, `num_features`]. We just set the size of the second dimension to `num_features` and the size of the first dimension to `-1` which means the size in that dimension is calculated so the total size of the tensor is unchanged from the reshaping.
4. Finally, it returns both the flattened layer and the number of features.

The following code does exactly the same as we described earlier:

```
def flatten_layer(layer):
    layer_shape = layer.get_shape()
    num_features = layer_shape[1:4].num_elements()
    layer_flat = tf.reshape(layer, [-1, num_features])
    return layer_flat, num_features
```

Finally, we need to construct the fully connected layers. The following function `new_fc_layer()` takes the input batches, number of batches, number of output, that is predicted classes, and it uses the ReLU.

It then creates the weights and biases based on the methods we defined earlier in this step. Finally, it calculates the layer as the matrix multiplication of the input and weights, and then adds the bias-values:

```
def new_fc_layer(input, num_inputs, num_outputs, use_relu=True):
    weights = new_weights(shape=[num_inputs, num_outputs])
    biases = new_biases(length=num_outputs)
    layer = tf.matmul(input, weights) + biases
    if use_relu:
        layer = tf.nn.relu(layer)
    return layer
```

5. Prepare the TensorFlow graph.

   We now create the placeholders for the TensorFlow graph:

```
x = tf.placeholder(tf.float32, shape=[None, img_size_flat],
name='x')
x_image = tf.reshape(x, [-1, img_size, img_size, num_channels])
y_true = tf.placeholder(tf.float32, shape=[None, num_classes],
name='y_true')
y_true_cls = tf.argmax(y_true, dimension=1)
```

6. Create the CNN model.

   Now that we have the input, that is `x_image` it is ready to feed to the convolutional layer, we formally create a convolutional layer as follows followed by the max pooling:

```
layer_conv1, weights_conv1 = \
    LayersConstructor.new_conv_layer(input=x_image,
                    num_input_channels=num_channels,
                    filter_size=filter_size1,
                    num_filters=num_filters1,
                    use_pooling=True)
```

   We have to have the second convolutional layer where the input is the first convolutional layer, that is `layer_conv1` followed by the max pooling:

```
layer_conv2, weights_conv2 = \
    LayersConstructor.new_conv_layer(input=layer_conv1,
                    num_input_channels=num_filters1,
                    filter_size=filter_size2,
                    num_filters=num_filters2,
                    use_pooling=True)
```

We now have the third convolutional layer where the input is the output of the second convolutional layer, that is `layer_conv2` followed by the max pooling:

```
layer_conv3, weights_conv3 = \
    LayersConstructor.new_conv_layer(input=layer_conv2,
                    num_input_channels=num_filters2,
                    filter_size=filter_size3,
                    num_filters=num_filters3,
                    use_pooling=True)
```

Once the third convolutional layer is instantiated, we then instantiate the flatten layer as follows:

```
layer_flat, num_features = LayersConstructor.flatten_layer(layer_
conv3)
```

Once we have flattened the images, they are ready to be fed to the first fully connected layer. We use the ReLU:

```
layer_fc1 = LayersConstructor.new_fc_layer(input=layer_flat,
                        num_inputs=num_features,
                        num_outputs=fc_size,
                        use_relu=True)
```

Finally, we have to have the 2nd and the final fully connected layer where the input is the output of the first fully connected layer:

```
layer_fc2 = LayersConstructor.new_fc_layer(input=layer_fc1,
                        num_inputs=fc_size,
                        num_outputs=num_classes,
                        use_relu=False)
```

7. Run a TensorFlow graph to train the CNN model.

   The following steps are used to perform the training. The code is self-explanatory; we used it in our previous examples already.

   We use the softmax to predict the classes by comparing with the true classes:

```
y_pred = tf.nn.softmax(layer_fc2)
y_pred_cls = tf.argmax(y_pred, dimension=1)
cross_entropy = tf.nn.softmax_cross_
entropy_with_logits(logits=layer_fc2,
labels=y_true)
```

We define the cost function and then the optimizer (Adam optimizer in this case). Then we compute the accuracy:

```
cost _op= tf.reduce_mean(cross_entropy)
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).
minimize(cost_op)
correct_prediction = tf.equal(y_pred_cls, y_true_cls)
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

Then we initialize all the variables using the `global_variables_initializer()` function from TensorFlow:

```
init_op = tf.global_variables_initializer()
Then we create and run the TensorFlow session to carry the
training across the tensors:
session = tf.Session()
session.run(init_op)
```

We feed out training data such that the batch size is `32` (see step 2):

```
train_batch_size = batch_size
```

We maintain two lists to track the training and validation accuracy:

```
acc_list = []
val_acc_list = []
```

We then count the total number of iterations performed so far and create an empty list to keep track of all the iterations:

```
total_iterations = 0
iter_list = []
```

We formally start the training by invoking the `optimize()` function that takes the number of iterations. It needs two: `x_batch` of training examples that holds a batch of images and `y_true_batch` are the true labels for those images. It then converts the shape of each image from [num examples, rows, columns, depth] to [num examples, flattened image shape]. After that, we put the batch into a dict for placeholder variables in the TensorFlow graph.

Later on, we run the optimizer on the batch of training data. Then, TensorFlow assigns the variables in `feed_dict_train` to the placeholder variables. The optimizer is then executed to print the status at the end of each epoch. Finally, it updates the total number of iterations that we performed:

```
def optimize(num_iterations):
    global total_iterations
    best_val_loss = float("inf")
    patience = 0
```

```
    for i in range(total_iterations, total_iterations + num_
iterations):
        x_batch, y_true_batch, _, cls_batch = data.train.next_
batch(train_batch_size)
        x_valid_batch, y_valid_batch, _, valid_cls_batch = data.
valid.next_batch(train_batch_size)
        x_batch = x_batch.reshape(train_batch_size, img_size_flat)
        x_valid_batch = x_valid_batch.reshape(train_batch_size,
img_size_flat)
        feed_dict_train = {x: x_batch, y_true: y_true_batch}
        feed_dict_validate = {x: x_valid_batch, y_true: y_valid_
batch}
        session.run(optimizer, feed_dict=feed_dict_train)

        if i % int(data.train.num_examples/batch_size) == 0:
            val_loss = session.run(cost, feed_dict=feed_dict_
validate)
            epoch = int(i / int(data.train.num_examples/batch_
size))
            acc, val_acc = print_progress(epoch, feed_dict_train,
feed_dict_validate, val_loss)
            acc_list.append(acc)
            val_acc_list.append(val_acc)
            iter_list.append(epoch+1)

            if early_stopping:
                if val_loss < best_val_loss:
                    best_val_loss = val_loss
                    patience = 0
                else:
                    patience += 1
                if patience == early_stopping:
                    break
    total_iterations += num_iterations
```

We will show how our training went in the next section.

8. Model evaluation.

   We have managed to finish the training. It's time to evaluate the model. Before, we start the evaluation formally, we need some auxiliary functions for plotting the example errors, plotting the confusion function, and printing the validation accuracy.

The `plot_example_errors()` takes two parameters, `cls_pred` which is an array of the predicted class-number for all images in the test-set. The second parameter, `correct`, is a boolean array and checks whether the predicted class is equal to the true class for each image in the test-set. At first, it gets the images from the test-set that have been incorrectly classified, then it gets the predicted and true classes for those images, and finally it plots the first nine images with their classes (that is predicted versus true labels):

```
def plot_example_errors(cls_pred, correct):
    incorrect = (correct == False)
    images = data.valid.images[incorrect]
    cls_pred = cls_pred[incorrect]
    cls_true = data.valid.cls[incorrect]
    plot_images(images=images[0:9], cls_true=cls_true[0:9], cls_
pred=cls_pred[0:9])
```

Then, we have the second auxiliary method called `plot_confusion_matrix()` that plots the confusion matrix. It utilizes the `confusion_matrix()` method from sklearn to compute the confusion matrix. This method is also used to print other performance metrics such as precision, recall, and f1 score using the `precision_recall_fscore_support()` function from sklearn:

```
def plot_confusion_matrix(cls_pred):
    cls_true = data.valid.cls
    cm = confusion_matrix(y_true=cls_true, y_pred=cls_pred)
    p, r, f, s = precision_recall_fscore_support(cls_true, cls_
pred, average='weighted')
    print('Precision:', p)
    print('Recall:', r)
    print('F1-score:', f)
    plt.matshow(cm)
    plt.colorbar()
    tick_marks = np.arange(num_classes)
    plt.xticks(tick_marks, range(num_classes))
    plt.yticks(tick_marks, range(num_classes))
    plt.xlabel('Predicted')
    plt.ylabel('True')
    plt.show()
```

The third auxiliary function is called `print_validation_accuracy()` that prints the validation accuracy. It allocates an array for the predicted classes which will be calculated in batches and added into this array, then it calculates the predicted classes for the batches:

```
def print_validation_accuracy(show_example_errors=False, show_
confusion_matrix=False):
    num_test = len(data.valid.images)
    cls_pred = np.zeros(shape=num_test, dtype=np.int)
    i = 0
    while i < num_test:
        # The ending index for the next batch is denoted j.
        j = min(i + batch_size, num_test)
        images = data.valid.images[i:j, :].reshape(batch_size,
img_size_flat)
        labels = data.valid.labels[i:j, :]
        feed_dict = {x: images, y_true: labels}
        cls_pred[i:j] = session.run(y_pred_cls, feed_dict=feed_
dict)
        i = j

    cls_true = np.array(data.valid.cls)
    cls_pred = np.array([classes[x] for x in cls_pred])
    correct = (cls_true == cls_pred)
    correct_sum = correct.sum()
    acc = float(correct_sum) / num_test

    msg = "Accuracy on Test-Set: {0:.1%} ({1} / {2})"
    print(msg.format(acc, correct_sum, num_test))

    if show_example_errors:
        print("Example errors:")
        plot_example_errors(cls_pred=cls_pred, correct=correct)
    if show_confusion_matrix:
        print("Confusion Matrix:")
        plot_confusion_matrix(cls_pred=cls_pred)
```

Now that we have our auxiliary functions, we now can start the optimization. First, let's iterate the fine-tuning `10000` times and see the performance:

```
optimize(num_iterations=1000)
```

We now plot the training and validation loss over time:

```
plt.plot(iter_list, acc_list, 'r--', label='CNN training accuracy
per iteration', linewidth=4)
plt.title('CNN training accuracy per iteration')
plt.xlabel('Iteration')
plt.ylabel('CNN training accuracy')
plt.legend(loc='upper right')
plt.show()
```

The preceding code prints the CNN training accuracy per epoch:

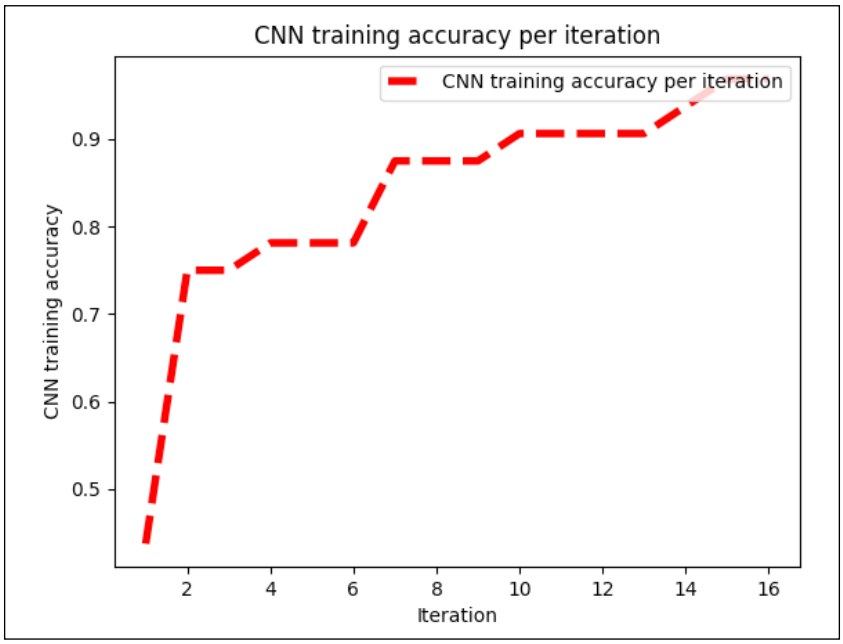


Figure 16: CNN training accuracy per epoch

Now let's plot validation loss over time:

```
plt.plot(iter_list, val_acc_list, 'r--', label='CNN validation
accuracy per iteration', linewidth=4)
plt.title('CNN validation accuracy per iteration')
plt.xlabel('Iteration')
plt.ylabel('CNN validation accuracy')
plt.legend(loc='upper right')
plt.show()
```

The preceding codes prints the CNN validation accuracy per epoch:

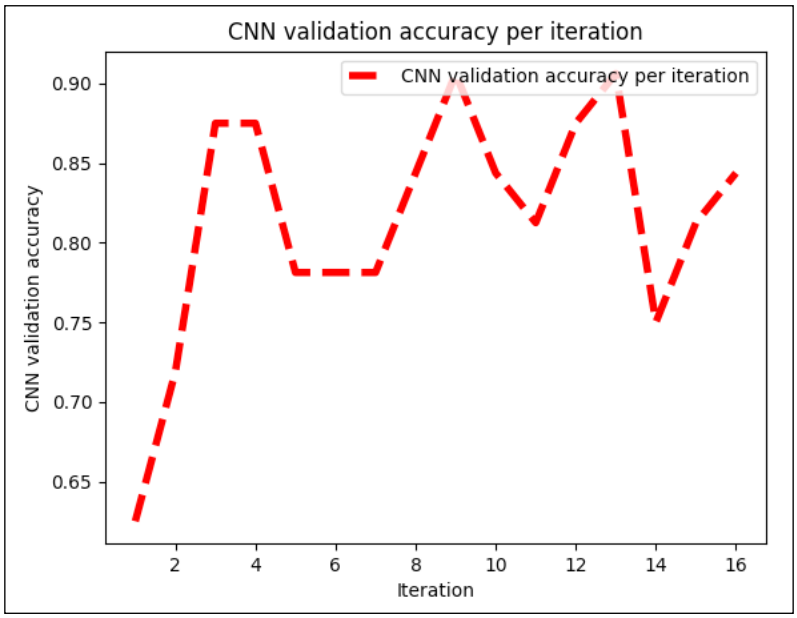


Figure 16: CNN validation accuracy per epoch

After 10,000 iterations, we observed the following result:

```
Accuracy on Test-Set: 78.8% (3150 / 4000)
Precision: 0.793378626929
Recall: 0.7875
F1-score: 0.786639298213
```

Which means the accuracy on the test set is about 79%. Also, let's see how well our classifier is perms for sample images:



Figure 17: Random prediction on the test set (after 10,000 iterations)

Our corresponding confusion matrix can be seen as follows:

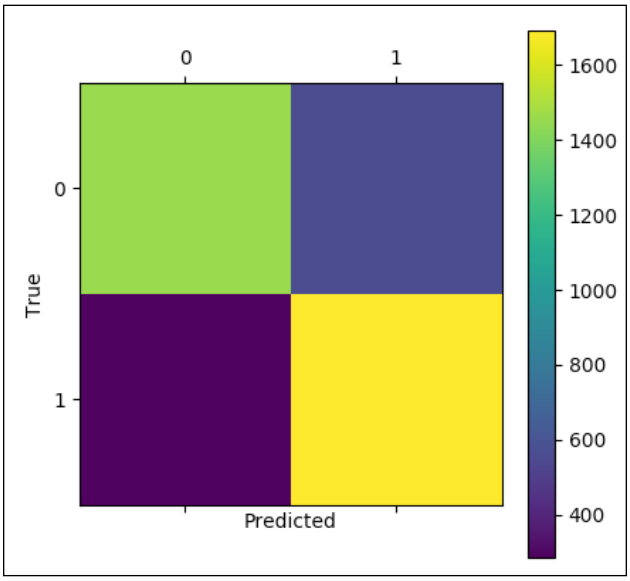


Figure 18: Confusion matrix after 10,000 iterations showing predicted versus true class

After that, we further iterate the optimization up to 100,000 times and observe better accuracy:



Figure 19: Random prediction on the test set (after 100,000 iterations)

```
>>>
Accuracy on Test-Set: 81.1% (3244 / 4000)
Precision: 0.811057239265
Recall: 0.811
F1-score: 0.81098298755
```

So, it did not improve that much, only a 2% increase on the overall accuracy. Here's the corresponding confusion matrix generated:

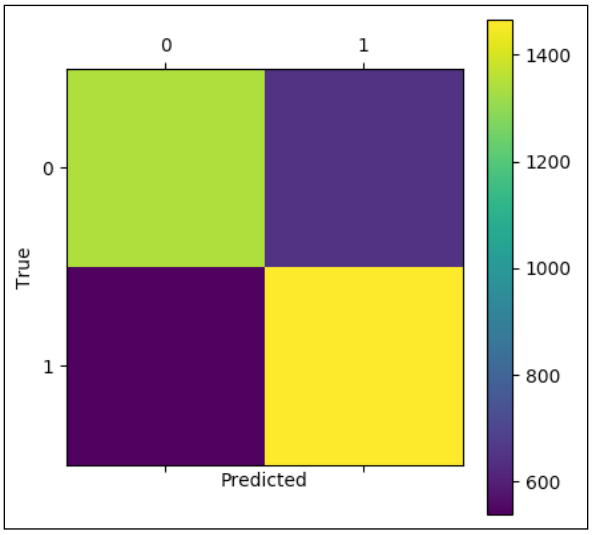


Figure 20: Confusion matrix after 100,000 iterations showing predicted versus true class

Now it's time to evaluate our model for a single image. For simplicity, we will take two random images of a dog and a cat and will see the prediction power of our model:



Figure 21: Example image for the cat and dog to be classified

At first, we load these two images and prepare the test set accordingly. We have seen this in an earlier step in this example:

```
test_cat = cv2.imread('Test_image/cat.jpg')
test_cat = cv2.resize(test_cat, (img_size, img_size), cv2.INTER_
LINEAR) / 255
preview_cat = plt.imshow(test_cat.reshape(img_size, img_size, num_
channels))

test_dog = cv2.imread('Test_image/dog.jpg')
test_dog = cv2.resize(test_dog, (img_size, img_size), cv2.INTER_
LINEAR) / 255
preview_dog = plt.imshow(test_dog.reshape(img_size, img_size, num_
channels))
```

Then we have the following function for making the prediction:

```
def sample_prediction(test_im):
    feed_dict_test = {
        x: test_im.reshape(1, img_size_flat),
        y_true: np.array([[1, 0]])
    }
    test_pred = session.run(y_pred_cls, feed_dict=feed_dict_test)
    return classes[test_pred[0]]

print("Predicted class for test_cat: {}".format(sample_
prediction(test_cat)))
print("Predicted class for test_dog: {}".format(sample_
prediction(test_dog)))

>>>
Predicted class for test_cat: cats
Predicted class for test_dog: dogs
```

Finally, when we're done, we close the TensorFlow session by invoking the `close()` method:

```
session.close()
```

Now that we have come to the end of this chapter, one thing I need to confess is that in the last example, we did not show any steps to visualize the progress on the TensorBoard. Secondly, there is no option for saving the model so that we can reuse it later on by simply resorting. I leave it up to the readers to add such functionality. One clue is to just try to follow the second example in this chapter and you're done.

Summary

In this chapter, we have discussed how to use CNNs which are a type of feedforward artificial neural network in which the connectivity pattern between its neurons is inspired by the organization of the animal visual cortex. Individual cortical neurons respond to stimuli in a restricted region of space known as the receptive field. In the last few years, CNNs have managed to achieve and demonstrate superhuman performance on some complex visual tasks: image search services, self-driving cars, automatic video classification systems, and more.

Moreover, CNNs are not restricted to visual perception: they are also successful at other tasks, such as voice recognition and NLP. We have seen how to develop predictive analytics applications such as emotion recognition, image classification, and text classification using the convolutional neural network on real image/text datasets.

Out text prediction pipeline using CNN can predict the product and movie review with an accuracy of 98%. Then, we built a CNN to classify emotions starting from a dataset of images; we tested the network on a single image and we evaluated the limits and the performance of our model. Finally, we have seen how to increase the model complexity by adding more hidden layers and raw image files rather than the textual representation of the images that shows up to 81% prediction accuracy.

An RNN is a class of artificial neural network where connections between units form a directed cycle. This creates an internal state of the network which allows it to exhibit dynamic temporal behavior.

In *Chapter 9*, *Using Recurrent Neural Networks for Predictive Analytics*, we will develop several real-life predictive models for making the predictive analytics easier using RNN and its different architectural variants.

9
Using Recurrent Neural Networks for Predictive Analytics

A **recurrent neural network** (**RNN**) is a class of artificial neural network where connections between units form a directed cycle. RNNs make use of information from the past. That way, they can make predictions in data with high temporal dependencies. This creates an internal state of the network that allows it to exhibit dynamic temporal behavior. In this chapter, we will develop several real-life predictive models for making the predictive analytics easier using an RNN and its different architectural variants.

First, we will provide some theoretical background to RNNs, then show a few examples that will show a step-by-step way of implementing predictive models for image classification, sentiment analysis of movies, and product spam prediction for NLP. Finally, we will show how to develop predictive models for time series data. In a nutshell, the following topics will be covered in this chapter:

- RNN architecture
- Using **bi-directional RNNs** (**BRNNs**) for image classification
- Implementing an RNN for spam prediction
- Developing a predictive model for time series data
- An LSTM predictive model for sentiment analysis

RNN architecture

In this section, we will first provide some contextual information about RNNs. Then the second part of this chapter will show the various architectures of RNNs, including bi-directional RNN, **Long Short-Term Memory** (**LSTM**), and **Gated Recurrent Unit** (**GRU**).

Contextual information and the architecture of RNNs

Human beings don't start thinking from scratch; the human mind has the so-called persistence of memory, the ability to associate the past with recent information. Traditional neural networks instead ignore past events. Taking as an example a movie scenes classifier, it's not possible for a neural network to use past scene to classify current ones.

RNNs were developed to try to solve this problem. In contrast to conventional neural networks, RNNs are networks with a loop that allows the information to be persistent. An RNN is different from a traditional neural network because it introduces a transition weight *W* to transfer information between times.

RNNs process a sequential input one at a time, updating a kind of vector state that contains information about all past elements of the sequence. The following figure shows a neural network that takes as input a value of **X(t)**, and then outputs a value **Y(t)**:

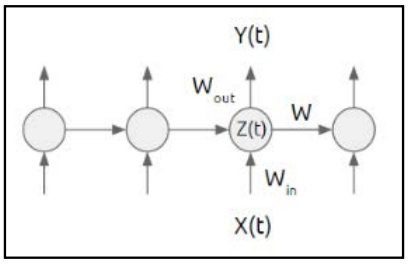


Figure 1: An RNN architecture can use the previous states of the network to its advantage

As shown in figure 1, the first half of the neural network is characterized by the function $Z(t) = X(t) * W_{in}$, and the second half of the neural network takes the form $Y(t) = Z(t) * W_{out}$. If you prefer, the whole neural network is just the function $Y(t) = (X(t) * W_{in}) * W_{out}$. At each time *t*, when calling the learned model, this architecture does not take into account knowledge about the previous runs. It's like predicting stock market trends by only looking at data from the current day. A better idea would be to exploit overarching patterns from a week's worth or months' worth of data.

A more explicit architecture can be found in figure 2, where the temporally shared weights *w2* (for the hidden layer) must be learned in addition to *w1* (for the input layer) and *w3* (for the output layer):

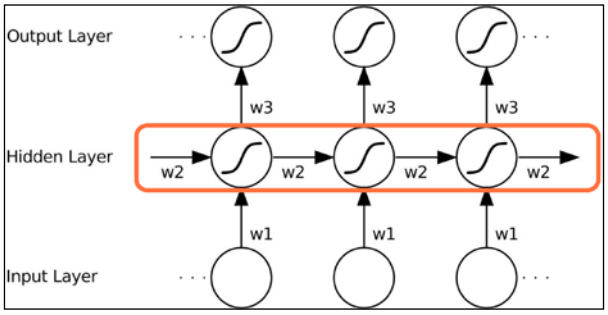


Figure 2: An RNN architecture where all the weights in all the layers have to be learned with time

Now let's see some commonly used architectures of RNNs such as LSTM, BRNN, and GRU in the next subsection.

BRNNs

BRNNs are based on the idea that the output at time *t* may depend on the previous and future elements in the sequence. To realize this, the output of two RNNs must be mixed: one executes the process in a direction and the second runs the process in the opposite direction. Figure 3 shows the basic difference between a regular RNN and a BRNN.

A more explicit BRNN architecture can be found in the following figure where the temporally shared weights **w2**, **w3**, **w4**, and **w5** (for the forward and the backward layer) must be learned in addition to **w1** (for the input layers) and **w6** (for the output layer):

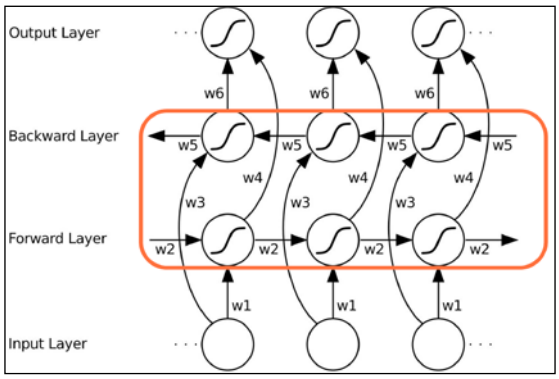


Figure 3: A BRNN architecture where all the weights in all the layers have to be learned with time

The unrolled architecture is also a very common implementation of a BRNN. The unrolled architecture of a BRNN is depicted in the following figure. The network splits neurons of a regular RNN into two directions, one for positive time direction (forward states), and another for negative time direction (backward states).With this structure, the output layer can get information from the past and future states as shown in figure 4:

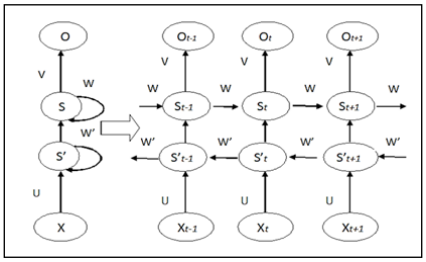


Figure 4: Unrolled BRNN

LSTM networks

One type of RNN model is called LSTM. The precise implementation details of LSTM are not in the scope of this book. An LSTM is a special RNN architecture, which was originally conceived by Hochreiter and Schmidhuber in 1997.

This type of neural network has been recently rediscovered in the context of deep learning, because it is free from the problem of vanishing gradient, and offers excellent results and performance. LSTM-based networks are ideal for prediction and classification of temporal sequences and are replacing many traditional approaches to deep learning.

It's a hilarious name, but it means exactly what it sounds like. The name signifies that short-term patterns aren't forgotten in the long-term. An LSTM network is composed of cells (LSTM blocks) linked to each other. Each LSTM block contains three types of the gate, input gate, output gate, and forget gate, respectively, that implement the functions of writing, reading, and reset on the cell memory. These gates are not binary, but analogical (generally managed by a sigmoidal activation function mapped in the range [0, 1], where 0 indicates total inhibition, and 1 shows total activation).

If you consider the LSTM cell as a black box, it can be used very much like a basic cell, except it will perform much better; training will converge faster, and it will detect long-term dependencies in the data. In TensorFlow, you can simply use a `BasicLSTMCell` instead of a `BasicRNNCell`:

```
lstm_cell = tf.contrib.rnn.BasicLSTMCell(num_units=n_neurons)
```

LSTM cells manage two state vectors, and for performance reasons, they are kept separate by default. You can change this default behavior by setting `state_is_tuple=False` when creating the `BasicLSTMCell`. So how does an LSTM cell work? The architecture of a basic LSTM cell is shown in figure 5:

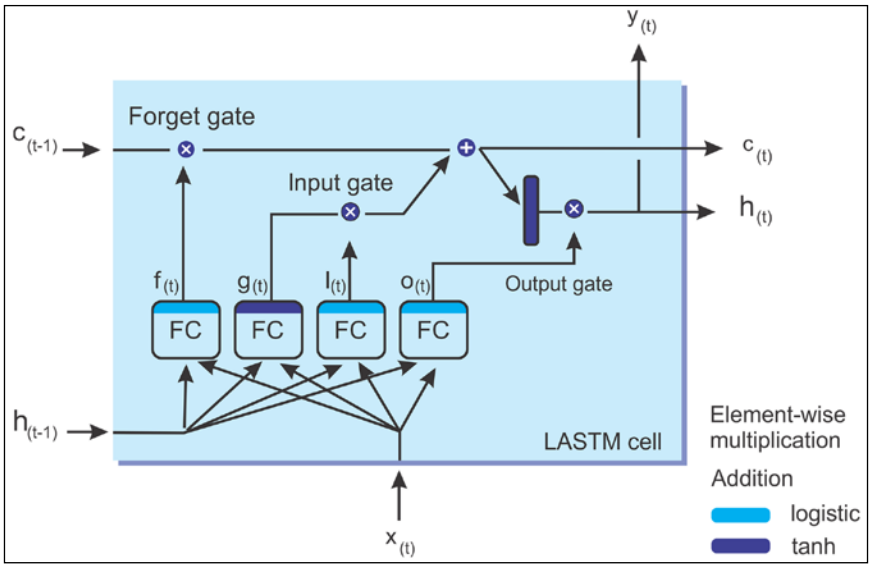


Figure 5: Block diagram of an LSTM cell

Now let's see the mathematical notation behind this architecture. If we don't look at what's inside the LSTM box, the LSTM cell itself looks exactly like a regular memory cell, except that its state is split into two vectors, *h(t)* and *c(t)*:

- *c* is a cell
- *h(t)* is the short-term state
- *c(t)* is the long-term state

Now let's open the box! The key idea is that the network can learn what to store in the long-term state, what to throw away, and what to read from it. As the long-term state c(t–1) traverses the network from left to right, you can see that it first goes through a forget gate, dropping some memories, and then it adds some new memories via the addition operation (which adds the memories that were selected by an input gate). The resulting *c(t)* is sent straight out, without any further transformation.

So, at each timestep, some memories are dropped and some memories are added. Moreover, after the addition operation, the long-term state is copied and passed through the *tanh* function, and then the result is filtered by the output gate. This produces the short-term state *h(t)* (which is equal to the cell's output for this time step *y(t)*). Now let's look at where new memories come from and how the gates work. First, the current input vector *x(t)* and the previous short-term state *h(t–1)* are fed to four different fully connected layers.

The presence of these gates allows LSTM cells to remember information for an indefinite time; if the input gate is below the activation threshold, the cell will retain the previous state, and if the current state is enabled, it will be combined with the input value. As the name suggests, the forget gate resets the current state of the cell (when its value is cleared to 0), and the output gate decides whether the value of the cell must be carried out or not. The following equations are used to do the LSTM computations of a cell's long-term state, its short-term state, and its output at each time step for a single instance:

$$\mathbf{i}_{(t)} = \sigma\left(\mathbf{W}_{xi}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hi}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_{i}\right)$$
$$\mathbf{f}_{(t)} = \sigma\left(\mathbf{W}_{xf}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hf}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_{f}\right)$$
$$\mathbf{o}_{(t)} = \sigma\left(\mathbf{W}_{xo}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{ho}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_{o}\right)$$
$$\mathbf{g}_{(t)} = \tanh\left(\mathbf{W}_{xg}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_{g}\right)$$
$$\mathbf{c}_{(t)} = \mathbf{f}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{i}_{(t)} \otimes \mathbf{g}_{(t)}$$
$$\mathbf{y}_{(t)} = \mathbf{h}_{(t)} = \mathbf{o}_{(t)} \otimes \tanh\left(\mathbf{c}_{(t)}\right)$$

In the preceding equation, W_{xi}, W_{xf}, W_{xo}, and W_{xg} are the weight matrices of each of the four layers for their connection to the input vector *x(t)*. On the other hand, W_{hi}, W_{hf}, W_{ho}, and W_{hg} are the weight matrices of each of the four layers for their connection to the previous short-term state *h(t–1)*. Finally, b_i, b_f, b_o, and b_g are the bias terms for each of the four layers.



TensorFlow initializes bf to a vector full of 1s instead of 0s. This prevents it from forgetting everything at the beginning of training.

GRU cell

There are many other variants of the LSTM cell. One particularly popular variant is the GRU cell, which we will look at now. The cell was proposed by Kyunghyun Cho et al. in a 2014 paper that also introduced the encoder–decoder network we mentioned earlier.



For this type of LSTM, interested readers should refer to the following publications:

- *Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation*," K. Cho et al. (2014).
- A 2015 paper by Klaus Greff et al., "*LSTM: A Search Space Odyssey*," seems to show that all LSTM variants perform roughly the same.

Technically, a GRU cell is a simplified version of an LSTM cell, where both the state vectors are merged into a single vector called *h(t)*. A single gate controller controls both the forget gate and the input gate. If the gate controller outputs a 1, the input gate is open and the forget gate is closed:

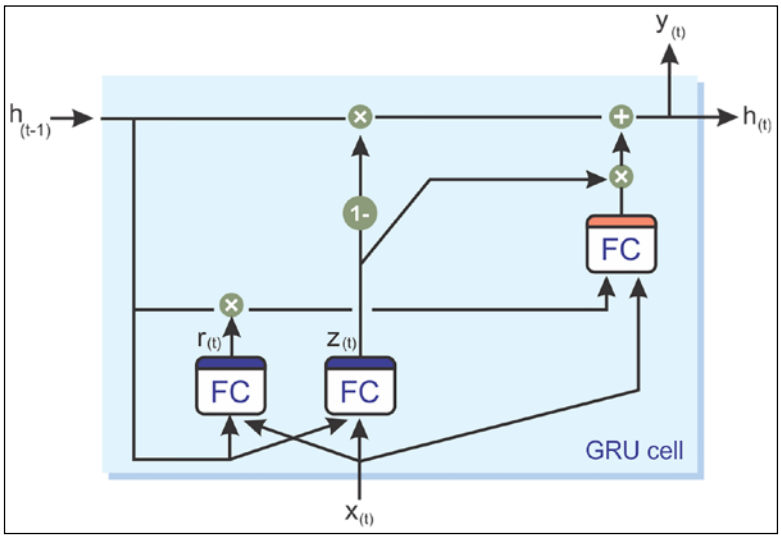


Figure 7: A GRU cell

On the other hand, if it outputs a 0, the opposite happens. Whenever a memory must be stored, the location where it will be stored is erased first, which is actually a frequent variant to the LSTM cell in and of itself. The second simplification is that since the full state vector is output at every time step, there is no output gate. However, there is a new gate controller introduced that controls which part of the previous state will be shown to the main layer.

The following equations are used to do the GRU computations of a cell's long-term state, its short-term state, and its output at each time step for a single instance:

$$\mathbf{z}_{(t)} = \sigma\left(\mathbf{W}_{xz}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hz}{}^{T} \cdot \mathbf{h}_{(t-1)}\right)$$
$$\mathbf{r}_{(t)} = \sigma\left(\mathbf{W}_{xr}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hr}{}^{T} \cdot \mathbf{h}_{(t-1)}\right)$$
$$\mathbf{g}_{(t)} = \tanh\left(\mathbf{W}_{xg}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}{}^{T} \cdot \left(\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)}\right)\right)$$
$$\mathbf{h}_{(t)} = \left(1 - \mathbf{z}_{(t)}\right) \otimes \mathbf{h}_{(t-1)} + \mathbf{z}_{(t)} \otimes \mathbf{g}_{(t)}$$

Creating a GRU cell in TensorFlow is pretty straightforward. An example is given as follows:

```
gru_cell = tf.contrib.rnn.GRUCell(num_units=n_neurons)
```

The preceding simplifications are not a weakness of this type of architecture, but it seems to perform just as well. The LSTM and GRU cells are one of the main reasons behind the success of RNNs in recent years, in particular for applications in NLP. We will see examples using LSTM in this chapter, but the next section shows an example of BRNN for image data classification.

Using BRNN for image classification

Let's now see how to implement a BRNN implementation example using the TensorFlow library (see `bidirectional_rnn.py`). This example is using the MNIST database of handwriting. We begin importing the libraries; notice that `rnn` and `rnn_cell` are TensorFlow libraries:

```
import tensorflow as tf
from tensorflow.contrib import rnnimportnumpy as np
```

The network will classify the MNIST images, so we have to load them:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
```

We define the learning parameters:

```
learning_rate = 0.001
training_steps = 10000
batch_size = 256
display_step = 200
```

To classify images using a BRRN, we consider every image row as a sequence of pixels. Because an MNIST image's shape is 28*28px, we will then handle 28 sequences of 28 steps for every sample.



To execute the script, use the following command:

```
$ python3 bidirectional_rnn.py
```

Configure the network's parameters:

```
# BRNN Network Parameters
num_input = 28 # MNIST data input (img shape: 28*28)
timesteps = 28 # timesteps
num_hidden = 128 # hidden layer num of features
num_classes = 10 # MNIST total classes (0-9 digits)
```

Set up the placeholders, which we use to feed to our network. First, we define a placeholder variable for the input images. This allows us to change the images that are input to the TensorFlow graph. The data-type is set to `float` and the tensor's shape is set to [`None`, `n_steps`, `n_input`], `None` stands for a tensor that may hold an arbitrary number of images:

```
x = tf.placeholder("float", [None, n_steps, n_input])
```

Then we fix a second placeholder variable for the labels associated with the images, which were input in the placeholder variable x. The shape of this placeholder variable is set to `[None, n_classes]`, which means that it may hold an arbitrary number of labels and each label is a vector of length `num_classes`, which is 10 in this case:

```
y = tf.placeholder("float", [None, n_classes])
```

The first variable that must be optimized is weights and it is defined here as a TensorFlow variable that must be initialized with random uniform values and whose shape is [`2*n_hidden`, `n_classes`].Here's the weights definition:

```
# define weights
weights = {'out': tf.Variable(tf.random_normal([2*n_hidden, n_
classes]))}
```

Then we define the corresponding biases:

```
biases = {'out': tf.Variable(tf.random_normal([n_classes]))}
```

With the following BRNN function, we define the weights and network biases:

```
def BiRNN(x, weights, biases):
```

To achieve this purpose, we apply the following sequence of TensorFlow transformations:

```
x = tf.transpose(x, [1, 0, 2])
    x = tf.reshape(x, [-1, n_input])
x = tf.split(axis=0, num_or_size_splits=n_steps, value=x)
```

Unlike in the previous model, we define two types of LSTM cell, a forward cell and a backward cell:

```
lstm_fw_cell = rnn.BasicLSTMCell(n_hidden, forget_bias=1.0)
lstm_bw_cell = rnn.BasicLSTMCell(n_hidden, forget_bias=1.0)
```

Then we build the bi-directional network using the imported class `rnn.bidirectional_rnn()`.Similarly to the unidirectional case, `rnn.bidirectional_rnn()` takes an input and builds independent forward and backward RNNs, with the final forward and backward outputs depth-concatenated:

```
try:
     outputs, _, _ = rnn.static_bidirectional_rnn\
                      (lstm_fw_cell, lstm_bw_cell, x,dtype=tf.float32)
except Exception:
        outputs = rnn.static_bidirectional_rnn\
                      (lstm_fw_cell, lstm_bw_cell, x,dtype=tf.float32)
```

The `input_size` of forward and backward cells must match. Notice that outputs will have the following format:

```
[time][batch][cell_fw.output_size + cell_bw.output_size]
```

The `BiRNN` returns an output tensor for determining which of the 10 classes the input image belongs to:

```
return tf.matmul(outputs[-1], weights['out']) + biases['out']
```

The value returned by `BiRNN` will then be passed to the pred tensor:

```
pred = BiRNN(x, weights, biases)
```

We have to compute the cross-entropy value for each classified image because we must have a measure of how well the model works individually on each image. Using the cross-entropy to guide the network's optimization procedure we need a single scalar value, so we simply take the cross-entropy average (`tf.reduce_mean`) evaluated for all the classified images:

```
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_
logits(logits=pred, labels=y))
```

The obtained cost measure will be minimized by an optimizer. We use the `AdamOptimizer`, which is an advanced form of Adam optimizer:

```
optimizer = tf.train.AdamOptimizer\
(learning_rate=learning_rate).minimize(cost)
```

We add performance measures to be able to display the progress during the training phase. It is a vector of Booleans where the predicted class equals the true class of each image:

```
correct_pred = tf.equal(tf.argmax(pred,1), tf.argmax(y,1))
```

The `correct_pred` is used here to compute the classification accuracy by first type-casting the vector of Booleans to floats, so that False becomes 0 and True becomes 1, and then calculating the average of these numbers:

```
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

All the variables must be initialized before we start optimizing them:

```
Init_op = tf.global_variables_initializer()
```

```
Now let's create three separate lists that hold the training loss,
training accuracy, and the step obtained in the BRNN training step:
loss_list = []
accuracy_list = []
step_list = []
```

We then create a session that will execute the graph:

```
with tf.Session() as sess:
    sess.run(init_op)
    step = 1
print("Optimization started!")
```

During the session, we get a batch of training examples:

```
while step * batch_size < training_iters:
```

`batch_x` now holds a subset of training images and `batch_yis` a subset of true labels for those images:

```
batch_x, batch_y = mnist.train.next_batch(batch_size)
batch_x = batch_x.reshape((batch_size, n_steps, n_input))
```

We put the batch sets into a `feed_dict` with the proper names for the placeholder variables, then we run the optimizer through `sess.run`:

```
        sess.run(optimizer, feed_dict={x: batch_x, y: batch_y})
        if step % display_step == 0:
```

We calculate the accuracy and the loss values on these sets. Also, we save the minibatch loss, training accuracy, and step information:

```
acc = sess.run(accuracy,\
                      feed_dict={x: batch_x, y: batch_y})
            loss = sess.run(cost,\
                        feed_dict={x: batch_x, y: batch_y})
            print("Iter " + str(step*batch_size) +\
                                    ", Minibatch Loss= " + \
                  "{:.6f}".format(loss) + ", Training Accuracy= " + \
                  "{:.5f}".format(acc))
loss_list.append(loss)
            accuracy_list.append(acc)
            step_list.append(step)
        step += 1
    print("Optimization Finished!")
```

At the end of the training session, we get a batch of testing examples:

```
    test_len = 128
    test_data = mnist.test.images\
                    [:test_len].reshape((-1, n_steps, n_input))
    test_label = mnist.test.labels[:test_len]
```

Finally, we can calculate and display the accuracy on this test set:

```
    print("Testing Accuracy:", \
sess.run(accuracy, feed_dict={x: test_data, y: test_label}))
```

We show only an excerpt of the output. Here you can visualize the loss value and the accuracy evaluated on the batch sets:

```
Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes.
Extracting /tmp/data/train-images-idx3-ubyte.gz
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
Optimization started!
Step 1, Minibatch Loss= 2.4194, Training Accuracy= 0.094
Step 200, Minibatch Loss= 0.1818, Training Accuracy= 0.938
Step 400, Minibatch Loss= 0.1370, Training Accuracy= 0.945
Step 600, Minibatch Loss= 0.1032, Training Accuracy= 0.953
Step 800, Minibatch Loss= 0.0879, Training Accuracy= 0.961
Step 1000, Minibatch Loss= 0.0516, Training Accuracy= 0.977
Optimization Finished!

Finally, the evaluated accuracy on the test set is reported as
follows:
Testing Accuracy: 0.992188
```

Now let's see the training error and accuracy per iteration from the graph:

```
# Plot loss over time
plt.plot(step_list, loss_list, 'r--', label='BRNN minibatch loss per
iteration', linewidth=4)
plt.title('BRNN minibatch loss per iteration')
plt.xlabel('Iteration')
plt.ylabel('BRNN minibatch loss')
plt.legend(loc='upper right')
plt.show()
>>>
```

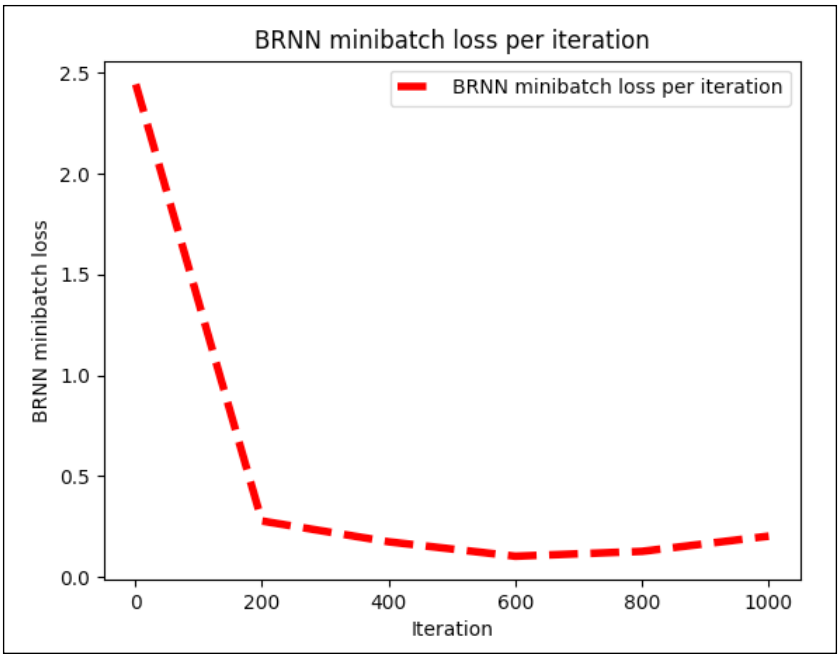


Figure 9: The minibatch loss per iteration for the BRNN

From the preceding figure it is clear that the training loss decreased, was lowest after **600** iterations, and slightly increased until **1000** iterations. Now let's plot accuracy over time:

```
plt.plot(step_list, accuracy_list, 'r--', label='BRNN training
accuracy per iteration', linewidth=4)
plt.title('BRNN training accuracy per iteration')
plt.xlabel('Iteration')
plt.ylabel('BRNN training accuracy')
plt.legend(loc='upper right')
plt.show()
>>>
```

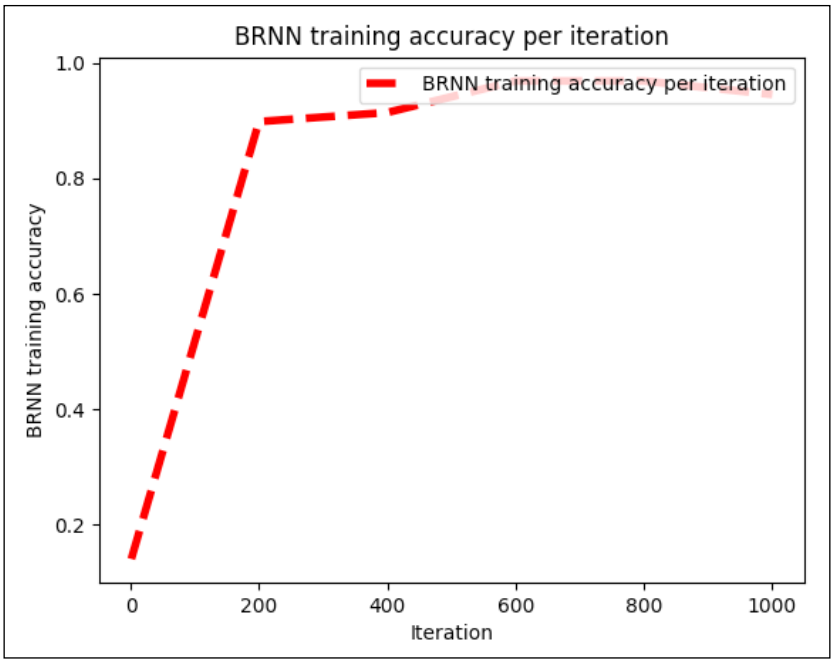


Figure 10: The training accuracy per iteration for the bi-directional RNN

From the preceding figure, it is clear that the training accuracy increased to about 100% and had hit a plateau by 1000 iterations.

Implementing an RNN for spam prediction

In this section, we will see how to implement an RNN in TensorFlow to predict spam/ham from texts. A popular spam dataset from the UCI ML repository will be used, which will be downloaded from the link at `http://archive.ics.uci.edu/ml/machine-learning-databases/00228/smsspamcollection.zip`.

In *Chapter 6*, *Predictive Analytics Pipelines for NLP*, we used this same dataset in the *Using BOW for predictive analytics* section and achieved an overall accuracy of 88.30703012912483% on the test set. So, I am skipping the trivial preprocessing and the expletory analysis of the dataset. In this section, we will see how we can achieve even higher accuracy using RNN classifier.

We start by importing required libraries and models:

```
import os
import re
import io
import requests
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from zipfile import ZipFile
from tensorflow.python.framework import ops
import warnings
```

Additionally, you can stop printing warning produced by TensorFlow if you want:

```
warnings.filterwarnings("ignore")
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
ops.reset_default_graph()
```

Now let's create the TensorFlow session for the graph:

```
sess = tf.Session()
```

Now the next task that I am going to show is setting RNN parameters:

```
epochs = 1000
batch_size = 250
max_sequence_length = 25
rnn_size = 10
embedding_size = 50
min_word_frequency = 10
learning_rate = 0.0001
dropout_keep_prob = tf.placeholder(tf.float32)
```

Let's manually download the dataset and store it in a `text_data.txt` file under the `temp` directory. First we set the path:

```
data_dir = 'temp'
data_file = 'text_data.txt'
if not os.path.exists(data_dir):
    os.makedirs(data_dir)
```

Now directly download the dataset in zipped format:

```
if not os.path.isfile(os.path.join(data_dir, data_file)):
    zip_url = 'http://archive.ics.uci.edu/ml/machine-learning-
databases/00228/smsspamcollection.zip'
    r = requests.get(zip_url)
```

```
    z = ZipFile(io.BytesIO(r.content))
    file = z.read('SMSSpamCollection')
```

Now we still need to format the data:

```
    text_data = file.decode()
    text_data = text_data.encode('ascii',errors='ignore')
    text_data = text_data.decode().split('\n')
```

Now store the data in the directory mentioned earlier, to save the data to a text file:

```
    with open(os.path.join(data_dir, data_file), 'w') as file_conn:
        for text in text_data:
            file_conn.write("{}\n".format(text))
else:
    text_data = []
    with open(os.path.join(data_dir, data_file), 'r') as file_conn:
        for row in file_conn:
            text_data.append(row)
    text_data = text_data[:-1]
```

Now let's split off the words with a word length of at least two:

```
text_data = [x.split('\t') for x in text_data if len(x)>=1]
[text_data_target, text_data_train] = [list(x) for x in zip(*text_
data)]
```

Now we create a text cleaning function:

```
def clean_text(text_string):
    text_string = re.sub(r'([^\s\w]|_|[0-9])+', '', text_string)
    text_string = " ".join(text_string.split())
    text_string = text_string.lower()
    return(text_string)
```

We call the preceding method to clean the text:

```
text_data_train = [clean_text(x) for x in text_data_train]
```

Now one of the most important tasks that needs to be done is creating word embedding– we change the text into numeric vectors:

```
vocab_processor = tf.contrib.learn.preprocessing.
VocabularyProcessor(max_sequence_length, min_frequency=min_word_
frequency)
text_processed = np.array(list(vocab_processor.fit_transform(text_
data_train)))
```

Now let's shuffle to make the dataset balanced:

```
text_processed = np.array(text_processed)
text_data_target = np.array([1 if x=='ham' else 0 for x in text_data_
target])
shuffled_ix = np.random.permutation(np.arange(len(text_data_target)))
x_shuffled = text_processed[shuffled_ix]
y_shuffled = text_data_target[shuffled_ix]
```

Now that we have the data shuffled, we then split data into training and test sets:

```
ix_cutoff = int(len(y_shuffled)*0.75)
x_train, x_test = x_shuffled[:ix_cutoff], x_shuffled[ix_cutoff:]
y_train, y_test = y_shuffled[:ix_cutoff], y_shuffled[ix_cutoff:]
vocab_size = len(vocab_processor.vocabulary_)
print("Vocabulary size: {:d}".format(vocab_size))
print("Training set size: {:d}".format(len(y_train)))
print("Test set size: {:d}".format(len(y_test)))
>>>
Vocabulary size: 933
Training set size: 4180
Test set size: 1394
```

Now before we start the training, let's create placeholders for the TensorFlow graph:

```
x_data = tf.placeholder(tf.int32, [None, max_sequence_length])
y_output = tf.placeholder(tf.int32, [None])
```

Now let's create the embedding:

```
embedding_mat = tf.Variable(tf.random_uniform([vocab_size, embedding_
size], -1.0, 1.0))
embedding_output = tf.nn.embedding_lookup(embedding_mat, x_data)
```

Now it's time to construct our RNN network. The following code defines the RNN cell:

```
cell = tf.nn.rnn_cell.BasicRNNCell(num_units = rnn_size)
output, state = tf.nn.dynamic_rnn(cell, embedding_output, dtype=tf.
float32)
output = tf.nn.dropout(output, dropout_keep_prob)
```

Now let's define the way to get output of RNN sequence:

```
output = tf.transpose(output, [1, 0, 2])
last = tf.gather(output, int(output.get_shape()[0]) - 1)
```

Now we define the weights and the biases for the RNN:

```
weight = tf.Variable(tf.truncated_normal([rnn_size, 2], stddev=0.1))
bias = tf.Variable(tf.constant(0.1, shape=[2]))
```

The logits output is then defined, which uses both the weight and the bias from the previous code:

```
logits_out = tf.nn.softmax(tf.matmul(last, weight) + bias)
```

Now we define the losses for each prediction so that later on these can contribute to the loss function:

```
losses = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits_
out, labels=y_output)
```

We then define the loss function:

```
loss = tf.reduce_mean(losses)
```

We now define the accuracy of each prediction:

```
accuracy = tf.reduce_mean(tf.cast(tf.equal(tf.argmax(logits_out, 1),
tf.cast(y_output, tf.int64)), tf.float32))
We then create the training_op with RMSPropOptimizer:
optimizer = tf.train.RMSPropOptimizer(learning_rate)
train_step = optimizer.minimize(loss)
```

Now let's initialize all the variables using the `global_variables_initializer()` method:

```
init_op = tf.global_variables_initializer()
sess.run(init_op)
```

Additionally, we can create some empty lists to keep track of the training loss, test loss, training accuracy, and the test accuracy in each epoch:

```
train_loss = []
test_loss = []
train_accuracy = []
test_accuracy = []
```

Now we are ready to perform the training, so let's get started. The workflow of the training goes as follows:

1. Shuffle training data.
2. Select the training set and calculate generations.
3. Run train step for each batch.

4. Run loss and accuracy for training.
5. Run the evaluation steps.

The following code includes all the previous steps:

```
for epoch in range(epochs):
    shuffled_ix = np.random.permutation(np.arange(len(x_train)))
    x_train = x_train[shuffled_ix]
    y_train = y_train[shuffled_ix]
    num_batches = int(len(x_train)/batch_size) + 1

    for i in range(num_batches):
        min_ix = i * batch_size
        max_ix = np.min([len(x_train), ((i+1) * batch_size)])
        x_train_batch = x_train[min_ix:max_ix]
        y_train_batch = y_train[min_ix:max_ix]
        train_dict = {x_data: x_train_batch, y_output: y_train_batch,
dropout_keep_prob:0.5}
        sess.run(train_step, feed_dict=train_dict)
   temp_train_loss, temp_train_acc = sess.run([loss, accuracy], feed_
dict=train_dict)
    train_loss.append(temp_train_loss)
    train_accuracy.append(temp_train_acc)
    test_dict = {x_data: x_test, y_output: y_test, dropout_keep_
prob:1.0}
    temp_test_loss, temp_test_acc = sess.run([loss, accuracy], feed_
dict=test_dict)
    test_loss.append(temp_test_loss)
    test_accuracy.append(temp_test_acc)
    print('Epoch: {}, Test Loss: {:.2}, Test Acc: {:.2}'.
format(epoch+1, temp_test_loss, temp_test_acc))

print('\nOverall accuracy on test set (%): {}'.format(np.mean(temp_
test_acc)*100.0))
>>>
Epoch: 1, Test Loss: 0.68, Test Acc: 0.82
Epoch: 2, Test Loss: 0.68, Test Acc: 0.82
Epoch: 3, Test Loss: 0.67, Test Acc: 0.82
…
Epoch: 997, Test Loss: 0.36, Test Acc: 0.96
Epoch: 998, Test Loss: 0.36, Test Acc: 0.96
Epoch: 999, Test Loss: 0.35, Test Acc: 0.96
Epoch: 1000, Test Loss: 0.35, Test Acc: 0.96
Overall accuracy on test set (%): 96.19799256324768
```

Well done! The accuracy of the RNN network is above 96%, which is outstanding and is much better than the linear CBOW model. Now let's observe how the loss propagated across each iteration and over time:

```
epoch_seq = np.arange(1, epochs+1)
plt.plot(epoch_seq, train_loss, 'k--', label='Train Set')
plt.plot(epoch_seq, test_loss, 'r-', label='Test Set')
plt.title('RNN training/test loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(loc='upper left')
plt.show()
```

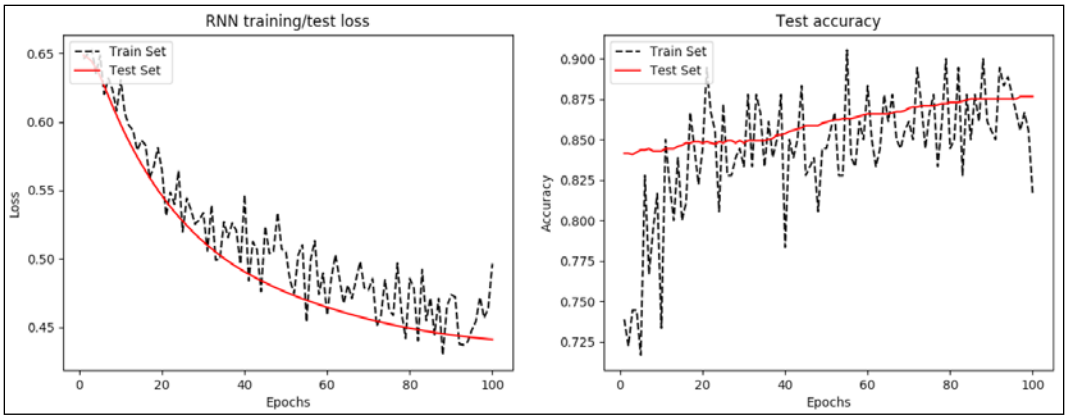


Figure 11: a) RNN training and test loss per epoch b) test accuracy per epoch

We also plot the accuracy over time:

```
plt.plot(epoch_seq, train_accuracy, 'k--', label='Train Set')
plt.plot(epoch_seq, test_accuracy, 'r-', label='Test Set')
plt.title('Test accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(loc='upper left')
plt.show()
```

The next application is for using time series data for predictive modeling. We will also see how we can develop more complex RNN using the LSTM network.

Developing a predictive model for time series data

RNNs, specifically LSTM models, are often a difficult topic to understand. Time series prediction is a useful application for RNNs because of temporal dependencies in the data. Time series data is abundantly available online. In this section, we will see an example of using LSTM for handling time series data. Our LSTM network will be able to predict a number of passengers for future years.

Description of the dataset

The dataset that I will be using is data about international airline passengers from 1949 to 1960. The dataset can be downloaded from

`https://datamarket.com/data/set/22u3/international-airline-passengers-monthly-totals-in#!ds=22u3&display=line`. Figure 12 shows the metadata of the international airline passengers:

Dataset title	**International airline passengers: monthly totals in thousands. Jan 49 – Dec 60**
Last updated	1 Feb 2014, 19:52
Last updated by source	20 Jun 2012
Provider	Time Series Data Library
Provider source	Box & Jenkins (1976)
Source URL	http://datamarket.com/data/list/?q=provider:tsdl
Units	Thousands of passengers
Dataset metrics	144 fact values in 1 timeseries.
Time granularity	Month
Time range	Jan 1949 – Dec 1960
Language	English
License	Default open license
License summary	This data release is licensed as follows: You may copy and redistribute the data. You may make derivative works from the data. You may use the data for commercial purposes. You may not sublicense the data when redistributing it. You may not redistribute the data under a different license. Source attribution on any use of this data: Must refer source.
Description	Transport and tourism, Source: Box & Jenkins (1976), in file: data/airpass, Description: International airline passengers: monthly totals in thousands. Jan 49 – Dec 60

Figure 12: Metadata of international airline passengers (source: `https://datamarket.com/`)

You can download the data by choosing the Export tab and then selecting **CSV (,)** in the **Export** group. You'll have to manually edit the CSV file to remove the header line, as well as the additional footer line. I have downloaded and saved the data file named `international-airline-passengers.csv`. Figure 13 shows a nice plot of the time series data:

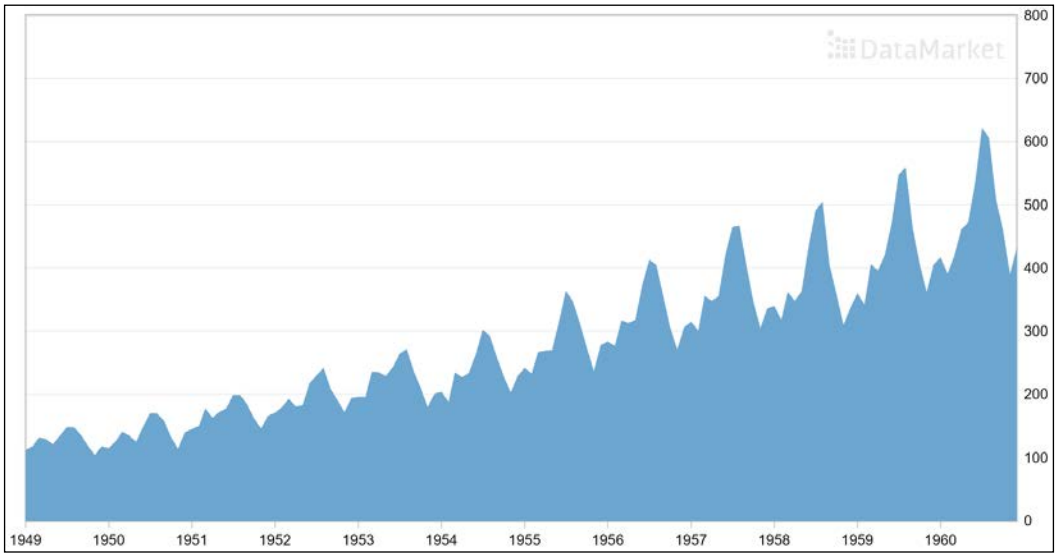


Figure 13: International airline passengers; monthly totals in thousands from Jan 49–Dec 60

Preprocessing and exploratory analysis

Now let's load the original dataset and see some facts. First, we load the time series as follows (see `time_series_preprocessor .py`):

```
import csv
import numpy as np
```

Here we can see the signature of `load_series()`, which is a user-defined method that loads the time series and normalizes it:

```
def load_series(filename, series_idx=1):
    try:
        with open(filename) as csvfile:
csvreader = csv.reader(csvfile)
            data = [float(row[series_idx]) for row in csvreader if
len(row) > 0]
            normalized_data = (data - np.mean(data)) / np.std(data)
```

```
            return normalized_data
        except IOError:
            return None
```

Now let's invoke the preceding method to load the time series and print (issue `$ python3 plot_time_series.py` on the terminal) the number of entries in the time series in the dataset:

```
import csv
import numpy as np
import matplotlib.pyplot as plt
import time_series_preprocessor as tsp
timeseries = tsp.load_series('international-airline-passengers.csv')
print(timeseries)
>>>
[-1.40777884 -1.35759023 -1.24048348 -1.26557778 -1.33249593
-1.21538918
 -1.10664719 -1.10664719 -1.20702441 -1.34922546 -1.47469699
-1.35759023
…..
 2.85825285  2.72441656  1.9046693   1.5115252   0.91762667
1.26894693]
print(np.shape(timeseries))
>>>
144
```

That means there are `144` entries in the time series. Now let's plot the time series as follows:

```
plt.figure()
plt.plot(timeseries)
plt.title('Normalized time series')
plt.xlabel('ID')
plt.ylabel('Normalized value')
plt.legend(loc='upper left')
plt.show()
>>>
```

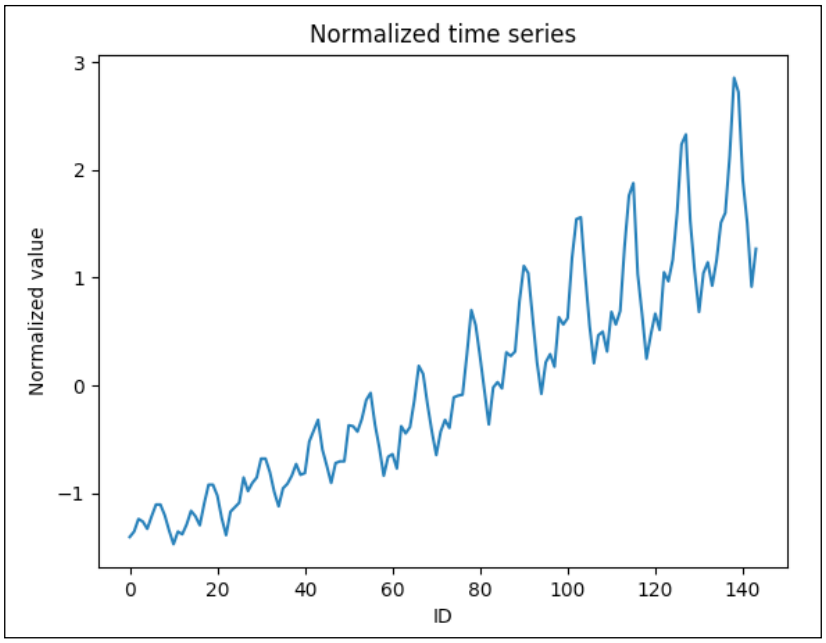


Figure 14: Time series (y-axis normalized value versus x-axis ID)

Once we have loaded the time series dataset, the next task is to prepare the training set. Since we'll be evaluating the model multiple times to predict future values, for this we will split the data into training and test. To be more specific, the `split_data()` function divides the dataset into two components for training and testing–that is, 75% for the training, and the test 25% will be dedicated for the testing:

```
def split_data(data, percent_train):
    num_rows = len(data)
    train_data, test_data = [], []
    for idx, row in enumerate(data):
        if idx < num_rows * percent_train:
            train_data.append(row)
        else:
            test_data.append(row)
    return train_data, test_data
```

LSTM predictive model

Once we have our dataset ready, we can now train the predictor by loading the data in the acceptable format. For this step, I have written a Python script called `TimeSeriesPredictor.py` that starts with the necessary library and modules (issue `$ python3 TimeSeriesPredictor.py` code on the terminal for this script):

```
import numpy as np
import tensorflow as tf
from tensorflow.python.ops import rnn, rnn_cell
import time_series_preprocessor as tsp
import matplotlib.pyplot as plt
```

Next we define the hyperparameters for the LSTM network (tune it accordingly):

```
input_dim = 1
seq_size = 5
hidden_dim = 5
```

We now define the weight variables (no biases) and input placeholders:

```
W_out = tf.Variable(tf.random_normal([hidden_dim, 1]), name='W_out')
b_out = tf.Variable(tf.random_normal([1]), name='b_out')
x = tf.placeholder(tf.float32, [None, seq_size, input_dim])
y = tf.placeholder(tf.float32, [None, seq_size])
```

The next task is to construct the LSTM network. The following method, `LSTM_Model()`, takes three parameters as follows:

- `x`: Inputs of size [`T`, `batch_size`, `input_size`]
- `W`: Matrix of fully-connected output layer weights
- `b`: Vector of fully-connected output layer biases

Now let's see the signature of the method:

```
def LSTM_Model():
        cell = rnn_cell.BasicLSTMCell(hidden_dim)
        outputs, states = rnn.dynamic_rnn(cell, x, dtype=tf.float32)
        num_examples = tf.shape(x)[0]
        W_repeated = tf.tile(tf.expand_dims(W_out, 0), [num_examples,
1, 1])
        out = tf.matmul(outputs, W_repeated) + b_out
        out = tf.squeeze(out)
        return out
```

Additionally, we also create three empty lists to store the training loss, test loss, and the step:

```
train_loss = []
test_loss = []
step_list = []
```

The next method called `train()` is used to train the previous LSTM network:

```
def trainNetwork(train_x, train_y, test_x, test_y):
        with tf.Session() as sess:
            tf.get_variable_scope().reuse_variables()
            sess.run(tf.global_variables_initializer())
            max_patience = 3
            patience = max_patience
            min_test_err = float('inf')
            step = 0
            while patience > 0:
                _, train_err = sess.run([train_op, cost], feed_
dict={x: train_x, y: train_y})
                if step % 100 == 0:
                    test_err = sess.run(cost, feed_dict={x: test_x, y:
test_y})
                    print('step: {}\t\ttrain err: {}\t\ttest err: {}'.
format(step, train_err, test_err))
                    train_loss.append(train_err)
                    test_loss.append(test_err)
                    step_list.append(step)
                    if test_err < min_test_err:
                        min_test_err = test_err
                        patience = max_patience
                    else:
                        patience -= 1
                step += 1
            save_path = saver.save(sess, 'model.ckpt')
            print('Model saved to {}'.format(save_path))
```

Now, the next task is to create the cost optimizer and instantiate the `training_op`:

```
cost = tf.reduce_mean(tf.square(LSTM_Model()- y))
train_op = tf.train.AdamOptimizer(learning_rate=0.003).minimize(cost)
```

Additionally, here we have an auxiliary op called saving the model:

```
saver = tf.train.Saver()
```

Now that we have the model created, the next method called `testLSTM()` is used to test the prediction power of the model:

```
def testLSTM(sess, test_x):
        tf.get_variable_scope().reuse_variables()
        saver.restore(sess, 'model.ckpt')
        output = sess.run(LSTM_Model(), feed_dict={x: test_x})
        return output
```

Now for plotting the predicted results, we have another function called `plot_results()`. The signature is given as follows:

```
def plot_results(train_x, predictions, actual, filename):
    plt.figure()
    num_train = len(train_x)
    plt.plot(list(range(num_train)), train_x, color='b',
label='training data')
    plt.plot(list(range(num_train, num_train + len(predictions))),
predictions, color='r', label='predicted')
    plt.plot(list(range(num_train, num_train + len(actual))), actual,
color='g', label='test data')
    plt.legend()
    if filename is not None:
        plt.savefig(filename)
    else:
        plt.show()
```

Model evaluation

For model evaluation, we have another method called `main()`, which actually invokes the previously mentioned methods to create and train the LSTM network. The workflow of the following code is given as follows:

1. Load the data.
2. Slide a window through the time-series data to construct the training dataset.
3. Do the same window sliding strategy to construct the test dataset.
4. Train a model on the training dataset.
5. Visualize the model's performance.

Now let's see the signature of the method:

```
def main():
    data = tsp.load_series('international-airline-passengers.csv')
    train_data, actual_vals = tsp.split_data(data=data, percent_
train=0.75)
    train_x, train_y = [], []
    for i in range(len(train_data) - seq_size - 1):
        train_x.append(np.expand_dims(train_data[i:i+seq_size],
axis=1).tolist())
        train_y.append(train_data[i+1:i+seq_size+1])
    test_x, test_y = [], []
    for i in range(len(actual_vals) - seq_size - 1):
        test_x.append(np.expand_dims(actual_vals[i:i+seq_size],
axis=1).tolist())
        test_y.append(actual_vals[i+1:i+seq_size+1])
    trainNetwork(train_x, train_y, test_x, test_y)
    with tf.Session() as sess:
        predicted_vals = testLSTM(sess, test_x)[:,0]
        print('predicted_vals', np.shape(predicted_vals))
        # Following prediction results of the model given ground truth
values
        plot_results(train_data, predicted_vals, actual_vals, 'ground_
truth_predition.png')
        prev_seq = train_x[-1]
        predicted_vals = []
        for i in range(1000):
            next_seq = testLSTM(sess, [prev_seq])
            predicted_vals.append(next_seq[-1])
            prev_seq = np.vstack((prev_seq[1:], next_seq[-1]))
        # Following predictions results where only the training data
was given
        plot_results(train_data, predicted_vals, actual_vals,
'prediction_on_train_set.png')
>>>
```

Finally, we call the method to perform the training:

```
main()
```

It also plots the prediction results of the model given ground truth values and predictions results where only the training data was given:

```
>>>
```

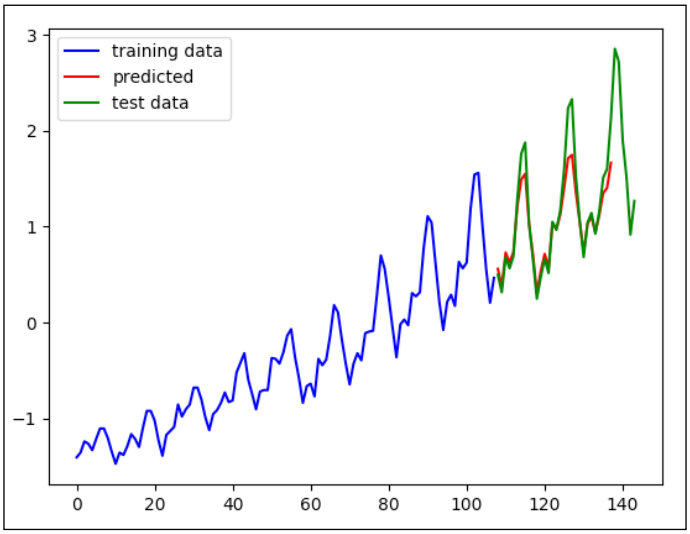


Figure 15: Results of the model on the ground truth values

The next graph shows the predictions results on the training data. This procedure has less information available, but it still did a good job matching trends of the data:

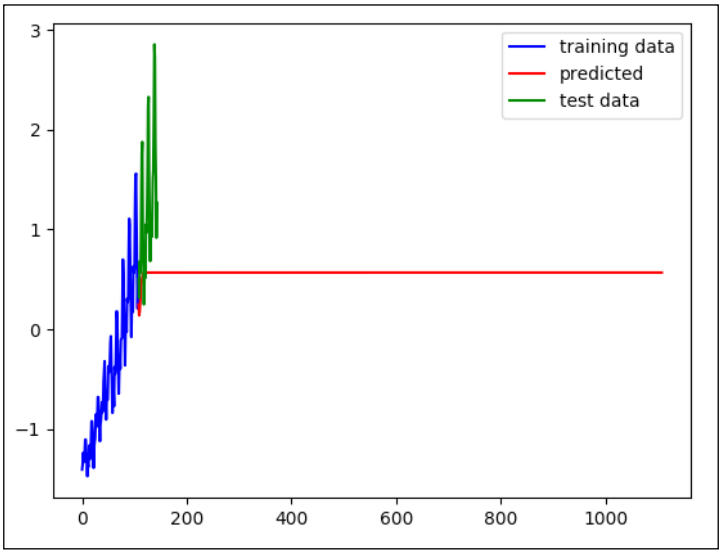


Figure 16: Results of the model on the training set

The following method helps us plot the training and test errors:

```
def plot_error():
  # Plot traimning loss over time
  plt.plot(step_list, train_loss, 'r--', label='LSTM training loss per
iteration', linewidth=4)
  plt.title('LSTM training loss per iteration')
  plt.xlabel('Iteration')
  plt.ylabel('Training loss')
  plt.legend(loc='upper right')
  plt.show()

  # Plot test loss over time
  plt.plot(step_list, test_loss, 'r--', label='LSTM test loss per
iteration', linewidth=4)
  plt.title('LSTM test loss per iteration')
  plt.xlabel('Iteration')
  plt.ylabel('Test loss')
  plt.legend(loc='upper left')
  plt.show()
```

Now we call the previous method as follows:

```
plot_error()
>>>
```

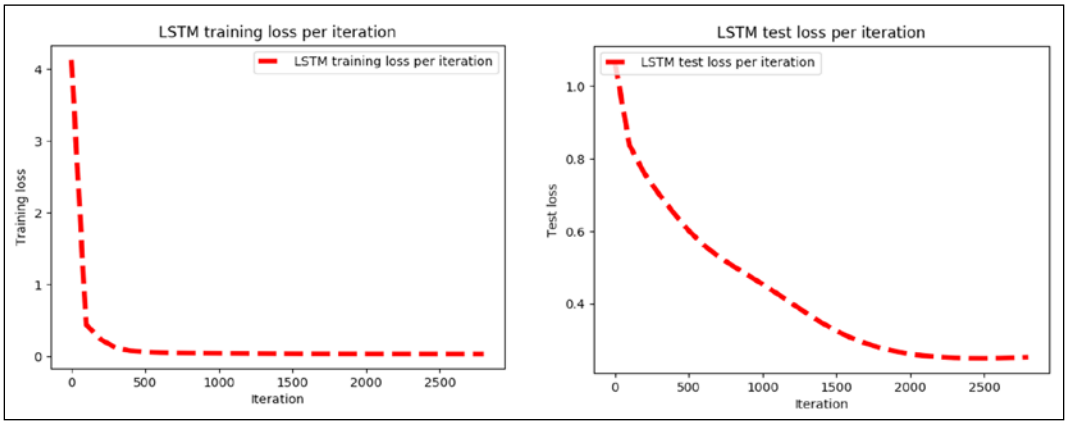


Figure 17: a) LSTM training loss per iteration, b) LSTM test loss per iteration

We can use a time series predictor to reproduce realistic fluctuations in data. Now you can prepare your own dataset and do some other predictive analytics. The next application is about sentiment analysis from the product or movie review dataset. We will also see how we can develop more complex RNNs using the LSTM network.

An LSTM predictive model for sentiment analysis

Sentiment analysis is one of the most widely performed tasks in NLP. An LSTM network can be used to classify short texts into desired categories–that is, classification problems. For example, a set of tweet texts can be categorized as either positive or negative. In this section, we will see such an example.

Network design

The implemented LSTM network will have three layers: embedding layer, RNN layer, and softmax layer. The high-level view can be seen in figure 18. Here I summarize the functionalities of all the layers:

- **Embedding layer**: As we have seen in *Chapter 8*, *Using Convolutional Neural Networks for Predictive Analytics*, text datasets cannot be fed to DNNs directly, but an additional layer called the embedding layer is required. For this layer, we transform each input–that is, a tensor of k words into a tensor of kN-dimensional vectors. This is called the word embedding, where N is the embedding size. Every word will be associated with a vector of weights that needs to be learned during the training process. You can gain more insight into word embedding at vector representations of words.
- **RNN layer**: Once we have the embedding layer constructed, there will be a new layer called RNN, which is made out of LSTM cells with a dropout wrapper. LSTM weights need to be learned during the training process, as described in the previous sections. The RNN layer is unrolled dynamically (as shown in figure 4), taking k-word embedding as input and outputting kM-dimensional vectors, where M is the hidden size of LSTM cells.
- **Softmax or sigmoid layer**: The RNN-layer output is averaged across k time steps, obtaining a single tensor of size M. Finally, a softmax layer, for example, is used to compute classification probabilities:

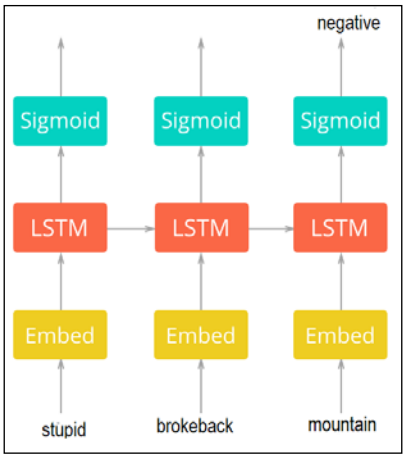


Figure 18: The high-level view of the LSTM network for sentiment analysis

We will see later how cross-entropy can be used as the loss function and RMSProp is the optimizer that minimizes it.

LSTM model training

The UMICH SI650 - Sentiment Classification dataset (with duplication removed), which contains the data about product or movie reviews donated by the University of Michigan was downloaded from `https://inclass.kaggle.com/c/si650winter11/data`. Later on, I cleaned unwanted or special characters, before getting the tokens (see `data.csv` file). The following script also removes stop words while preparing the dataset (see `data_preparation.py`). Some samples are given that are either labeled as negative or positive (1 is positive and 0 is the negative sentiment about an item):

Sentiment	SentimentText
1	The Da Vinci Code book is just awesome.
1	I liked the Da Vinci Code a lot.
0	OMG, I HATE BROKEBACK MOUNTAIN.
0	I hate Harry Potter.

Table 1: A sample of the sentiment dataset

Now let's see a step-by-step example of training of the LSTM network for this task. First, we import the necessary modules and packages (execute the `train.py` file):

```
from data_preparation import Preprocessing
from lstm_network import LSTM_RNN_Network
import tensorflow as tf
import pickle
import datetime
import time
import os
import matplotlib.pyplot as plt
```

In the preceding import declaration, `data_preparation` and `lstm_network` are two helper Python scripts that are used for dataset preparation and network design. We will see more details shortly. Now let's define parameters for the LSTM:

```
tf.flags.DEFINE_string('data_dir', 'data/', 'Data directory containing
\'data.csv\' (must have columns \'SentimentText\' and \'Sentiment\').'
                       ' Intermediate files will automatically be
stored here')
tf.flags.DEFINE_string('stopwords_file', 'data/stopwords.txt', 'Path
to stopwords file. If stopwords_file is None, no stopwords will be
used')
tf.flags.DEFINE_integer('n_samples', None,'Number of samples to use
from the dataset. Set n_samples=None to use the whole dataset')
tf.flags.DEFINE_string('checkpoints_root', 'checkpoints', 'Checkpoints
directory. Parameters will be saved there')
tf.flags.DEFINE_string('summaries_dir', 'logs', 'Directory where
TensorFlow summaries will be stored')
tf.flags.DEFINE_integer('batch_size', 100, 'Batch size')
tf.flags.DEFINE_integer('train_steps', 1000, 'Number of training
steps')
tf.flags.DEFINE_integer('hidden_size', 75, 'Hidden size of LSTM
layer')
tf.flags.DEFINE_integer('embedding_size', 75, 'Size of embeddings
layer')
tf.flags.DEFINE_integer('random_state', 0, 'Random state used for data
split, default is 0')
tf.flags.DEFINE_float('learning_rate', 0.01, 'RMSProp learning rate')
tf.flags.DEFINE_float('test_size', 0.2, '0<test_size<1. Proportion of
the dataset to be included in the test split.')
tf.flags.DEFINE_float('dropout_keep_prob', 0.5, '0<dropout_keep_
prob<=1. Dropout keep-probability')
```

```
tf.flags.DEFINE_integer('sequence_len', None, 'Maximum sequence
length. Let m be the maximum sequence length in the' ' dataset. Then,
it\'s required that sequence_len >= m. If sequence_len'' is None, then
it\'ll be automatically assigned to m')
tf.flags.DEFINE_integer('validate_every', 100, 'Step frequency in
order to evaluate the model using a validation set')
FLAGS = tf.flags.FLAGS
```



For this step, execute the train.py script as follows:

```
$ python3 train.py
```

I believe the previous parameters are self-explanatory from their descriptions. The next task is to prepare summaries to be used by the TensorBoard:

```
summaries_dir = '{0}/{1}'.format(FLAGS.summaries_dir, datetime.
datetime.now().strftime('%d_%b_%Y-%H_%M_%S'))
train_writer = tf.summary.FileWriter(summaries_dir + '/train')
validation_writer = tf.summary.FileWriter(summaries_dir + '/
validation')
```

Now let's prepare the model directory:

```
model_name = str(int(time.time()))
model_dir = '{0}/{1}'.format(FLAGS.checkpoints_root, model_name)
if not os.path.exists(model_dir):
    os.makedirs(model_dir)
```

Finally, we save the preceding configuration:

```
FLAGS._parse_flags()
config = FLAGS.__dict__['__flags']
with open('{}/config.pkl'.format(model_dir), 'wb') as f:
    pickle.dump(config, f)
```

Let's prepare data and build a TensorFlow graph (see the `data_preparation.py` file):

```
data_lstm = Preprocessing(data_dir=FLAGS.data_dir,
                  stopwords_file=FLAGS.stopwords_file,
                  sequence_len=FLAGS.sequence_len,
                  test_size=FLAGS.test_size,
                  val_samples=FLAGS.batch_size,
                  n_samples=FLAGS.n_samples,
                  random_state=FLAGS.random_state)
```

In the preceding code segment, Preprocessing is a class that creates (see `data_preparation.py` for details) several functions and constructors that help us preprocess the training and test sets to train the LSTM network. Here I provide the code of each function with their functionality. The constructor of this class initializes data preprocessor. This class provides an interface to load, preprocess, and split data into train, validation, and test sets. It takes the following parameters:

- `data_dir`: Data directory containing the dataset file data.csv with columns SentimentText and Sentiment.
- `stopwords_file`: Optional. If provided, discards each stop word from original data.
- `sequence_len`: Optional. Let m be the maximum sequence length in the dataset. Then, it's required that `sequence_len >= m`. If `sequence_len` is None, then it'll be automatically assigned to `m`.
- `n_samples`: Optional. Number of samples to load from the dataset (useful for large datasets). If `n_samples` is None, then the whole dataset will be loaded (be careful, if a dataset is large it may take a while to preprocess every sample).
- `test_size`: Optional. `0<test_size<1`. Represents the proportion of the dataset to include in the test split (default is `0.2`).
- `val_samples`: Optional, but can be used to represent the absolute number of validations samples (default is `100`).
- `random_state`: Is an optional parameter for the random seed used for splitting data into train, test, and validation sets (default is `0`).
- `ensure_preprocessed`: Is optional. If `ensure_preprocessed=True`, ensures that the dataset is already preprocessed (default is `False`).

The code for the constructor is given as follows:

```
def __init__(self, data_dir, stopwords_file=None, sequence_len=None,
n_samples=None, test_size=0.2, val_samples=100, random_state=0,
ensure_preprocessed=False):
        self._stopwords_file = stopwords_file
        self._n_samples = n_samples
        self.sequence_len = sequence_len
        self._input_file = os.path.join(data_dir, 'data.csv')
        self._preprocessed_file=os.path.join(data_
dir,"preprocessed_"+str(n_samples)+ ".npz")
        self._vocab_file = os.path.join(data_dir, "vocab_" + str(n_
samples) + ".pkl")
        self._tensors = None
        self._sentiments = None
```

```
            self._lengths = None
            self._vocab = None
            self.vocab_size = None

            # Prepare data
            if os.path.exists(self._preprocessed_file) and os.path.
    exists(self._vocab_file):
                print('Loading preprocessed files ...')
                self.__load_preprocessed()
            else:
                if ensure_preprocessed:
                    raise ValueError('Unable to find preprocessed files.')
                print('Reading data ...')
                self.__preprocess()
            # Split data in train, validation and test sets
            indices = np.arange(len(self._sentiments))
            x_tv, self._x_test, y_tv, self._y_test, tv_indices, test_
    indices = train_test_split(
                self._tensors,
                self._sentiments,
                indices,
                test_size=test_size,
                random_state=random_state,
                stratify=self._sentiments[:, 0])
                self._x_train,self._x_val,self._y_train,self._y_val,train_
    indices,val_indices=
     train_test_split(x_tv,  y_tv,  tv_indices,  test_size=val_
    samples,random_state = random_state,
    stratify=y_tv[:, 0])
            self._val_indices = val_indices
            self._test_indices = test_indices
            self._train_lengths = self._lengths[train_indices]
            self._val_lengths = self._lengths[val_indices]
            self._test_lengths = self._lengths[test_indices]
            self._current_index = 0
            self._epoch_completed = 0
```

Now let's see the signature of the preceding method. We start with the `_preprocess()` method, which loads data from `data_dir/data.csv`, preprocesses each sample loaded, and stores intermediate files to avoid preprocessing later. The workflow goes as follows:

1. Loads data.
2. Cleans samples text.

3. Prepares vocabulary dictionary.
4. Remove the most uncommon words (they're probably grammar mistakes), encodes samples into tensors, and pads each tensor with zeros according to `self.sequence_len`.
5. Saves intermediate files.
6. Stores samples' lengths:

```
 def __preprocess(self):
        data = pd.read_csv(self._input_file, nrows=self._n_
samples)
        self._sentiments = np.squeeze(data.as_
matrix(columns=['Sentiment']))
        self._sentiments = np.eye(2)[self._sentiments]
        samples = data.as_matrix(columns=['SentimentText'])[:, 0]
        samples = self.__clean_samples(samples)
        vocab = dict()
        vocab[''] = (0, len(samples))  # add empty word
        for sample in samples:
            sample_words = sample.split()
            for word in list(set(sample_words)):  # distinct words
in list
                value = vocab.get(word)
                if value is None:
                    vocab[word] = (-1, 1)
                else:
                    encoding, count = value
                    vocab[word] = (-1, count + 1)
        sample_lengths = []
        tensors = []
        word_count = 1
        for sample in samples:
            sample_words = sample.split()
            encoded_sample = []
            for word in list(set(sample_words)):  # distinct words
in list
                value = vocab.get(word)
                if value is not None:
                    encoding, count = value
                    if count / len(samples) > 0.0001:
                        if encoding == -1:
                            encoding = word_count
                            vocab[word] = (encoding, count)
                            word_count += 1
                        encoded_sample += [encoding]
```

```
                        else:
                            del vocab[word]
                    tensors += [encoded_sample]
                    sample_lengths += [len(encoded_sample)]
                self.vocab_size = len(vocab)
                self._vocab = vocab
                self._lengths = np.array(sample_lengths)
                self.sequence_len, self._tensors = self.__apply_to_
    zeros(tensors, self.sequence_len)
                with open(self._vocab_file, 'wb') as f:
                    pickle.dump(self._vocab, f)
                np.savez(self._preprocessed_file, tensors=self._tensors,
    lengths=self._lengths, sentiments=self._sentiments)
```

Next we invoke the preceding method and load intermediate files, avoiding data preprocessing:

```
def __load_preprocessed(self):
        with open(self._vocab_file, 'rb') as f:
            self._vocab = pickle.load(f)
        self.vocab_size = len(self._vocab)
        load_dict = np.load(self._preprocessed_file)
        self._lengths = load_dict['lengths']
        self._tensors = load_dict['tensors']
        self._sentiments = load_dict['sentiments']
        self.sequence_len = len(self._tensors[0])
```

Once we have the preprocessed dataset, the next task is to clean the samples. The workflow is as follows:

1. Prepare regex patterns.
2. Clean each sample.
3. Restore HTML characters.
4. Remove @users and urls.
5. Transform to lowercase.
6. Remove punctuation symbols.
7. Replace CC(C+) (a character that occurs more than twice in a row) with `C`.
8. Remove stop words:

```
def __clean_samples(self, samples):
        print('Cleaning samples ...')
        ret = []
        reg_punct = '[' + re.escape(''.join(string.punctuation)) +
']'
```

```
            if self._stopwords_file is not None:
                stopwords = self.__read_stopwords()
                sw_pattern = re.compile(r'\b(' + '|'.join(stopwords) +
    r')\b')
            for sample in samples:
                text = html.unescape(sample)
                words = text.split()
                words = [word for word in words if not word.
    startswith('@') and not word.startswith('http://')]
                text = ' '.join(words)
                text = text.lower()
                text = re.sub(reg_punct, ' ', text)
                text = re.sub(r'([a-z])\1{2,}', r'\1', text)
                if stopwords is not None:
                    text = sw_pattern.sub('', text)
                ret += [text]
            return ret
```

The `__apply_to_zeros ()` method returns `padding_length` used and NumPy array of padded tensors. First it finds the maximum length `m` and ensure that `m>=sequence_len`. Then it pads the list (that is, the list to be padded) with zeros according to `sequence_len`:

```
def __apply_to_zeros(self, lst, sequence_len=None):
inner_max_len = max(map(len, lst))
if sequence_len is not None:
            if inner_max_len > sequence_len:
                raise Exception(,Error: Provided sequence length is
not sufficient')
            else:
                inner_max_len = sequence_len
result = np.zeros([len(lst), inner_max_len], np.int32)
for i, row in enumerate(lst):
            for j, val in enumerate(row):
                result[i][j] = val
        return inner_max_len, result
```

Now the next task is to remove all the stop words (provided in the `data/StopWords.txt` file). This method returns the stop words list:

```
def __read_stopwords(self):
        if self._stopwords_file is None:
            return None
        with open(self._stopwords_file, mode='r') as f:
            stopwords = f.read().splitlines()
        return stopwords
```

The `next_batch ()` method takes `batch_size>0` as the number of samples that'll be included and returns batch size samples (`text_tensor`, `text_target`, and `text_length`) after completing the epoch, and randomly shuffles train samples:

```
def next_batch(self, batch_size):
        start = self._current_index
        self._current_index += batch_size
        if self._current_index > len(self._y_train):
            self._epoch_completed += 1
            ind = np.arange(len(self._y_train))
            np.random.shuffle(ind)
            self._x_train = self._x_train[ind]
            self._y_train = self._y_train[ind]
            self._train_lengths = self._train_lengths[ind]
            start = 0
            self._current_index = batch_size
        end = self._current_index
        return self._x_train[start:end], self._y_train[start:end],
self._train_lengths[start:end]
```

The next method called `get_val_data()` is then used to get the validation set to be used during the training period. It takes the original texts and returns the validation data. If `original_text` returns (`original_samples`, `text_tensor`, `text_target`, `text_length`), it otherwise returns (`text_tensor`, `text_target`, `text_length`):

```
def get_val_data(self, original_text=False):
        if original_text:
            data = pd.read_csv(self._input_file, nrows=self._n_
samples)
            samples = data.as_matrix(columns=['SentimentText'])[:, 0]
            return samples[self._val_indices], self._x_val, self._y_
val, self._val_lengths
        return self._x_val, self._y_val, self._val_lengths
```

Finally, we have an additional method called `get_test_data()`, which is used to prepare the test set to be used during the model evaluation period:

```
    def get_test_data(self, original_text=False):
        if original_text:
            data = pd.read_csv(self._input_file, nrows=self._n_
samples)
            samples = data.as_matrix(columns=['SentimentText'])[:, 0]
            return samples[self._test_indices], self._x_test, self._y_
test, self._test_lengths
        return self._x_test, self._y_test, self._test_lengths
```

Now we prepare the data so that it can be fed to the LSTM network:

```
lstm_model = LSTM_RNN_Network(hidden_size=[FLAGS.hidden_size],
                    vocab_size=data_lstm.vocab_size,
                    embedding_size=FLAGS.embedding_size,
                    max_length=data_lstm.sequence_len,
                    learning_rate=FLAGS.learning_rate)
```

In the preceding code segment, `LSTM_RNN_Network` is a class containing several functions and constructors that help us create the LSTM network. Here I provide the code of each function with their functionality. The following constructor builds a TensorFlow LSTM model. It takes the following parameters:

- `hidden_size`: Array holding number of units in LSTM cell of RNN layers
- `vocab_size`: Vocabulary size that appears in a sample
- `embedding_size`: Words will be encoded using a vector of this size
- `max_length`: Maximum length of an input tensor
- `n_classes`: Number of classification classes
- `learning_rate`: Learning rate of the RMSProp algorithm
- `random_state`: Random state for dropout

The code for the constructor is given as follows:

```
def __init__(self, hidden_size, vocab_size, embedding_size, max_
length, n_classes=2, learning_rate=0.01, random_state=None):
        # Build TensorFlow graph
        self.input = self.__input(max_length)
        self.seq_len = self.__seq_len()
        self.target = self.__target(n_classes)
        self.dropout_keep_prob = self.__dropout_keep_prob()
        self.word_embeddings = self.__word_embeddings(self.input,
vocab_size, embedding_size, random_state)
        self.scores = self.__scores(self.word_embeddings, self.seq_
len, hidden_size, n_classes, self.dropout_keep_prob,
                                    random_state)
        self.predict = self.__predict(self.scores)
        self.losses = self.__losses(self.scores, self.target)
        self.loss = self.__loss(self.losses)
        self.train_step = self.__train_step(learning_rate, self.loss)
        self.accuracy = self.__accuracy(self.predict, self.target)
        self.merged = tf.summary.merge_all()
```

The next function is called `_input()` and takes a parameter called `param max_length`, which is the maximum length of an input tensor. It then returns the input placeholder with shape [`batch_size`, `max_length`] for the TensorFlow computation:

```
    def __input(self, max_length):
        return tf.placeholder(tf.int32, [None, max_length],
name='input')
```

Next, the function `_seq_len()` returns a sequence length placeholder with shape [`batch_size`]. It holds each tensor's real length in a given batch, allowing a dynamic sequence length:

```
def __seq_len(self):
        return tf.placeholder(tf.int32, [None], name='lengths')
```

The next function that I am going to describe is called `_target()` and takes a parameter called `param n_classes`, which contains the number of classification classes. Finally, it returns the target placeholder with shape [`batch_size`, `n_classes`]:

```
def __target(self, n_classes):
        return tf.placeholder(tf.float32, [None, n_classes],
name='target')
The _dropout_keep_prob() returns a placeholder holding the dropout
keep probability:
def __dropout_keep_prob(self):
        return tf.placeholder(tf.float32, name='dropout_keep_prob')
```

`method _cell()` is used to build an LSTM cell with a dropout wrapper. It takes several parameters such as `hidden_size`, which is the number of units in the LSTM cell; `dropout_keep_prob`, which indicates the tensor holding the dropout keep probability; and seed, which is an optional value to ensure the reproducibility of the computation for the random state for the dropout wrapper. Finally, it returns an LSTM cell with a dropout wrapper:

```
def __cell(self, hidden_size, dropout_keep_prob, seed=None):
        lstm_cell = tf.nn.rnn_cell.LSTMCell(hidden_size, state_is_
tuple=True)
        dropout_cell = tf.nn.rnn_cell.DropoutWrapper(lstm_cell, \
input_keep_prob=dropout_keep_prob, output_keep_prob=dropout_keep_prob,
seed=seed)
        return dropout_cell
```

Once we have created the LSTM cells, we can create the embedding of the input tokens. For this, `__word_embeddings()` does the trick. It builds the embedding layer with shape [`vocab_size`, `embedding_size`] and the input parameters such as `x` as the input with shape [`batch_size`, `max_length`]; `vocab_size` is the vocabulary size–that is, number of possible words that may appear in a sample; `embedding_size` is the words that will be represented using a vector of this size; and seed is optional, but it ensures the random state for the embedding initialization. Finally, it returns the embedding lookup tensor with shape [`batch_size`, `max_length`, `embedding_size`]:

```
def __word_embeddings(self, x, vocab_size, embedding_size, seed=None):
        with tf.name_scope('word_embeddings'):
            embeddings = tf.Variable(tf.random_uniform([vocab_size,
embedding_size], -1, 1, seed=seed))
          embedded_words = tf.nn.embedding_lookup(embeddings, x)
        return embedded_words
```

The method `__rnn_layer()` creates the LSTM layer. It takes several inputs, such as `hidden_size` as the number of units in the LSTM cell, `x` as the input with shape, `seq_len` as the sequence length tensor with shape, `dropout_keep_prob` is the tensor holding the dropout keep probability, `variable_scope` is the name of the variable scope (the default layer is the `rnn_layer`), and `random_state` is the random state for the dropout wrapper. Finally, it returns the outputs with shape [`batch_size`, `max_seq_len`, `hidden_size`]:

```
def __rnn_layer(self, hidden_size, x, seq_len, dropout_keep_prob,
variable_scope=None, random_state=None):
        with tf.variable_scope(variable_scope, default_name='rnn_
layer'):
            lstm_cell = self.__cell(hidden_size, dropout_keep_prob,
random_state)
outputs, _ = tf.nn.dynamic_rnn(lstm_cell, x, dtype=tf.float32,
sequence_length=seq_len)
        return outputs
```

The following method, `_score()`, is used to build the LSTM layers and the final fully connected layer. It takes `embedded_words` as the embedding lookup tensor with shape [`batch_size`, `max_length`, `embedding_size`], `seq_len` as the sequence length tensor with shape [`batch_size`], `hidden_size` is an array holding the number of units in the LSTM cell of each `rnn` layer, `n_classes` is the number of classification classes, `dropout_keep_prob` is the tensor holding the dropout keep probability, and `random_state` is an optional parameter, but it can be used to ensure the random state for the dropout wrapper. Finally, it returns the linear activation of each class with shape [`batch_size`, `n_classes`]:

```
def __scores(self, embedded_words, seq_len, hidden_size, n_classes,
dropout_keep_prob, random_state=None):
```

```
            outputs = embedded_words
            for h in hidden_size:
                outputs = self.__rnn_layer(h, outputs, seq_len, dropout_
    keep_prob)
            outputs = tf.reduce_mean(outputs, axis=[1])
            with tf.name_scope('final_layer/weights'):
                w = tf.Variable(tf.truncated_normal([hidden_size[-1], n_
    classes], seed=random_state))
                self.variable_summaries(w, 'final_layer/weights')
            with tf.name_scope('final_layer/biases'):
                b = tf.Variable(tf.constant(0.1, shape=[n_classes]))
                self.variable_summaries(b, 'final_layer/biases')
            with tf.name_scope('final_layer/wx_plus_b'):
                scores = tf.nn.xw_plus_b(outputs, w, b, name='scores')
                tf.summary.histogram('final_layer/wx_plus_b', scores)
            return scores
```

The `_predict()` method take scores as the linear activation of each class with shape [`batch_size`, `n_classes`] and returns softmax activations with shape [`batch_size`, `n_classes`]:

```
    def __predict(self, scores):
            with tf.name_scope('final_layer/softmax'):
                softmax = tf.nn.softmax(scores, name='predictions')
                tf.summary.histogram('final_layer/softmax', softmax)
            return softmax
```

The `_losses()` method returns the cross-entropy losses with shape [`batch_size`]. It also takes two parameters, such as scores, as the linear activation of each class with shape [`batch_size`, `n_classes`] and the target tensor with shape [`batch_size`, `n_classes`]:

```
    def __losses(self, scores, target):
            with tf.name_scope('cross_entropy'):
                cross_entropy = tf.nn.softmax_cross_entropy_with_
    logits(logits=scores, labels=target, name='cross_entropy')
            return cross_entropy
```

The `_loss()` function computes and returns the cross-entropy loss mean. It takes only one parameter called losses, which indicates the cross-entropy losses with shape [`batch_size`] computed by the previous function:

```
    def __loss(self, losses):
            with tf.name_scope('loss'):
                loss = tf.reduce_mean(losses, name='loss')
                tf.summary.scalar('loss', loss)
            return loss
```

Now the `_train_step()` computes and returns the RMSProp train step operation. It does take two parameters such as `learning_rate`, which is the learning rate for the RMSProp optimizer and the cross-entropy loss mean computed using the previous function:

```
 def __train_step(self, learning_rate, loss):
        return tf.train.RMSPropOptimizer(learning_rate).minimize(loss)
```

When the performance evaluation comes, the `_accuracy()` function computes the accuracy of the classification. It takes two parameters such as predict as the softmax activations with shape [`batch_size`, `n_classes`] and the target tensor with shape [`batch_size`, `n_classes`] and the accuracy mean obtained in the current batch:

```
def __accuracy(self, predict, target):
        with tf.name_scope('accuracy'):
            correct_pred = tf.equal(tf.argmax(predict, 1),
tf.argmax(target, 1))
            accuracy = tf.reduce_mean(tf.cast(correct_pred,
tf.float32), name='accuracy')
            tf.summary.scalar('accuracy', accuracy)
        return accuracy
```

The very next function called `initialize_all_variable()` initializes all variables:

```
def initialize_all_variables(self):
        return tf.global_variables_initializer()
```

Finally, we have a static method called `variable_summaries()`, which attaches a lot of summaries to a tensor for TensorBoard visualization. It takes two parameters called `var`, which is the variable to summarize and bane of the summary name:

```
    @staticmethod
    def variable_summaries(var, name):
        with tf.name_scope('summaries'):
            mean = tf.reduce_mean(var)
            tf.summary.scalar('mean/' + name, mean)
            with tf.name_scope('stddev'):
                stddev = tf.sqrt(tf.reduce_mean(tf.square(var -
mean)))
            tf.summary.scalar('stddev/' + name, stddev)
            tf.summary.scalar('max/' + name, tf.reduce_max(var))
            tf.summary.scalar('min/' + name, tf.reduce_min(var))
            tf.summary.histogram(name, var)
```

Now we need to create a TensorFlow session before we can train the train model:

```
sess = tf.Session()
```

Now let's initialize all the variables:

```
init_op = tf.global_variables_initializer()
sess.run(init_op)
```

We then save the TensorFlow model for future use:

```
saver = tf.train.Saver()
```

Now let's prepare the training set:

```
x_val, y_val, val_seq_len = data_lstm.get_val_data()
```

Now we should write the logs of the TensorFlow graph computation:

```
train_writer.add_graph(lstm_model.input.graph)
```

Additionally, we can create some empty lists to hold the training loss, validation loss, and the steps so that we can see them graphically:

```
train_loss_list = []
val_loss_list = []
step_list = []
sub_step_list = []
step = 0
```

Now we start the training. In each step, we also record the training error and the validation errors are recorded in each sub-step:

```
for i in range(FLAGS.train_steps):
    x_train, y_train, train_seq_len = data_lstm.next_batch(FLAGS.
batch_size)
    train_loss, _, summary = sess.run([lstm_model.loss, lstm_model.
train_step, lstm_model.merged],
                                      feed_dict={lstm_model.input:
x_train,
                                                 lstm_model.target:
y_train,
                                                 lstm_model.seq_len:
train_seq_len,
                                                 lstm_model.dropout_
keep_prob: FLAGS.dropout_keep_prob})
    train_writer.add_summary(summary, i)  # Write train summary for
step i (TensorBoard visualization)
    train_loss_list.append(train_loss)
```

```
    step_list.append(i)
        print('{0}/{1} train loss: {2:.4f}'.format(i + 1, FLAGS.train_
steps, train_loss))
    # Check validation performance
    if (i + 1) % FLAGS.validate_every == 0:
        val_loss, accuracy, summary = sess.run([lstm_model.loss, lstm_
model.accuracy, lstm_model.merged],
                                               feed_dict={lstm_model.
input: x_val,
                                                          lstm_model.
target: y_val,
lstm_model.seq_len: val_seq_len,
lstm_model.dropout_keep_prob: 1})
        validation_writer.add_summary(summary, i)
        print('   validation loss: {0:.4f} (accuracy {1:.4f})'.
format(val_loss, accuracy))
        step = step + 1
        val_loss_list.append(val_loss)
        sub_step_list.append(step)

>>>
1/100 train loss: 0.6807
….
20/100 train loss: 0.6948
   validation loss: 0.6644 (accuracy 0.6400)
21/100 train loss: 0.6893
…
40/100 train loss: 0.6612
   validation loss: 0.6476 (accuracy 0.6800)
41/100 train loss: 0.6577
…
60/100 train loss: 0.6173
   validation loss: 0.6134 (accuracy 0.7300)
61/100 train loss: 0.5994
…
80/100 train loss: 0.5750
   validation loss: 0.5803 (accuracy 0.7600)
81/100 train loss: 0.5350
…
100/100 train loss: 0.5839
   validation loss: 0.5255 (accuracy 0.7700)
```

As we can see in the preceding code, it also prints the training and the validation error. When the training is over, the model will be saved to the checkpoint directory with a unique ID for the folder:

```
checkpoint_file = '{}/model.ckpt'.format(model_dir)
save_path = saver.save(sess, checkpoint_file)
print('Model saved in: {0}'.format(model_dir))
>>>
Model saved in: checkpoints/1505148083
```

The preceding checkpoint directory (example: `checkpoints/1505148083`) will produce at least three files, as follows:

- `'- config.pkl`: Contains parameters used to train the model
- `'- model.ckpt`: Contains the weights of the model
- `'- model.ckpt.meta`: Contains the TensorFlow graph definition

As a regular job, let's see how the training went; what were the training and the validation losses:

```
# Plot loss over time
plt.plot(step_list, train_loss_list, 'r--', label='LSTM training loss
per iteration', linewidth=4)
plt.title('LSTM training loss per iteration')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.legend(loc='upper right')
plt.show()

# Plot accuracy over time
plt.plot(sub_step_list, val_loss_list, 'r--', label='LSTM validation
loss per validating interval', linewidth=4)
plt.title('LSTM validation loss per validatin interval')
plt.xlabel('Validatin interval')
plt.ylabel('Validation loss')
plt.legend(loc='upper left')
plt.show()
>>>
```

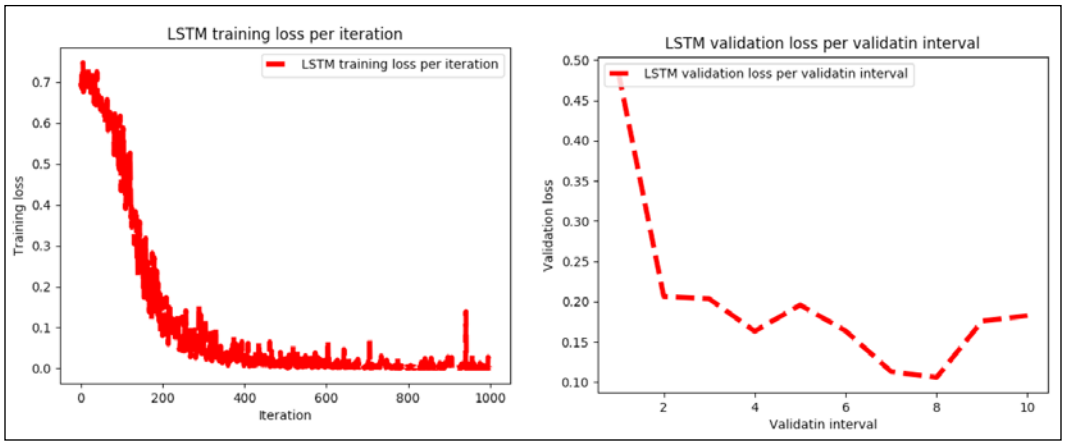


Figure 19: a) LSTM training loss per iteration, b) LSTM validation loss per validation interval

So from the preceding figure, it is clear that the training went pretty well in both the training phase and the validation phase with only 1000 steps. However, readers should increase the training steps and tune the hyperparameters and see how it goes.

Visualizing through TensorBoard

Now let's observe the TensorFlow computational graph on TensorBoard. Simply execute the following command and access TensorBoard at `<host_name>:6006`:

```
tensorboard --logdir /home/logs
```

The **GRAPH** tab shows the execution graph including the gradients used, `loss_op`, accuracy, final layer, optimizer used (in our case it's RMSPro), LSTM layer (that is, the RNN layer), embedding layer, `save_op`, and so on, as shown in figure 20:

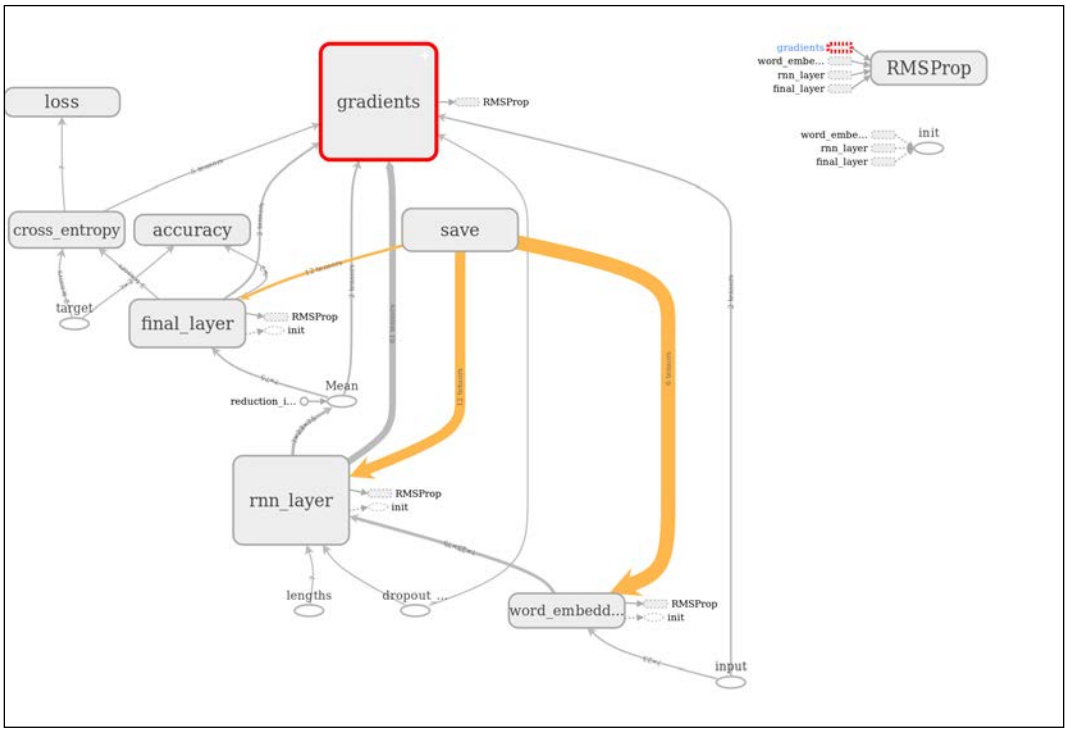


Figure 20: The execution graph on TensorBoard

The preceding graph shows that the computations we have done for this LSTM-based classifier for sentiment analysis are quite transparent. We can also observe the validation, training losses, accuracies, and the operations in the layers:

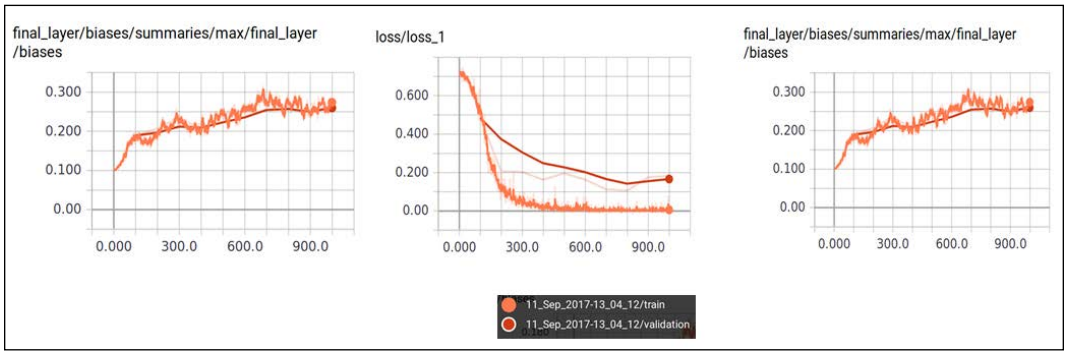


Figure 21: Validation, training losses, accuracies, and the operations in the layers on TensorBoard

LSTM model evaluation

Well, we have our LSTM model trained and saved. So we can easily restore the trained model and do some evaluation. The thing is that we prepare the test set and use the previously trained TensorFlow model to make predictions on it. Let's do that straight away. First, we load the required models:

```
import tensorflow as tf
from data_preparation import Preprocessing
import pickle
```

Then, we load to show the checkpoint directory where the model was saved. In our case, it was `checkpoints/1505148083`:



For this step, execute the predict.py script as follows:

$ python3 predict.py --checkpoints_dir checkpoints/1505148083

```
tf.flags.DEFINE_string('checkpoints_dir', 'checkpoints',
'checkpoints/1505148083')
FLAGS = tf.flags.FLAGS
if FLAGS.checkpoints_dir is None:
    raise ValueError('Please, a valid checkpoints directory is
required (--checkpoints_dir <file name>)')
```

Then we load the configuration:

```
with open('{}/config.pkl'.format(FLAGS.checkpoints_dir), 'rb') as f:
    config = pickle.load(f)
```

Now we load the test dataset and prepare the test set to evaluate the model:

```
data_lstm= Preprocessing(data_dir=config['data_dir'],
                 stopwords_file=config['stopwords_file'],
                 sequence_len=config['sequence_len'],
                 n_samples=config['n_samples'],
                 test_size=config['test_size'],
                 val_samples=config['batch_size'],
                 random_state=config['random_state'],
                 ensure_preprocessed=True)
```

The workflow for this evaluation method goes as follows:

1. Import the graph and evaluate the model using test data. First, prepare the test set.
2. Create a TensorFlow session for the computation.

3. Import the graph and restore its weights.
4. Recover input/output tensors.
5. Perform the prediction.
6. Finally, print the accuracy and the result on the simple test set.

Step 1 is already done. Now the remaining code does steps 2 to 5:

```
original_text, x_test, y_test, test_seq_len = data_lstm.get_test_
data(original_text=True)
graph = tf.Graph()
with graph.as_default():
    sess = tf.Session()
    print('Restoring graph ...')
    saver = tf.train.import_meta_graph("{}/model.ckpt.meta".
format(FLAGS.checkpoints_dir))
    saver.restore(sess, ("{}/model.ckpt".format(FLAGS.checkpoints_
dir)))
    input = graph.get_operation_by_name('input').outputs[0]
    target = graph.get_operation_by_name('target').outputs[0]
    seq_len = graph.get_operation_by_name('lengths').outputs[0]
    dropout_keep_prob = graph.get_operation_by_name('dropout_keep_
prob').outputs[0]
    predict = graph.get_operation_by_name('final_layer/softmax/
predictions').outputs[0]
    accuracy = graph.get_operation_by_name('accuracy/accuracy').
outputs[0]
    pred, acc = sess.run([predict, accuracy],
                         feed_dict={input: x_test,
                                    target: y_test,
                                    seq_len: test_seq_len,
                                    dropout_keep_prob: 1})
      print("Evaluation done.")

>>>
Restoring graph ...
Evaluation was done.
```

Well done! The training is finished. So let's print the results:

```
print('\nAccuracy: {0:.4f}\n'.format(acc))
for i in range(100):
    print('Sample: {0}'.format(original_text[i]))
    print('Predicted sentiment: [{0:.4f}, {1:.4f}]'.format(pred[i, 0],
pred[i, 1]))
```

```
        print('Real sentiment: {0}\n'.format(y_test[i]))

>>>
Accuracy: 0.9362
Sample: Brokeback Mountain is THE most amazing / beautiful / romantic
/ heartbraking movie i have ever or will ever see in my life......
Predicted sentiment: [0.0000, 1.0000]
Real sentiment: [ 0.  1.]
….
Sample: I love Harry Potter (the books are much better than the
movies).
Predicted sentiment: [0.0000, 1.0000]
Real sentiment: [ 0.  1.]
```

So the accuracy is above 93%. Not bad at all! However, you can try to iterate the training for higher iterations with the tuned hyperparameters and you might achieve even higher accuracy. I will leave this up to the readers.

Summary

LSTM networks are equipped with special hidden units, called memory cells, whose behavior is to remember the previous input for a long time. These cells take in input at each point in time, the previous state, and the current input of the network. Combining them with the current contents of the memory, and deciding using a gating mechanism by other units what to keep and what to delete from memory, LSTM has proved very useful and effective learning of long-term dependency.

In this chapter, we have discussed RNNs. We have seen how to make predictions in data with high temporal dependencies. We have seen how to develop several real-life predictive models for making predictive analytics easier using RNNs and their different architectural variants. We started with some theoretical background to RNNs. Then we have seen a few examples that showed a step-by-step way of implementing predictive models for image classification, sentiment analysis of movies, and product spam prediction for NLP. Finally, we have seen how to develop predictive models for time series data.

Factorization models are very popular in recommendation systems because they can be used to discover latent features underlying the interactions between two different kinds of entity. In *Chapter 10*, *Recommendation Systems for Predictive Analytics*, we will provide several examples of how to develop recommendation systems for predictive analytics. We will see some theoretical background to recommendation systems, such as matrix factorization used in developing recommendation systems. Finally, we will discuss how to develop a recommendation engine with factorization machine for predictive analytics.

10
Recommendation Systems for Predictive Analytics

Factorization models are very popular in recommendation systems because they can be used to discover latent features underlying the interactions between two different kinds of entities. In this chapter, we will provide several examples on how to develop recommendation systems for predictive analytics.

We will look at some theoretical background of recommendation systems, such as matrix factorization used in developing recommendation systems. Later in the chapter, we will look at how to use SVD and k-means for developing movie recommendation systems. Finally, we will look at how we can use Factorization Machines and it's improved versions for developing recommendation systems.

In summary, the following topics will be covered in this chapter:

- Recommendation systems
- Matrix factorization and the collaborative filtering approach for recommendation systems
- K-means for clustering similar movies
- Factorization machine-based recommendation systems

Recommendation systems

Recommendation systems (that is, recommendation engine—RE) is a subclass of information filtering system that helps predict the "rating" or "preference" based on the rating provided by users of an item. In recent years, recommendation systems have become increasingly popular.

Consequently, they're being used in many areas such as movies, music, news, books, research articles, search queries, social tags, products, collaborators, jokes, restaurants, garments, financial services, life insurance, and online dating sites.

There are a couple of ways to develop recommendation engines that typically produce a list of recommendations, for example, collaborative and content-based filtering or the personality-based approach.

Collaborative filtering approaches

Using collaborative filtering approaches, an RE can be built based on a user's past behavior, where numerical ratings are given on purchased items. Sometimes, it can be developed on similar decisions made by other users who also have purchased the same items. From the following figure, you can get some idea of different recommendation systems:

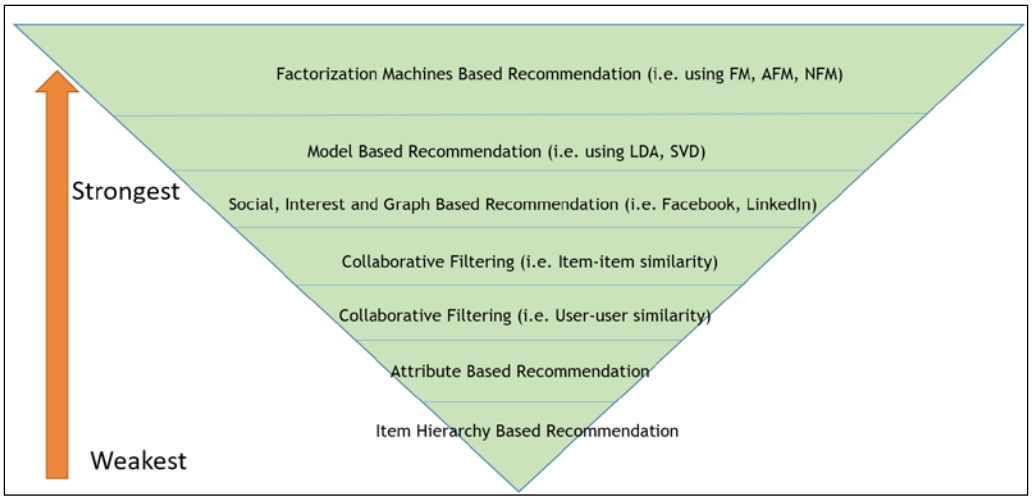


Figure 1: A comparative view of different recommendation systems

The Collaborative filtering-based approach often suffers from three problems: cold start, scalability, and sparsity:

- **Cold start**: Sometimes gets stuck when a large amount of data about users is required for making more accurate recommendation systems.
- **Scalability**: A large amount of computation power is often necessary to calculate recommendations out of a dataset with millions of users and products.

- **Sparsity**: Sparsity often happens with a crowd sourced dataset when a huge number of items are sold on major e-commerce sites. In such a case, active users may rate only a small subset of the whole items sold that is, even the most popular items have very few ratings. Accordingly, the user versus items matrix becomes very sparse. In other words, a large-scale sparse matrix cannot be handled.

To overcome these issues, a particular type of collaborative filtering algorithm uses matrix factorization, a low-rank matrix approximation technique. We will look at an example later in this chapter.

Content-based filtering approaches

Using a content-based filtering approach, a series of discrete characteristics of an item is utilized to recommend additional items with similar properties. Sometimes, it is based on a description of the item and a profile of the user's preferences. These approaches try to recommend items that are similar to those that a user liked in the past or those being used currently.

A key issue with content-based filtering is whether the system is able to learn user preferences from a user's actions regarding one content source and use them across other content types. When this type of RE is deployed, it can be used to predict items or ratings for items that the user may have an interest in.

Hybrid recommendation systems

As you have seen, there are several pros and cons of using collaborative filtering and content-based filtering approaches. Therefore, to overcome the limitations of these two approaches, the recent trend has shown that a hybrid approach can be more effective and accurate by combining collaborative filtering and content-based filtering. Sometimes, a factorization approach, such as **matrix factorization** (**FM**) and **singular value decomposition** (**SVD**) are used to make them robust. A hybrid approach can be implemented in several ways:

- Firstly, the content-based and collaborative-based predictions are computed separately and are later combined together that is, a unification of these two into one model. In this approach, often, FM and SVD are used extensively.
- Secondly, adding content-based capabilities to a collaborative-based approach or vice versa. Again, FM and SVD are used for better prediction.

Netflix is a good example that uses this hybrid approach for making a recommendation to its subscribers. This site makes recommendations in two ways:

- **Collaborative filtering**: By comparing the watching and searching habits of similar users
- **Content-based filtering**: By offering movies that share characteristics with films that a user has rated highly

Model-based collaborative filtering

Collaborative filtering methods are classified as memory-based that is, a user-based algorithm and model-based collaborative filtering that is kernel-mapping recommended. In the model-based collaborative filtering technique, users and products are described by a small set of factors, also called **latent factors** (**LFs**).

LFs are then used for predicting the missing entries. The **alternating least squares** (**ALS**) algorithm is used to learn these latent factors. From a computational perspective, model-based collaborative filtering is commonly used in many companies such as Netflix for a real-time movie recommendation.

Collaborative filtering approach for movie recommendations

In this section, we will see how to utilize SVD and collaborative filtering for developing a recommendation engine. First, let me introduce a model for recommendation systems, based on a utility matrix of preferences.

The utility matrix

In hybrid recommendation systems, there are two classes of entities: users and items, for example, movies, products and so on. Now, as a user, you might have preferences for certain items. Therefore, these preferences must be provoked out of the data about items, users, or ratings. Often, this data is represented as a utility matrix, such as user-item pair. This type of value can represent what is known about the degree of preference of that user for a particular item.

The entry in the matrix, that is, table can come from an ordered set. For example, integers 1–5 can be used to represent the number of stars that the user gave as rating for items. We have argued that often users might have rated items that are "unknown." This also means that the matrix might be sparse. An unknown rating implies that we have no explicit information about the user's preference for the item.

Table 1 shows an example of a utility matrix. The matrix represents the ratings of users for movies on a 1–5 scale, 5 being the highest rating. A blank entry represents that no users have provided any rating for those movies. **HP1**, **HP2**, and **HP3** are acronyms for the movies Harry Potter I, II, and III, **TW** for Twilight, and **SW1**, **SW2**, and **SW3** for Star Wars episodes 1, 2, and 3. The users are represented by capital letters A, B, C, and D:

	HP1	HP2	HP3	TW	SW1	SW2	SW3
A	4			5	1		
B	5	5	4				
C				2	4	5	
D		3					3

Table 1: A utility matrix (user versus movies matrix)

There are many blank entries for the user-movie pairs. This means that users have not rated these movies. In a real-life scenario, the matrix might be even sparser, that is, with the typical user rating only a tiny fraction of all available movies. Now, using this matrix, the goal is to predict the blanks in the utility matrix. Let's look at an example. Suppose we were curious to know whether user A would like **SW2**; however, it's really difficult to determine since there is little evidence in the matrix in table 1.

Thus, in practice, we might develop a movie recommendation engine to consider their uncommon properties of movies, such as producer name, director name, lead stars, or even the similarity of their names. This way, we can compute the similarity of movies **SW1** and **SW2**. This similarity would drive us to conclude that since A did not like **SW1**, they were unlikely to enjoy **SW2** as well.

However, this might not work for a larger dataset. Therefore, with more data, we might observe that the people who rated both **SW1** and **SW2** were inclined to give them similar ratings. Finally, we can conclude that A would also give **SW2** a low rating, similar to **A**'s rating of **SW1**.

In the next section, we will see how to develop a movie recommendation engine using an SVD algorithm for collaborative filtering, where we will see how to utilize these types of matrixes. The workflow is as follows:

1. First, we will train a model using available ratings.
2. Next, we will use the trained model to predict a missing rating in the user's versus movies matrix.
3. Then, with all the predicted ratings, a new user's versus movies matrix will be constructed and saved in the form of a `.pkl` file.
4. Next, we use this matrix for making a prediction of rating for particular users.
5. Finally, we will train the k-means model to cluster related movies.

Dataset description

Now, before we start implementing the movie recommendation engine, let's see some insights of the dataset that will be used. The 1m MovieLens dataset was downloaded from the MovieLens website at `http://files.grouplens.org/datasets/movielens/ml-1m.zip`.

I sincerely acknowledge and thank F. Maxwell Harper and Joseph A. Konstan for making the dataset. The dataset was published in *MovieLens Dataset: History and Context. ACM Transactions on Interactive Intelligent Systems (TiiS) 5, 4, Article 19 (December 2015), 19 pages*.

There are three files in the dataset about movies, ratings, and users. These files contain 1,000,209 anonymous ratings of approximately 3,900 movies made by 6,040 MovieLens users who joined MovieLens in 2000.

Ratings data

All ratings are contained in the `ratings.dat` file and are in the following format:

```
UserID::MovieID::Rating::Timestamp
```

- `UserID` ranges between 1 and 6040
- `MovieID` ranges between 1 and 3952
- `Rating` is made on a 5-star scale
- `Timestamp` is represented in seconds

Note that each user has rated at least 20 movies.

Movies data

Movie information is in the `movies.dat` file and is in the following format:

```
MovieID::Title::Genres
```

- Titles are identical to titles provided by IMDb (with the release year)
- Genres are pipe-separated (that is `::`), where each movie is categorized as action, adventure, animation, children's, comedy, crime, drama, war, documentary, fantasy, film-noir, horror, musical, mystery, romance, Sci-Fi, thriller, and western.5

User data

User information is in the `users.dat` file and is in the following format:

```
UserID::Gender::Age::Occupation::Zip-code
```

All demographic information is provided voluntarily by the users and is not checked for accuracy. Only users who have provided some demographic information are included in this dataset:

- Gender is denoted by an `M` for male and `F` for female
- Age is chosen from the following ranges:
    - 1: "Under 18"
    - 18: "18-24"
    - 25: "25-34"
    - 35: "35-44"
    - 45: "45-49"
    - 50: "50-55"
    - 56: "56+"
- Occupation is chosen from the following choices:
    - 0: "other" or not specified
    - 1: "academic/educator"
    - 2: "artist"
    - 3: "clerical/admin"
    - 4: "college/grad student"
    - 5: "customer service"
    - 6: "doctor/health care"
    - 7: "executive/managerial"
    - 8: "farmer"
    - 9: "homemaker"
    - 10: "K-12 student"
    - 11: "lawyer"
    - 12: "programmer"
    - 13: "retired"
    - 14: "sales/marketing"

- 15: "scientist"
- 16: "self-employed"
- 17: "technician/engineer"
- 18: "tradesman/craftsman"
- 19: "unemployed"
- 20: "writer"

Exploratory analysis of the dataset

Here, we will look at some exploratory descriptions of the dataset before we start developing the recommendation engine. Execute the `python3 eda.py` command for this section:

1. First, let's import the required libraries and packages:

```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
```

2. Now, load the users, ratings, and movies dataset, and create a pandas DataFrame:

```
ratings_list = [i.strip().split("::") for i in open('ratings.dat',
'r').readlines()]
users_list = [i.strip().split("::") for i in open('users.dat',
'r').readlines()]
movies_list = [i.strip().split("::") for i in open('movies.dat',
'r',encoding='latin-1').readlines()]
ratings_df = pd.DataFrame(ratings_list, columns = ['UserID',
'MovieID', 'Rating', 'Timestamp'], dtype = int)
movies_df = pd.DataFrame(movies_list, columns = ['MovieID',
'Title', 'Genres'])
user_df=pd.DataFrame(users_list, columns=['UserID','Gender','Age',
'Occupation','ZipCode'])
```

3. The next task is to convert the categorical columns, such as `MovieID`, `UserID`, and `Age` to numeric using the pandas built-in `to_numeric()` function:

```
movies_df['MovieID'] = movies_df['MovieID'].apply(pd.to_numeric)
user_df['UserID'] = user_df['UserID'].apply(pd.to_numeric)
user_df['Age'] = user_df['Age'].apply(pd.to_numeric)
```

4. Now, let's look at some samples from the user table:

```
print("User table description:")
print(user_df.head())
print(user_df.describe())
>>>
User table description:
   UserID Gender  Age Occupation Zip Code
0       1      F    1         10   48067
1       2      M   56         16   70072
2       3      M   25         15   55117
3       4      M   45          7   02460
4       5      M   25         20   55455
            UserID          Age
count  6040.000000  6040.000000
mean   3020.500000    30.639238
std    1743.742145    12.895962
min       1.000000     1.000000
25%    1510.750000    25.000000
50%    3020.500000    25.000000
75%    4530.250000    35.000000
max    6040.000000    56.000000
```

5. Now, let's look at some information from the rating table, that is, rating dataset:

```
print("Rating table description:")
print(ratings_df.head())
print(ratings_df.describe())
>>>
Rating table description:
   UserID  MovieID  Rating  Timestamp
0       1     1193       5  978300760
1       1      661       3  978302109
2       1      914       3  978301968
3       1     3408       4  978300275
4       1     2355       5  978824291
             UserID       MovieID        Rating     Timestamp
count  1.000209e+06  1.000209e+06  1.000209e+06  1.000209e+06
mean   3.024512e+03  1.865540e+03  3.581564e+00  9.722437e+08
std    1.728413e+03  1.096041e+03  1.117102e+00  1.215256e+07
min    1.000000e+00  1.000000e+00  1.000000e+00  9.567039e+08
25%    1.506000e+03  1.030000e+03  3.000000e+00  9.653026e+08
50%    3.070000e+03  1.835000e+03  4.000000e+00  9.730180e+08
75%    4.476000e+03  2.770000e+03  4.000000e+00  9.752209e+08
max    6.040000e+03  3.952000e+03  5.000000e+00  1.046455e+09
```

6. Finally, let's look at some information from the movie dataset:

```
>>>
print("Movies table description:")
print(movies_df.head())
print(movies_df.describe())
>>>
Movies table description:
   MovieID                                Title
Genres
0        1                    Toy Story (1995)
Animation|Children's|Comedy
1        2                      Jumanji (1995)
Adventure|Children's|Fantasy
2        3             Grumpier Old Men (1995)
Comedy|Romance
3        4            Waiting to Exhale (1995)
Comedy|Drama
4        5  Father of the Bride Part II (1995)
Comedy
MovieID
count  3883.000000
mean   1986.049446
std    1146.778349
min       1.000000
25%     982.500000
50%    2010.000000
75%    2980.500000
max    3952.000000
```

7. Now, let's look at the top five most rated movies:

```
print("Top Five most rated movies:")
print(ratings_df['MovieID'].value_counts().head())
>>>
Top 10 most rated movies with title
Title
American Beauty (1999)                                3428
Star Wars: Episode IV - A New Hope (1977)             2991
Star Wars: Episode V - The Empire Strikes Back (1980) 2990
Star Wars: Episode VI - Return of the Jedi (1983)     2883
Jurassic Park (1993)                                  2672
Saving Private Ryan (1998)                            2653
Terminator 2: Judgment Day (1991)                     2649
Matrix, The (1999)                                    2590
Back to the Future (1985)                             2583
Silence of the Lambs, The (1991)                      2578
dtype: int64
```

8. Now, let's look at the movie rating distribution. For this, let's use a histogram plot:

```
plt.hist(ratings_df.groupby(['MovieID'])['Rating'].mean().sort_
values(axis=0, ascending=False))
plt.title("Movie rating Distribution")
plt.ylabel('Count of movies')
plt.xlabel('Rating');
plt.show()
>>>
```

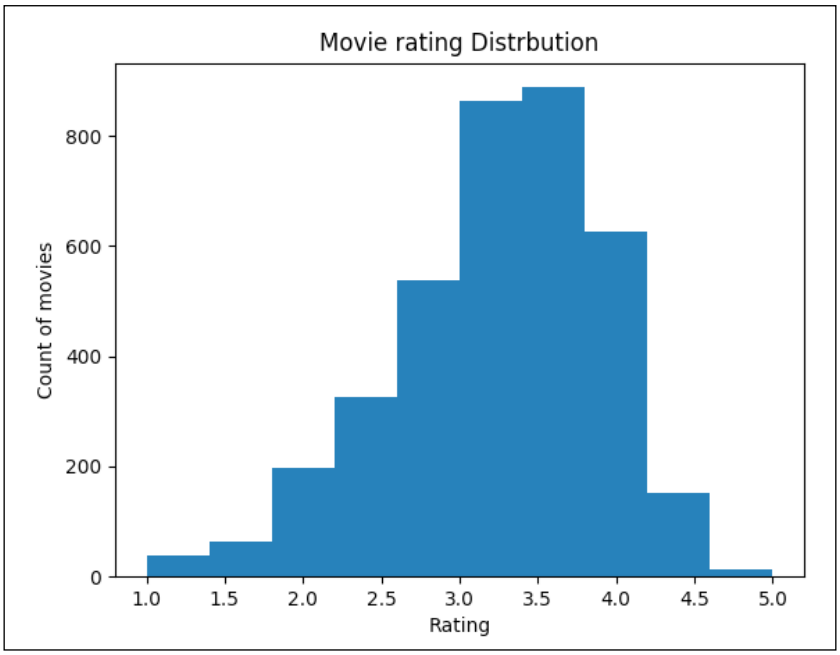


Figure 3: Movie rating distribution

9. Now, let's look at how the ratings are distributed across different age groups:

```
user_df.Age.plot.hist()
plt.title("Distribution of users (by ages)")
plt.ylabel('Count of users')
plt.xlabel('Age');
plt.show()
>>>
```

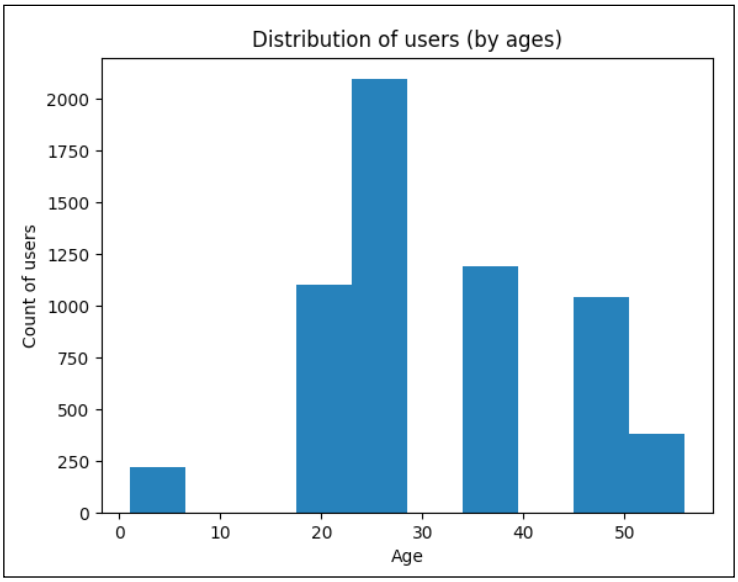


Figure 4: Distribution of users by ages

10. Now, let's look at the highest rated movie with minimum 150 ratings:

```
movie_stats = df.groupby('Title').agg({'Rating': [np.size,
np.mean]})
print("Highest rated movie with minimum 150 ratings")
print(movie_stats.Rating[movie_stats.Rating['size'] > 150].sort_
values(['mean'],ascending=[0]).head())
>>>
Top 5 and highest rated movie with minimum 150 ratings
-------------------------------------------------------------
Title                    size        mean
Seven Samurai (The Magnificent Seven) (Shichini...    628   4.560510
Shawshank Redemption, The (1994)     2227   4.554558
Godfather, The (1972)                                 2223   4.524966
Close Shave, A (1995)                                  657   4.520548
Usual Suspects, The (1995)                            1783   4.517106
```

11. Now, let's look at how the movies are rated by males and females that is, gender biasing toward movie rating:

```
>>>
pivoted = df.pivot_table(index=['MovieID', 'Title'],
columns=['Gender'], values='Rating',fill_value=0)
print("Gender biasing towards movie rating")
print(pivoted.head())
>>>
Gender biasing toward movie rating
Gender                                                F         M
MovieID Title
1       Toy Story (1995)                     4.187817  4.130552
2       Jumanji (1995)                       3.278409  3.175238
3       Grumpier Old Men (1995)              3.073529  2.994152
4       Waiting to Exhale (1995)             2.976471  2.482353
5       Father of the Bride Part II (1995)   3.212963  2.888298
>>>
```

12. Now, let's look at how the movies are rated by males and females that is, gender biasing toward movie rating and their difference:

```
>>>
pivoted['diff'] = pivoted.M - pivoted.F
print(pivoted.head())
>>>
Gender                                                F         M
diff
MovieID Title
1       Toy Story (1995)                     4.187817  4.130552
-0.057265
2       Jumanji (1995)                       3.278409  3.175238
-0.103171
3       Grumpier Old Men (1995)              3.073529  2.994152
-0.079377
4       Waiting to Exhale (1995)             2.976471  2.482353
-0.494118
5   Father of the Bride Part II (1995)  3.212963  2.888298
-0.324665
```

So, from the preceding output, it's clear that in most cases, males provided higher ratings than females. Now that we have seen some information and statistics about the dataset, it's time to build our TensorFlow recommendation model.

Implementing a movie recommendation engine

Through the recommendation engine, I will show how to recommend top `k` movies (where `k` = is the number of movies), predicted user rating, top k similar items (where `k` = number of items); we will also look at how to compute the user similarity. Then, we will look at the item-item correlation and user-user correlation using the Pearson correlation algorithm. Finally, we will look at clustering similar movies using the k-means algorithm.

In other words, it will be a movie recommendation engine using an SVD algorithm for collaborative filtering and k-means for clustering similar movies. To conclude, here's the workflow that will be used for developing this model:

1. First, train a model using available ratings.
2. Use this trained model to predict missing ratings in users versus movies matrix.
3. With all predicted rating, now users versus movies matrix becomes trained users versus movies matrix, and we save both in the form of a `.pkl` file.
4. Later, we use any user versus items/movies matrix or trained user's versus movies matrix by a Trained argument, for further processing.

Before training the model, the very first job is to prepare the training set utilizing the entire available dataset.

Training the model with available ratings

For this section, use the `train.py` script, which is again dependent on other scripts but we will see the dependencies:

1. First, let's import the necessary packages and modules:

```
# Imports for data io operations
from collections import deque
from six import next
import readers
import os
# Main imports for training
import tensorflow as tf
import numpy as np
import model as md
import pandas as pd
# Evaluate train times per epoch
```

```
import time
# For plotting
import matplotlib.pyplot as plt
```

2. Now, we set the random seed for reproducibility:

```
np.random.seed(12345)
```

3. The next task is to define the training parameters. First, let's define the required data parameters, such as the location of the ratings dataset, batch size, dimension of SVD, maximum epochs, and checkpoint directory:

```
tf.flags.DEFINE_string("data_file", "ratings.dat", "Input user-
movie-rating information file")
# Eval Parameters
tf.flags.DEFINE_integer("batch_size", 100, "Batch Size (default:
100)")
tf.flags.DEFINE_integer("dims", 15, "Dimensions of SVD (default:
15)")
tf.flags.DEFINE_integer("max_epochs", 25, "Dimensions of SVD
(default: 25)")
tf.flags.DEFINE_string("checkpoint_dir", "save/", "Checkpoint
directory from training run")
tf.flags.DEFINE_boolean("val", True, "True if Folders with files
and False if single file")
tf.flags.DEFINE_boolean("is_gpu", True, "Want to train model with
GPU")
```

4. We also need some other parameters such as allowing soft placement if you want to allow the device soft device placement and log device placement if you want to place the logs of operations on devices:

```
tf.flags.DEFINE_boolean("allow_soft_placement", True, "Allow
device soft device placement")
tf.flags.DEFINE_boolean("log_device_placement", False, "Log
placement of ops on devices")
```

5. Then, we initialize the flags:

```
FLAGS = tf.flags.FLAGS
FLAGS._parse_flags()
 Now, let's look at the parameters we have set (default):
print("\nParameters:")
for attr, value in sorted(FLAGS.__flags.items()):
    print("{}={}".format(attr.upper(), value))
print("")
>>>
Parameters:
ALLOW_SOFT_PLACEMENT=True
```

```
BATCH_SIZE=100
CHECKPOINT_DIR=save/
DATA_FILE=ratings.dat
DIMS=15
IS_GPU=True
LOG_DEVICE_PLACEMENT=False
MAX_EPOCHS=25
VAL=True
```

6. Now that we don't want to contaminate our fresh training with the old metadata, checkpoint, and model files, let's remove any such existing files:

```
print("Start removing previous Files ...")
if os.path.isfile("model/user_item_table.pkl"):
    os.remove("model/user_item_table.pkl")
if os.path.isfile("model/user_item_table_train.pkl"):
    os.remove("model/user_item_table_train.pkl")
if os.path.isfile("model/item_item_corr.pkl"):
    os.remove("model/item_item_corr.pkl")
if os.path.isfile("model/item_item_corr_train.pkl"):
    os.remove("model/item_item_corr_train.pkl")
if os.path.isfile("model/user_user_corr.pkl"):
    os.remove("model/user_user_corr.pkl")
if os.path.isfile("model/user_user_corr_train.pkl"):
    os.remove("model/user_user_corr_train.pkl")
if os.path.isfile("model/clusters.csv"):
    os.remove("model/clusters.csv")
if os.path.isfile("model/val_error.pkl"):
    os.remove("model/val_error.pkl")
print("Done ...")
>>>
Start removing previous Files ...
Done ...
Number of train samples 750156, test samples 250053, samples per
batch 7501
```

7. Now, let's define the checkpoint directory. TensorFlow assumes this directory already exists, so we need to create it:

```
checkpoint_prefix = os.path.join(FLAGS.checkpoint_dir, "model")
if not os.path.exists(FLAGS.checkpoint_dir):
    os.makedirs(FLAGS.checkpoint_dir)
```

8. Now, before getting the data, let's set the number of samples per batch, the dimension of the data, and number of times the network sees all the training data as follows:

```
batch_size = FLAGS.batch_size
dims = FLAGS.dims
max_epochs = FLAGS.max_epochs
```

9. Now, let's specify devices to be used for all TensorFlow computations that is, CPU or GPU:

```
if FLAGS.is_gpu:
    place_device = "/gpu:0"
else:
    place_device="/cpu:0"
```

10. Now, let's read the rating file with the delimiter `::` through the following `get_data()` function.

    I am assuming that you have already downloaded the necessary files and placed them in the `data_file` directory. Alternatively, download the MovieLens data from `http://files.grouplens.org/datasets/movielens/ml-1m.zip`. A sample column consists of user ID, item ID, rating, and timestamp, for example, `3::1196::4::978297539`.

    Then, it does the purely integer-location-based indexing for selection by position. After this, it splits the data into train and test, 75% for train and 25% for test. Finally, it uses the indices to separate the data and returns the dataframe to be used for training:

```
def get_data():
    print("Inside get data ...")
    df = readers.read_file(FLAGS.data_file, sep="::")
    rows = len(df)
    df = df.iloc[np.random.permutation(rows)].reset_
index(drop=True)
    split_index = int(rows * 0.75)
    df_train = df[0:split_index]
    df_test = df[split_index:].reset_index(drop=True)
    print("Done !!!")
    print(df.shape)
    return df_train, df_test,df['user'].max(),df['item'].max()
```

11. We then clip the limit of the values in an array; given an interval, values outside the interval are clipped to the interval edges. For example, if an interval of [`0`, `1`] is specified, values smaller than `0` become `0`, and values larger than `1` become `1`:

```
def clip(x):
    return np.clip(x, 1.0, 5.0)
```

12. We then invoke the `read_data()` method to read data from the ratings file to build a TF model:

```
df_train, df_test,u_num,i_num = get_data()
>>>
Inside get data ...
Done !!!
```

13. We then define the number of users in the dataset who rated the movies and number of movies in the dataset:

```
u_num = 6040 # Number of users in the dataset
i_num = 3952 # Number of movies in the dataset
Now, let's generate the samples per batch:
samples_per_batch = len(df_train) // batch_size
print("Number of train samples %d, test samples %d, samples per
batch %d" % (len(df_train), len(df_test), samples_per_batch"))
>>>
Number of train samples 750156, test samples 250053, samples per
batch 7501
```

14. Now using a shuffle iterator, we generate random batches for training; this helps prevent biased results:

```
iter_train = readers.ShuffleIterator([df_train["user"], df_train
["item"],                              df_train["rate"]], batch_
size=batch_size)
```

15. Now, for more on this class, refer to the `readers.py` script. For your ease, here I have provided the source of this class:

```
class ShuffleIterator(object):

    def __init__(self, inputs, batch_size=10):
        self.inputs = inputs
        self.batch_size = batch_size
        self.num_cols = len(self.inputs)
        self.len = len(self.inputs[0])
        self.inputs = np.transpose(np.vstack([np.array(self.
inputs[i]) for i in range(self.num_cols)]))
    def __len__(self):
        return self.len
```

```
def __iter__(self):
        return self
   def __next__(self):
        return self.next()
    def next(self):
        ids = np.random.randint(0, self.len, (self.batch_size,))
        out = self.inputs[ids, :]
        return [out[:, i] for i in range(self.num_cols)]
```

16. Next, we sequentially generate one-epoch batches for testing (refer to `readers.py`):

```
iter_test = readers.OneEpochIterator([df_test["user"], df_
test["item"],df_test["rate"]], batch_size=-1)
```

17. Now, for more on this class, refer to the `readers.py` script. For your ease, here I have provided the source of this class:

```
class OneEpochIterator(ShuffleIterator):
   def __init__(self, inputs, batch_size=10):
        super(OneEpochIterator, self).__init__(inputs, batch_
size=batch_size)
        if batch_size > 0:
            self.idx_group = np.array_split(np.arange(self.len),
np.ceil(self.len / batch_size))
        else:
            self.idx_group = [np.arange(self.len)]
        self.group_id = 0
   def next(self):
        if self.group_id >= len(self.idx_group):
            self.group_id = 0
            raise StopIteration
        out = self.inputs[self.idx_group[self.group_id], :]
        self.group_id += 1
        return [out[:, i] for i in range(self.num_cols)]
```

18. Now, it's time to create the TensorFlow placeholders:

```
user_batch = tf.placeholder(tf.int32, shape=[None], name="id_
user")
item_batch = tf.placeholder(tf.int32, shape=[None], name="id_
item")
rate_batch = tf.placeholder(tf.float32, shape=[None])
```

19. Now that our training set and placeholders are ready to hold the batches of training values, it is time to instantiate the model. For this, we use the `model()` method implemented and use the `l2` regularizations to avoid overfitting (refer to the `models.py` script):

```
infer, regularizer = md.model(user_batch, item_batch, user_num=u_
num, item_num=i_num, dim=dims, device=place_device)
```

20. The `model()` method can be seen as follows:

```
def model(user_batch, item_batch, user_num, item_num, dim=5,
device="/cpu:0"):
    with tf.device("/cpu:0"):
        # Using a global bias term
        bias_global = tf.get_variable("bias_global", shape=[])
        # User and item bias variables: get_variable: Prefixes the
name with the current variable scope and performs reuse checks.
        w_bias_user = tf.get_variable("embd_bias_user",
shape=[user_num])
        w_bias_item = tf.get_variable("embd_bias_item",
shape=[item_num])
        # embedding_lookup: Looks up 'ids' in a list of embedding
tensors
        # Bias embeddings for user and items, given a batch
        bias_user = tf.nn.embedding_lookup(w_bias_user, user_
batch, name="bias_user")
        bias_item = tf.nn.embedding_lookup(w_bias_item, item_
batch, name="bias_item")
        # User and item weight variables
        w_user = tf.get_variable("embd_user", shape=[user_num,
dim],
                                 initializer=tf.truncated_normal_
initializer(stddev=0.02))
        w_item = tf.get_variable("embd_item", shape=[item_num,
dim],
                                 initializer=tf.truncated_normal_
initializer(stddev=0.02))
        # Weight embeddings for user and items, given a batch
        embd_user = tf.nn.embedding_lookup(w_user, user_batch,
name="embedding_user")
        embd_item = tf.nn.embedding_lookup(w_item, item_batch,
name="embedding_item")
    with tf.device(device):
        # reduce_sum: Computes the sum of elements across
dimensions of a tensor
        infer = tf.reduce_sum(tf.multiply(embd_user, embd_item),
1)
        infer = tf.add(infer, bias_global)
```

```
            infer = tf.add(infer, bias_user)
            infer = tf.add(infer, bias_item, name="svd_inference")
            # l2_loss: Computes half the L2 norm of a tensor without
    the sqrt
            regularizer = tf.add(tf.nn.l2_loss(embd_user), tf.nn.l2_
    loss(embd_item),                    name="svd_regularizer")
        return infer, regularizer
```

21. Now, let's define the training ops (for more, refer to the `models.py` script):

```
_, train_op = md.loss(infer, regularizer, rate_batch, learning_
rate=0.001, reg=0.05, device=place_device)
The loss() method is given as follows:
def loss(infer, regularizer, rate_batch, learning_rate=0.1,
reg=0.1, device="/cpu:0"):
    with tf.device(device):
        cost_l2 = tf.nn.l2_loss(tf.subtract(infer, rate_batch))
        penalty = tf.constant(reg, dtype=tf.float32, shape=[],
name="l2")
        cost = tf.add(cost_l2, tf.multiply(regularizer, penalty))
        train_op = tf.train.FtrlOptimizer(learning_rate).
minimize(cost)
    return cost, train_op
```

Once we have instantiated the model and training ops, we can save the model for future reuses:

```
saver = tf.train.Saver()
init_op = tf.global_variables_initializer()
session_conf = tf.ConfigProto(
  allow_soft_placement=FLAGS.allow_soft_placement, log_device_
placement=FLAGS.log_device_placement)
```

Now, we start training the model:

```
with tf.Session(config = session_conf) as sess:
    sess.run(init_op)
    print("%s\t%s\t%s\t%s" % ("Epoch", "Train err", "Validation err",
"Elapsed Time"))
    errors = deque(maxlen=samples_per_batch)
    train_error=[]
    val_error=[]
    start = time.time()
    for i in range(max_epochs * samples_per_batch):
        users, items, rates = next(iter_train)
        _, pred_batch = sess.run([train_op, infer], feed_dict={user_
batch: users, item_batch: items, rate_batch: rates})
```

```
            pred_batch = clip(pred_batch)
            errors.append(np.power(pred_batch - rates, 2))
            if i % samples_per_batch == 0:
                train_err = np.sqrt(np.mean(errors))
                test_err2 = np.array([])
                for users, items, rates in iter_test:
                    pred_batch = sess.run(infer, feed_dict={user_batch:
    users, item_batch: items})
                    pred_batch = clip(pred_batch)
                    test_err2 = np.append(test_err2, np.power(pred_batch -
    rates, 2))
                end = time.time()
                print("%02d\t%.3f\t\t%.3f\t\t%.3f secs" % (i // samples_
    per_batch, train_err, np.sqrt(np.mean(test_err2)), end - start))
                train_error.append(train_err)
                val_error.append(np.sqrt(np.mean(test_err2)))
                start = end
        saver.save(sess, checkpoint_prefix)
        pd.DataFrame({'train':train_error,'val':val_error}).to_
    pickle("val_error.pkl")
        print("Training Done !!!")
    sess.close()
```

So, the preceding code segment carries out the training and saves the errors in the pickle file. Finally, it prints the training and validation error and the time taken:

```
>>>
Epoch  Train err    Validation err     Elapsed Time

00     2.816        2.812        0.118 secs

01     2.813        2.812        4.898 secs

…      …            …            …
48     2.770        2.767        1.618 secs

49     2.765        2.760        1.678 secs

Training Done!!!
```

The preceding result is abridged and only a few steps have been shown. Now, let's look at these errors graphically:

```
    error = pd.read_pickle("val_error.pkl")
    error.plot(title="Train vs Validation error")
    plt.show()
    >>>
```

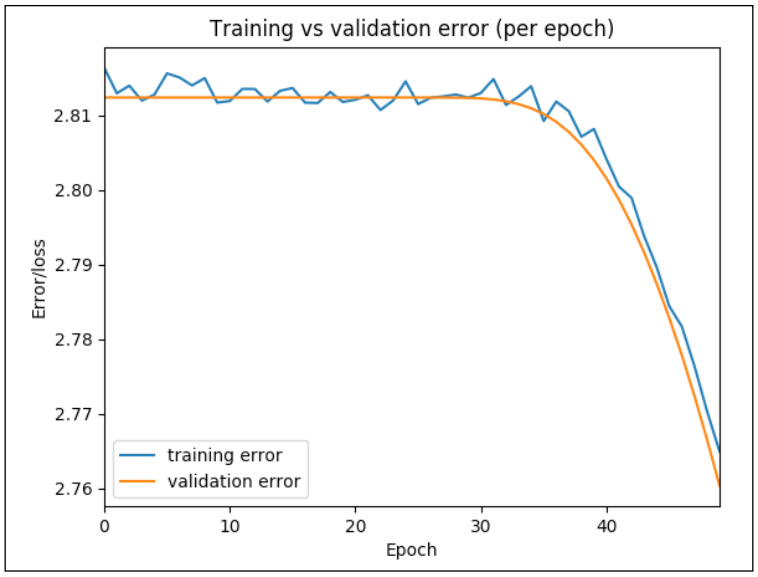


Figure 5: Training versus validation error per epoch

From the preceding figure, it can be observed that over time, both the training and validation errors decrease. Nevertheless, you can still try to increase the steps and see if these two values can be reduced toward better accuracy.

Inferencing the saved model

The following code inference uses the saved model and it prints the overall validation error:

```
if FLAGS.val:
    print("Validation ...")
    init_op = tf.global_variables_initializer()
    #checkpoint_file = tf.train.latest_checkpoint(FLAGS.checkpoint_
dir)
    session_conf = tf.ConfigProto(
      allow_soft_placement=FLAGS.allow_soft_placement,
      log_device_placement=FLAGS.log_device_placement)
    with tf.Session(config = session_conf) as sess:
        #sess.run(init_op)
        new_saver = tf.train.import_meta_graph("{}.meta".
format(checkpoint_prefix))
        new_saver.restore(sess, tf.train.latest_checkpoint(FLAGS.
checkpoint_dir))
        test_err2 = np.array([])
```

```
        for users, items, rates in iter_test:
            pred_batch = sess.run(infer, feed_dict={user_batch: users,
item_batch: items})
            pred_batch = clip(pred_batch)
            test_err2 = np.append(test_err2, np.power(pred_batch -
rates, 2))
            print("Validation Error: ",np.sqrt(np.mean(test_err2)))
    print("Done !!!")
sess.close()
>>>
Validation Error:  2.14626890224
Done!!!
```

Generating a user-item table

The following method creates a user-item dataframe. It is used to create a trained `DataFrame`; all missing values in the user-item table is filled here using the SVD trained model. It takes the rating dataframe and stores all the user ratings for the respective movies. Finally, it generates a filled rating dataframe, where user is the row and item is the column:

```
def create_df(ratings_df=readers.read_file(FLAGS.data_file,
sep="::")):
    if os.path.isfile("model/user_item_table.pkl"):
        df=pd.read_pickle("user_item_table.pkl")
    else:
        df = ratings_df.pivot(index = 'user', columns ='item', values
= 'rate').fillna(0)
        df.to_pickle("user_item_table.pkl")
    df=df.T
    users=[]
    items=[]
    start = time.time()
    print("Start creating user-item dense table")
    total_movies=list(ratings_df.item.unique())
    for index in df.columns.tolist():
        #rated_movies=ratings_df[ratings_df['user']==index].
drop(['st', 'user'], axis=1)
        rated_movie=[]
rated_movie=list(ratings_df[ratings_df['user']==index].drop(['st',
'user'], axis=1)['item'].values)
        unseen_movies=[]
        unseen_movies=list(set(total_movies) - set(rated_movie))
        for movie in unseen_movies:
            users.append(index)
```

```
            items.append(movie)
    end = time.time()
    print(("Found in %.2f seconds" % (end-start)))
    del df
    rated_list = []
    init_op = tf.global_variables_initializer()
    #checkpoint_file = tf.train.latest_checkpoint(FLAGS.checkpoint_
dir)
    session_conf = tf.ConfigProto(
      allow_soft_placement=FLAGS.allow_soft_placement,
      log_device_placement=FLAGS.log_device_placement)
    with tf.Session(config = session_conf) as sess:
        #sess.run(init_op)
        print("prediction started ...")
        new_saver =tf.train.import_meta_graph("{}.meta".
format(checkpoint_prefix))
        new_saver.restore(sess, tf.train.latest_checkpoint(FLAGS.
checkpoint_dir))
        test_err2 = np.array([])
        rated_list = sess.run(infer, feed_dict={user_batch: users,
item_batch: items})
        rated_list = clip(rated_list)
        print("Done !!!")
    sess.close()
    df_dict={'user':users,'item':items,'rate':rated_list}
    df = ratings_df.drop(['st'],axis=1).append(pd.DataFrame(df_dict)).
pivot(index = 'user', columns ='item', values = 'rate').fillna(0)
    df.to_pickle("user_item_table_train.pkl")
    return df
```

Now, let's invoke the preceding method to generate the user-item table as a pandas dataframe:

```
create_df(ratings_df = readers.read_file(FLAGS.data_file, sep="::"))
```

This line will create the user versus item table for the training set and save the dataframe as the `user_item_table_train.pkl` file on your specified directory.

Clustering similar movies

For this part, refer to the `kmean.py` script. This script takes the rating data file as input and returns movies along with their respective clusters. However, first, let's import the required packages and modules:

```
import tensorflow as tf
import numpy as np
import pandas as pd
```

```
import time
import readers
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.decomposition import PCA
```

Now, let's define the data parameters to be used, including the path of the rating data file, a number of cluster K, the maximum number of iterations, and define whether it should be used as a trained user versus item matrix:

```
tf.flags.DEFINE_string("data_file", "ratings.dat", "Data source for
the positive data.")
tf.flags.DEFINE_string("K", 5, "Number of clusters")
tf.flags.DEFINE_string("MAX_ITERS", 1000, "Maximum number of
iterations")
tf.flags.DEFINE_string("TRAINED", False, "Use TRAINED user vs item
matrix")
FLAGS = tf.flags.FLAGS
FLAGS._parse_flags()
```

Then, the `k_mean_clustering()` function is used, which returns the movies along with their respective clusters. It takes the rating dataset that is, `ratings_df`, which is a rating dataframe and stores all the user ratings for the respective movies, K as the number of clusters, `MAX_ITERS` as the maximum number of recommendations, and `TRAINED` is a Boolean type that signifies whether to use a trained user versus movie table or untrained. Finally, it returns the list of movies/items and list of clusters:

```
def k_mean_clustering(ratings_df,K,MAX_ITERS,TRAINED=False):
    if TRAINED:
        df=pd.read_pickle("user_item_table_train.pkl")
    else:
        df=pd.read_pickle("user_item_table.pkl")
    df = df.T
    start = time.time()
    N=df.shape[0]
    points = tf.Variable(df.as_matrix())
    cluster_assignments = tf.Variable(tf.zeros([N], dtype=tf.int64))
 centroids = tf.Variable(tf.slice(points.initialized_value(), [0,0],
[K,df.shape[1]]))
    rep_centroids = tf.reshape(tf.tile(centroids, [N, 1]), [N, K,
df.shape[1]])
    rep_points = tf.reshape(tf.tile(points, [1, K]), [N, K,
df.shape[1]])
    sum_squares = tf.reduce_sum(tf.square(rep_points - rep_
centroids),reduction_indices=2)
```

```
    best_centroids = tf.argmin(sum_squares, 1)    did_assignments_
change = tf.reduce_any(tf.not_equal(best_centroids, cluster_
assignments))
    means = bucket_mean(points, best_centroids, K)
    with tf.control_dependencies([did_assignments_change]):
        do_updates = tf.group(
            centroids.assign(means),
            cluster_assignments.assign(best_centroids))
    init = tf.global_variables_initializer()
    sess = tf.Session()
    sess.run(init)
    changed = True
    iters = 0
    while changed and iters < MAX_ITERS:
        iters += 1
        [changed, _] = sess.run([did_assignments_change, do_updates])
    [centers, assignments] = sess.run([centroids, cluster_
assignments])
    end = time.time()
    print (("Found in %.2f seconds" % (end-start)), iters,
"iterations")
    cluster_df=pd.DataFrame({'movies':df.index.
values,'clusters':assignments})
    cluster_df.to_csv("clusters.csv",index=True)
    return assignments,df.index.values
```

In the preceding code, we have a silly initialization, in a sense that we use the first K points as the starting centroids. In the real world, it can be further improved. Readers should refer to the k-mean implementation in other chapters to get some clues on how to improve this.

We replicate to N copies of each centroid and K copies of each point, then subtract and compute the sum of squared distances. We use `argmin` to select the lowest-distance point. Do not write to the assigned clusters variable until after computing whether the assignments have changed - hence the dependencies.

If you look at the preceding code carefully, there is another function called `bucket_mean()`. It takes the data points, the best centroids, and the number of tentative clusters, that is K and computes the mean to be used in the cluster computation:

```
def bucket_mean(data, bucket_ids, num_buckets):
    total = tf.unsorted_segment_sum(data, bucket_ids, num_buckets)
    count = tf.unsorted_segment_sum(tf.ones_like(data), bucket_ids,
num_buckets)
    return total / count
```

Once we have trained our k-means model, the next task is to visualize these clusters representing similar movies. For this, I have written a function called `showClusters()` that takes the user item table, the clustered data written in a CSV file, that is `clusters.csv`, the number of principal components (default is 2), and the SVD solver (possible values are randomized and full).

The thing is that in a 2D space, it would be really difficult to plot all the data points representing the movie clusters. This is why we have applied PCA to reduce the dimensionality without sacrificing the quality much:

```
def showClusters(user_item_table, clustered_data,number_of_PCA_
components, svd_solver):
  user_item=pd.read_pickle(user_item_table)
  cluster=pd.read_csv(clustered_data, index_col=False)
  user_item=user_item.T
  pcs = PCA(number_of_PCA_components, svd_solver)
  cluster['x']=pcs.fit_transform(user_item)[:,0]
  cluster['y']=pcs.fit_transform(user_item)[:,1]
  fig = plt.figure()
  ax = plt.subplot(111)
  ax.scatter(cluster[cluster['clusters']==0]['x'].values,cluster[clust
er['clusters']==0]['y'].values,color="r", label='cluster 0')
  ax.scatter(cluster[cluster['clusters']==1]['x'].values,cluster[clust
er['clusters']==1]['y'].values,color="g", label='cluster 1')
  ax.scatter(cluster[cluster['clusters']==2]['x'].values,cluster[clust
er['clusters']==2]['y'].values,color="b", label='cluster 2')
  ax.scatter(cluster[cluster['clusters']==3]['x'].values,cluster[clust
er['clusters']==3]['y'].values,color="k", label='cluster 3')
  ax.scatter(cluster[cluster['clusters']==4]['x'].values,cluster[clust
er['clusters']==4]['y'].values,color="c", label='cluster 4')
  ax.legend()
  plt.title("Clusters of similar movies using K-means")
  plt.ylabel('PC2')
  plt.xlabel('PC1');
  plt.show()
```

Well done! We will evaluate our model and plot the clusters in the evaluation step.

Movie rating prediction by users

For movie rating prediction by users, I have written a function called `prediction()`. It takes the sample input about users and items that is, movies, then creates TensorFlow placeholders from the graph by name. It then evaluates these tensors. For the following code, it is to be noted that TensorFlow assumes that the checkpoint directory already exists, so make sure that it already exists (refer to the `main.py` file):

```
def prediction(users=FLAGS.predicted_user, items=FLAGS.predicted_item,
allow_soft_placement=FLAGS.allow_soft_placement,\
log_device_placement=FLAGS.log_device_placement, checkpoint_dir=FLAGS.
checkpoint):
    rating_prediction=[]
    checkpoint_prefix = os.path.join(FLAGS.checkpoint, "model")
    graph = tf.Graph()
    with graph.as_default():
        session_conf = tf.ConfigProto(allow_soft_placement=allow_soft_
placement,log_device_placement=log_device_placement)
        with tf.Session(config = session_conf) as sess:
            new_saver = tf.train.import_meta_graph("{}.meta".
format(checkpoint_prefix))
            new_saver.restore(sess, tf.train.latest_
checkpoint(checkpoint_dir))
            user_batch = graph.get_operation_by_name("id_user").
outputs[0]
            item_batch = graph.get_operation_by_name("id_item").
outputs[0]
            predictions = graph.get_operation_by_name("svd_
inference").outputs[0]
            pred = sess.run(predictions, feed_dict={user_batch: users,
item_batch: items})
            pred = clip(pred)
        sess.close()
    return  pred
```

We will look at how we can use this method to predict the top `K` movies and user rating for movies. In the preceding code segment, `clip()` is a user defined function that limits the values in an array. The idea for a given interval is that values outside the interval are clipped to the interval edges. For example, if an interval of [`0`, `1`] is specified, values smaller than `0` become `0`, and values larger than `1` become `1`. For our purpose, the interval is [`1.0`, `5.0`]. Here is the implementation:

```
def clip(x):
    return np.clip(x, 1.0, 5.0) # rating 1 to 5
```

Now, let's look at how we can use the `prediction()` method for making a prediction of a rating set of movies by a user:

```
def user_rating(users,movies):
    if type(users) is not list:       users=np.array([users])
    if type(movies) is not list:
        movies=np.array([movies])
    return prediction(users,movies)
```

So, the preceding function returns user ratings for the respective user. It takes a list of numbers or number, list of user IDs or just a user id, a list of numbers or number, and a list of movie IDs or just movie ID. Finally, it returns a list of predicted movies.

Finding the top K movies

The following method extracts unseen top `K` movies, where `K` is an arbitrary integer, say `10`. The name of the function is `top_k_movies()` that returns top `K` movies for the respective user. First, it takes a list of users or number, list of user IDs, and the rating dataframe. It then stores all the user ratings for the respective movies. The output is a dictionary containing the user ID as a key and list of top `K` movies for that user as a value:

```
def top_k_movies(users,ratings_df,k):
    dicts={}
    if type(users) is not list:
        users = [users]
    for user in users:
        rated_movies = ratings_df[ratings_df['user']==user].
drop(['st', 'user'], axis=1)
        rated_movie = list(rated_movies['item'].values)
        total_movies = list(ratings_df.item.unique())
        unseen_movies = list(set(total_movies) - set(rated_movie))
        rated_list = []
        rated_list = prediction(np.full(len(unseen_movies),user),np.
array(unseen_movies))
        useen_movies_df = pd.DataFrame({'item': unseen_
movies,'rate':rated_list})
        top_k = list(useen_movies_df.sort_values(['rate','item'],
ascending=[0, 0])['item'].head(k).values)
        dicts.update({user:top_k})
    result = pd.DataFrame(dicts)
    result.to_csv("user_top_k.csv")
    return dicts
```

In the preceding code segment, `prediction()` is a user defined function that we described previously. We will look at an example of how to predict top `K` movies (refer to `Test.py` for more, in a later section).

Predicting top K similar movies

I have written a function called `top_k_similar_items()` that computes and returns `K` similar movies for the respective movie. It takes a list of numbers or number, list of movie IDs, and the rating dataframe and stores all user ratings for the respective movies. It also takes `K` as a natural number.

The value of `TRAINED` can be either `True` or `False`, whether we use trained user versus movie table or untrained. Finally, it returns a list of `K` similar movies for the respective movie:

```
def top_k_similar_items(movies,ratings_df,k,TRAINED=False):
    if TRAINED:
        df=pd.read_pickle("user_item_table_train.pkl")
    else:
        df=pd.read_pickle("user_item_table.pkl")
    corr_matrix=item_item_correlation(df,TRAINED)
    if type(movies) is not list:
        return corr_matrix[movies].sort_values(ascending=False).
drop(movies).index.values[0:k]
    else:
        dict={}
        for movie in movies:           dict.update({movie:corr_
matrix[movie].sort_values(ascending=False).drop(movie).index.
values[0:k]})
        pd.DataFrame(dict).to_csv("movie_top_k.csv")
        return dict
```

In the preceding code segment, the `item_item_correlation()` function is a user-defined function that computes the movie-movie correlation that is used in predicting the top `K` similar movies. The method can be seen as follows:

```
def item_item_correlation(df,TRAINED):
    if TRAINED:
        if os.path.isfile("model/item_item_corr_train.pkl"):
            df_corr=pd.read_pickle("item_item_corr_train.pkl")
        else:
            df_corr=df.corr()
            df_corr.to_pickle("item_item_corr_train.pkl")
    else:
        if os.path.isfile("model/item_item_corr.pkl"):
```

```
            df_corr=pd.read_pickle("item_item_corr.pkl")
        else:
            df_corr=df.corr()
            df_corr.to_pickle("item_item_corr.pkl")
    return df_corr
```

Computing the user-user similarity

For computing the user-user similarity, I have written the `user_similarity()` function that returns the similarity between two users. It takes three parameters: i) user 1, user 2 ii) Rating dataframe, iii) The value of `TRAINED` that can be either `TRUE` or `FALSE`, whether we use a trained user versus movie table or untrained. Finally, it computes the Pearson coefficient between users [value in between `-1` to `1`]:

```
def user_similarity(user_1,user_2,ratings_df,TRAINED=False):
corr_matrix=user_user_pearson_corr(ratings_df,TRAINED)
    return corr_matrix[user_1][user_2]
```

In the preceding function, `user_user_pearson_corr()` is a function that computes the user-user Pearson correlation:

```
def user_user_pearson_corr(ratings_df,TRAINED):
    if TRAINED:
        if os.path.isfile("model/user_user_corr_train.pkl"):
            df_corr=pd.read_pickle("user_user_corr_train.pkl")
        else
            df =pd.read_pickle("user_item_table_train.pkl")
            df=df.T
            df_corr=df.corr()
            df_corr.to_pickle("user_user_corr_train.pkl")
    else:
        if os.path.isfile("model/user_user_corr.pkl"):
            df_corr=pd.read_pickle("user_user_corr.pkl")
        else:
            df = pd.read_pickle("user_item_table.pkl")
            df=df.T
            df_corr=df.corr()
            df_corr.to_pickle("user_user_corr.pkl")
    return df_corr
```

Evaluating the recommendation system

Now, we should recommend the model. In this sub-section, we will evaluate the cluster and plot them to see the high-level movies. We will also see how the movies are spread across different clusters.

We will then see top `K` movies and see the user-user similarity and so on. Now, let's get started. First, let's import the required libraries:

```
import tensorflow as tf
import pandas as pd
import readers
import main
import kmean as km
import numpy as np
```

Then, let's define the data parameters to be used for evaluation:

```
DATA_FILE = "ratings.dat" # Data source for the positive data.
K = 5 #Number of clusters
MAX_ITERS = 1000 #Maximum number of iterations
TRAINED = False # Use TRAINED user vs item matrix
USER_ITEM_TABLE = "user_item_table.pkl" #File location for the user
item table
COMPUTED_CLUSTER_CSV = "clusters.csv" #name of the computed file -i.e.
csv
NO_OF_PCA_COMPONENTS = 2 #number of pca components
SVD_SOLVER = "randomized" #svd solver -e.g. randomized, full etc.
```

Let's load the rating dataset that will be used in the invoke call to the `k_mean_clustering()` method:

```
ratings_df = readers.read_file("ratings.dat", sep="::")
clusters,movies = km.k_mean_clustering(ratings_df, K, MAX_ITERS,
TRAINED = False)
cluster_df=pd.DataFrame({'movies':movies,'clusters':clusters})
```

Well done! Now, let's look at clusters of simple input: movies along with their respective clusters:

```
print(cluster_df.head(10))
>>>
clusters  movies
0         0       0
1         4       1
2         4       2
3         3       3
4         4       4
5         2       5
6         4       6
7         3       7
8         3       8
9         2       9
```

```
print(cluster_df[cluster_df['movies']==1721])
>>>
clusters  movies
1575         2    1721
print(cluster_df[cluster_df['movies']==647])
>>>
    clusters  movies
627         2     647
```

Let's look at how the movies are scattered across clusters that is, clusters of similar movies:

```
km.showClusters(USER_ITEM_TABLE, COMPUTED_CLUSTER_CSV, NO_OF_PCA_
COMPONENTS, SVD_SOLVER)
>>>
```

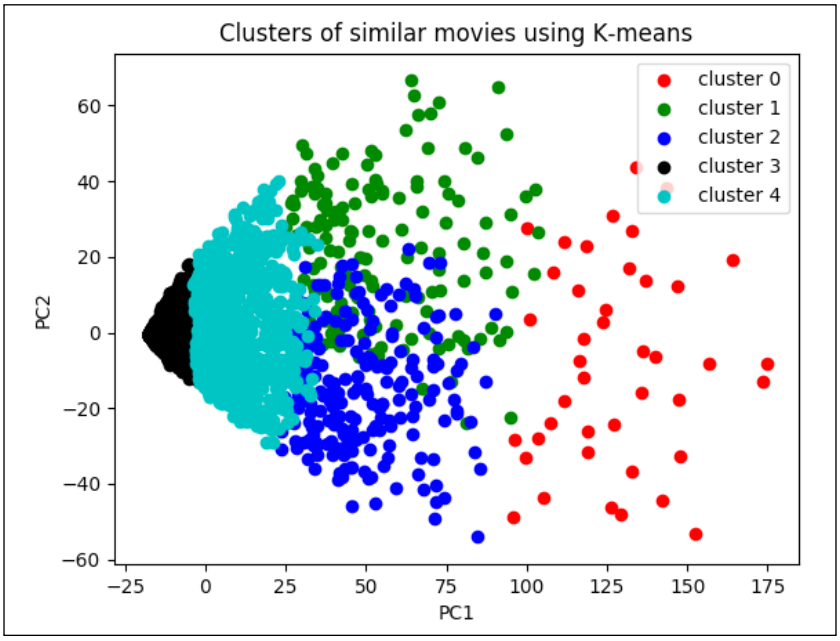


Figure 6: Clusters of similar movies

From the preceding figure, it is clear that the data points are more accurately clustered across cluster 3 and 4. However, cluster 0, 1, and 2 are more scattered and did not cluster well. Now, let's compute the top k similar movies for the respective movie and print them:

```
ratings_df = readers.read_file("ratings.dat", sep="::")
topK = main.top_k_similar_items(9,ratings_df = ratings_df,k =
10,TRAINED = False)
```

```
print(topK)
>>>
[1721, 1369,  164, 3081,  732,  348,  647, 2005,  379, 3255]
```

Now, let's compute the Pearson correlation between user-user. When you run this user similarity function, on first run, it will take time to give output but after that its response is in real time:

```
print(main.user_similarity(1,345,ratings_df))
>>>
0.15045477803357316
```

Now, let's compute the aspect rating for a user for a movie:

```
print(main.user_rating(0,1192))
>>>
4.25545645

print(main.user_rating(0,660))
>>>
3.20203304
```

Finally, let's look at the top `K` movie recommendations for the user:

```
print(main.top_k_movies([768],ratings_df,10))
>>>
{768: [2857, 2570, 607, 109, 1209, 2027, 592, 588, 2761, 479]}
print(main.top_k_movies(1198,ratings_df,10))
>>>

{1198: [2857, 1195, 259, 607, 109, 2027, 592, 857, 295, 479]}
```

So far, we have seen how we can develop a simple recommendation engine using the movie and rating dataset. However, often this type of engine cannot make a robust and accurate prediction.

Therefore, researchers are trying to use FM for developing more accurate and robust recommendation engines. In the next section, we will look at some examples using the FM algorithm and its variations.

Factorization machines for recommendation systems

In this section, we will look at two examples for developing a more robust recommendation systems using FM. We will start with a brief explanation of FM and their applications in the cold-start recommendation problem.

Then, we will see a short example using FM for developing a real-life recommendation system. Then, we will see another example using the improved version of the FM algorithm called **neural factorization machines** (**NFM**).

Factorization machines

FM-based techniques are at the cutting edge for personalization. They have proven to be an extremely powerful tool with enough expressive capacity to generalize existing models such as Matrix/Tensor Factorization and Polynomial Kernel regression. In other words, this type of algorithm is a supervised learning approach for enhancing the performance of linear models by incorporating second-order feature interactions that are absent in the matrix factorization, such as algorithms.

Existing recommendation algorithms demand to have a consumption (for example, product) or rating (for example, movie) dataset in a (user, item, rating) tuple. These types of dataset are mostly used by the variations of **collaborative filtering** (**CF**) algorithms. CF algorithms have gained wide adoption and have proven to yield nice results. However, in many instances, we have plenty of item metadata (tags, categories, genres, and so on) that can be used to make better predictions too. Unfortunately, these types of metadata are not used by CF algorithms.

Factorization machines can make proper utilization of these feature-rich (meta) datasets. FM can consume these extra features to model higher-order interactions specifying the dimensionality parameter d. Most importantly, FM is also optimized enough for handing large-scale sparse datasets. Therefore, a second order FM model would be sufficient as there is not enough information to estimate more complex interactions:

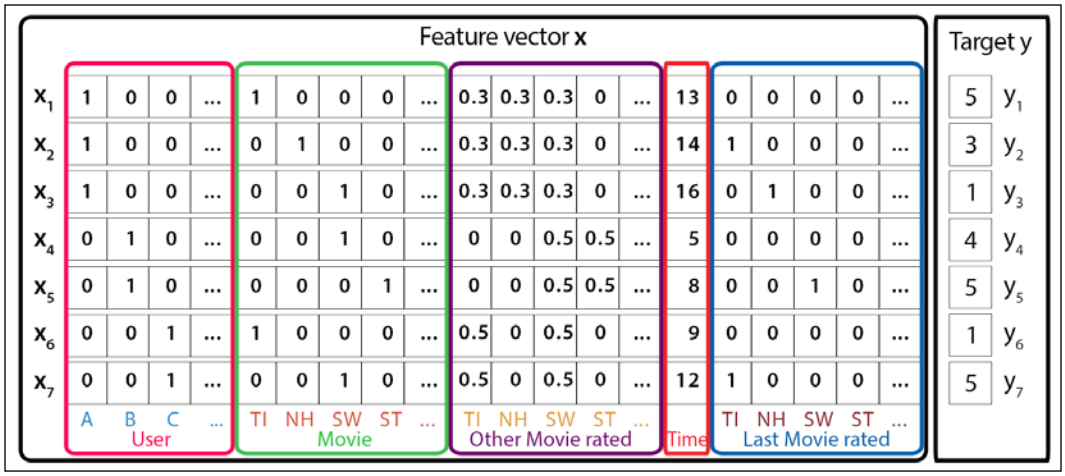


Figure 7: An example training dataset for representing personalization problem feature vectors x and target y

Let's assume that the dataset of a prediction problem is described by a design matrix $X \in \mathbb{R}^{nxp}$ as shown in figure 7. In figure 1, i^{th} row $\mathbf{x}_i \in \mathbb{R}^p$ of X describes one case with p real-valued variables and where y_i is the prediction target of the ith case. Alternatively, one can describe this set as a set S of tuples (x,y), where (again) $\mathbf{x} \in \mathbb{R}^p$ is a feature vector and y is its corresponding target that is a label.

In other words, in figure 7, every row represents a feature vector x_i with its corresponding target y_i. For easier interpretation, the features are grouped into indicators for the active user (blue), active item (red), other movies rated by the same user (orange), the time in months (green), and the last movie rated (brown). Then, the FM algorithm models all the nested interactions up to order d between p input variables in x using factorized interaction parameters:

$$\hat{y}(x) = w_0 + \sum_{i=i}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} \langle v_i, v_j \rangle x_i x_j$$

In the preceding equation, v's represent the k-dimensional latent vectors associated with each variable (that is, users and items) and the bracket operator represents the inner product. This kind of representation with data matrices and feature vectors is common in many machine learning approaches, for example, in linear regression or support vector machines (SVM).

However, if you are familiar with the **matrix factorization** (**MF**) models, the preceding equation should look familiar: it contains a global bias as well as user/item specific bias and includes user-item interactions. Now, if we assume that each $x(j)$ vector is only non-zero at positions u and i, we get the classic MF model:

$$\hat{y}(x) = w_0 + w_i + w_u + \langle v_i, v_j \rangle$$

Nevertheless, MF models for recommendation systems often suffer from the cold start problem. We will look at this in the next section.

The cold start problem in recommendation systems

The term cold start problem sounds funny, but as the name implies, it is derived from cars. Suppose you're living in Alaska, so due to the intense cold, your car's engine might not start smoothly. However, once it reaches its optimal operating temperature, it will start, run, and operate normally.

In the realm of recommendation engines, the term "cold start" simply means a circumstance that is not yet optimal for the engine to provide the best possible results. In e-commerce, there are two distinct categories for a cold start: product cold start and user cold start.

Recommendation engines using a CF-based approach and recommends each item based on user actions. The more user actions an item has, the easier it is to tell which user would be interested in it and what other items are similar to it. As time progresses, the system will be able to give more and more accurate recommendations. At a certain stage, when new items or users are added to the user-item matrix (refer to figure 2), this problem occurs.

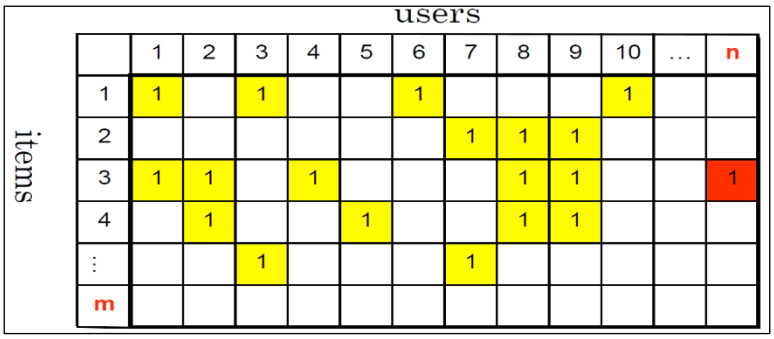


Figure 10: User's versus items matrix sometimes leads to a cold start problem

In this case, the recommendation engine does not have enough knowledge about this new user or this new item yet. The content-based filtering approach, similar to FM, is the method that can be incorporated to alleviate the cold start problem.

So, the main difference between the previous two equations is that FM introduces higher order interactions in terms of latent vectors that are also affected by categorical or tag data. This means that the models go beyond co-occurrences in order to find stronger relationships between latent representations of each feature.

Problem definition and formulation

Given a sequence of click events performed by some users during a typical session on an e-commerce website, the goal is to predict whether the user is going to buy something or not, and if he is buying, what will be the items that he might buy. The task could, therefore, be divided into two sub-goals:

- Is the user going to buy items in this session? Yes or No
- If yes, what are the items that are going to be bought?

Now, to predict the quantity of an item bought in a session, a robust classifier can help predict whether the user will buy or not in this session. The following is the original implementation of FM; the training data should be structured as follows:

	User				Item				Categories				History				Quantity	
x_1	1	0	1	...	0	1	0	...	1	2	3	...	1	1	0	...	3	y_1
x_2	0	0	1	...	1	0	1	...	8	9	6	...	0	1	0	...	7	y_2
x_3	0	1	1	...	1	0	0	...	5	2	7		1	1	1	...	9	y_3
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
x_n	1	0	1	...	1	1	1	...	2	4	6	...	0	1	1	...	8	y_n

Figure 11: User versus item/category/history table can be used to train the recommendation model

Now, to prepare our training set like this, we can use the `get_dummies()` method from pandas to transform all the columns with categorical data since FM models work with categorical data represented as integers.

Therefore, I will use `tffm`, which is a TensorFlow-based implementation of FM (refer to `https://github.com/geffy/tffm` for more) and pandas for pre-processing and structuring the data. There are two functions called `TFFMClassifier` and `TFFMRegressor` that can be used for making a prediction and calculating MSE respectively. Before this, let's look at some insights of the dataset to be used for this example.

Dataset description

For this example, I will use the RecSys 2015 challenge dataset to illustrate how to fit an FM model to get a personalized recommendation. The data contains click and purchase events for an e-commerce site, with additional item category data; the dataset size is about 275 MB that can be downloaded from `https://s3-eu-west-1.amazonaws.com/yc-rdata/yoochoose-data.7z`. There are three files and a readme file; however, we will use `youchoose-buys.dat` (buy events) and `youchoose-clicks.dat` (that is, click events):

1. `yoochoose-clicks.dat`: Each record/line in the file has the following fields:
    - **Session ID**: Session ID that is, there are one or many clicks in one session
    - **Timestamp**: The time when the click occurred
    - **Item ID**: The unique identifier of the item
    - **Category**: The category of the item

2. `yoochoose-buys.dat`: Each record/line in the file has the following fields:
    - **Session ID**: Session ID, one or many buying events in a session
    - **Timestamp**: The time when the buy occurred
    - **Item ID**: Unique identifier of items
    - **Price**: The price of the item
    - **Quantity**: How many of this item were bought?

The Session IDs in `yoochoose-buys.dat` also existed in the `yoochoose-clicks.dat file`. This means that the records with the same Session ID together form the sequence of click events of a certain user during the session. The session could be short (for example, a few minutes) or very long (for example, a few hours); it can have one click or hundreds of clicks. It all depends on the activity of the user.

Preprocessing

If we want to make the full use of the categories and expanded historical data, we need to load and convert the data into the right format. Thus, some preprocessing before getting the training set prepared is necessary. Let's start by loading the packages and modules:

```
import pandas as pd
from collections import Counter
import tensorflow as tf
from tffm import TFFMRegressor
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
import numpy as np
```

I am assuming that you have already downloaded the dataset from the previously mentioned link. Now, let's load the dataset:

```
buys = open('yoochoose-buys.dat', 'r')
clicks = open('yoochoose-clicks.dat', 'r')
```

Now, create the pandas dataframe for the click and buys dataset:

```
initial_buys_df = pd.read_csv(buys, names=['Session ID', 'Timestamp',
'Item ID', 'Category', 'Quantity'], dtype={'Session ID': 'float32',
'Timestamp': 'str', 'Item ID': 'float32','Category': 'str'})
initial_buys_df.set_index('Session ID', inplace=True)
initial_clicks_df = pd.read_csv(clicks, names=['Session ID',
'Timestamp', 'Item ID', 'Category'],dtype={'Category': 'str'})
initial_clicks_df.set_index('Session ID', inplace=True)
```

We don't need to use the timestamps in this example, so let's drop it from the dataframe:

```
initial_buys_df = initial_buys_df.drop('Timestamp', 1)
initial_clicks_df = initial_clicks_df.drop('Timestamp', 1)
```

Now, for illustrative purposes, let's use only a subset of the data. More specifically, let's take the top `10000` buying users:

```
x = Counter(initial_buys_df.index).most_common(10000)
top_k = dict(x).keys()
initial_buys_df = initial_buys_df[initial_buys_df.index.isin(top_k)]
initial_clicks_df = initial_clicks_df[initial_clicks_df.index.
isin(top_k)]
```

Now, let's create a copy of the index, since we will also apply one hot encoding on the index:

```
initial_buys_df['_Session ID'] = initial_buys_df.index
```

As we mentioned earlier, we can introduce historical engagement data into our FM model. We will use some `group_by` magic to generate a history profile of all of a user's engagement. First, we use one hot encoding on all columns for clicks and buys:

```
transformed_buys = pd.get_dummies(initial_buys_df)
transformed_clicks = pd.get_dummies(initial_clicks_df)
```

Now, it's time to aggregate historical data for items and categories:

```
filtered_buys = transformed_buys.filter(regex="Item.*|Category.*")
filtered_clicks = transformed_clicks.filter(regex="Item.*|Category.*")
historical_buy_data = filtered_buys.groupby(filtered_buys.index).sum()
historical_buy_data = historical_buy_data.rename(columns=lambda
column_name: 'buy history:' + column_name)
historical_click_data = filtered_clicks.groupby(filtered_clicks.
index).sum()
historical_click_data = historical_click_data.rename(columns=lambda
column_name: 'click history:' + column_name)
```

Finally, we merge historical data of every `user_id`:

```
merged1 = pd.merge(transformed_buys, historical_buy_data, left_
index=True, right_index=True)
merged2 = pd.merge(merged1, historical_click_data, left_index=True,
right_index=True)
```

Implementing an FM model

Since we have prepared the dataset, the next task is to create the MF model. Regarding the hyperparameter, you can tune these parameters:

```
model = TFFMRegressor(
    order=2,
    rank=7,
    optimizer=tf.train.AdamOptimizer(learning_rate=0.1),
    n_epochs=100,
    batch_size=-1,
    init_std=0.001,
    input_type='dense'
```

Now, we prepare the matrix as presented in figure 10:

```
merged2.drop(['Item ID', '_Session ID', 'click history:Item ID', 'buy
history:Item ID'], 1, inplace=True)
X = np.array(merged2)
X = np.nan_to_num(X)
y = np.array(merged2['Quantity'].as_matrix())
```

Now, let's split data into train test:

```
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.2)
```

Split testing data in half: full information versus cold start:

```
X_te, X_te_cs, y_te, y_te_cs = train_test_split(X_te, y_te, test_
size=0.5)
X_te_cs = pd.DataFrame(X_te_cs, columns=merged2.columns)
```

Now, let's see what happens if we only have access to categories and no historical click/purchase data. For doing this, let's delete the historical click and purchasing data for the `cold_start` test set:

```
for column in X_te_cs.columns:
    if ('buy' in column or 'click' in column) and ('Category' not in
column):
        X_te_cs[column] = 0
```

Now, it's time to compute the mean squared error for both test sets:

```
model.fit(X_tr, y_tr, show_progress=True)
predictions = model.predict(X_te)
cold_start_predictions = model.predict(X_te_cs)
print('MSE: {}'.format(mean_squared_error(y_te, predictions)))
```

```
print('Cold-start MSE: {}'.format(mean_squared_error(y_te_cs,
predictions)))
>>>
MSE: 0.4123308369523053
Cold-start MSE: 2.7871912983642093
```

Finally, we destroy the model to free the memory:

```
model.destroy()
```

Thus, dropping category columns in the training dataset makes the MSE even smaller but doing so means that we cannot tackle the cold start recommendation problem. The experimental results are pretty good under the given condition that we have used a relatively small dataset that is, top `K` items to fit the TF model. As expected, it is easier to generate predictions if we have access to the full information setting with item purchases and clicks, but we still get a decent predictor for cold start recommendations using only aggregated category data:

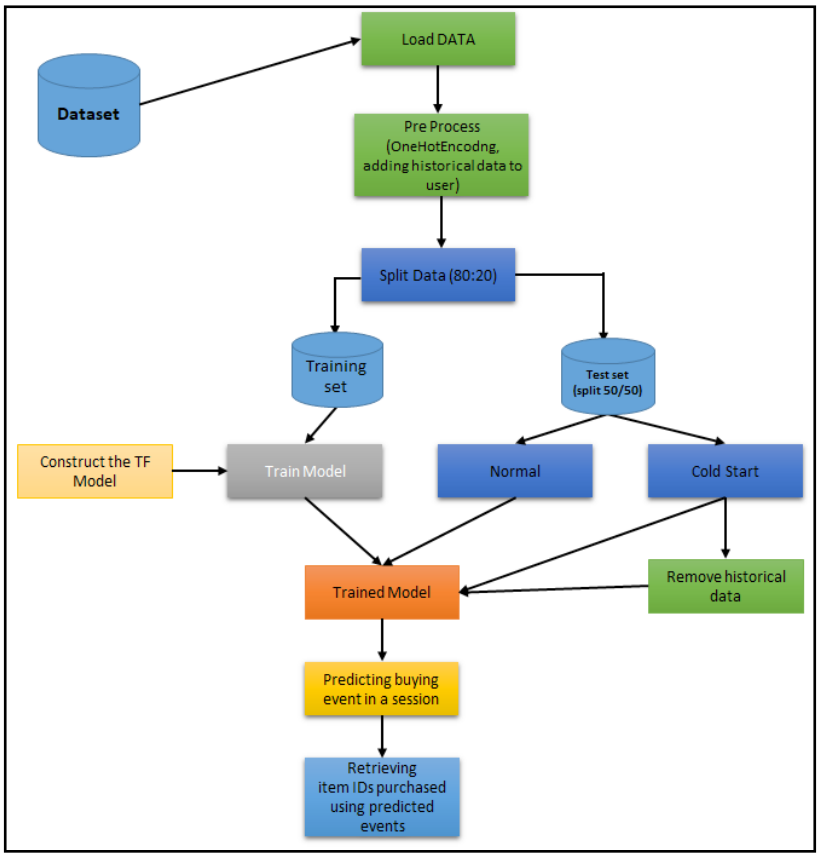


Figure 12: Workflow for predicting the list of bought items in a session using Factorization Machines.

I leave making the prediction and generation of the `solution.data` file to the readers as an exercise. Figure 12 shows a workflow that should help us to reach the solution. Nevertheless, here I have provided some clues to help you get the complete solution:

- Convert the whole scenario in the form of a classification task.
- Use the `tffm` classifier for predicting buying probability for each item for a single session.
- Also, predict buying probability for each item for a single session. A general warning—the overall job will be highly memory intensive. Therefore, try to get a powerful machine for this, for example, Corei7 CPU and 64GB of RAM recommended.
- `tffm` Classifier supports only binary classifications with 0/1 labels. So, do you plan to transform the prediction as a binary classification?

Improved factorization machines for predictive analytics

Many predictive tasks of web applications need to model categorical variables, such as user IDs and demographic information, such as genders and occupations. To apply standard machine learning techniques, these categorical predictors need to be converted to a set of binary features via one hot encoding (or any other techniques). This makes the resultant feature vector highly sparse. Thus, to learn from such sparse data commendably, it is important to consider the interactions between features.

In the previous section, we have seen that FM can be applied to model the second-order feature interactions effectively. However, FM models the feature interactions in a linear way, which can be insufficient for capturing the non-linear and complex inherent structure of real-world data. Therefore, several research initiatives have been proposed by Xiangnan He and Jun Xiao and others respectively, to overcome this limitation such as **neural factorization machine** (**NFM**) and **attentional factorization machine** (**AFM**).



For more, refer to the following reference papers:

- Xiangnan He and Tat-Seng Chua, Neural Factorization Machines for Sparse Predictive Analytics. In Proceedings of SIGIR '17, Shinjuku, Tokyo, Japan, August 07-11, 2017.
- Jun Xiao, Hao Ye, Xiangnan He, Hanwang Zhang, Fei Wu and Tat-Seng Chua (2017). Attentional Factorization Machines: Learning the Weight of Feature Interactions via Attention Networks IJCAI, Melbourne, Australia, August 19-25, 2017.

NFM can be used for making predictions under sparse settings by seamlessly combining the linearity of FM in modeling second-order feature interactions and the non-linearity of the neural network in modeling higher-order feature interactions.

On the other hand, AFM can be used to model the data even for all feature interactions with the same weight, as not all feature interactions are equally useful and predictive. In the next section, we will look at an example of using NFM for movie recommendations.

Neural factorization machines

Using the original FM algorithm, performance can be hindered by its modeling of all feature interactions with the same weight, as not all feature interactions are equally useful and predictive. For example, the interactions with useless features may even introduce noises and adversely degrade the performance.

Recently, Xiangnan H. and others proposed an improved version of the FM algorithm called NFM. NFM seamlessly combines the linearity of FM in modeling second-order feature interactions and the non-linearity of the neural network in modeling higher-order feature interactions. Conceptually, NFM is more expressive than FM since FM can be seen as a special case of NFM without hidden layers.

Dataset description

I used the MovieLens data for personalized tag recommendation that contains 668,953 tag applications of users on movies, where each tag application (user ID, movie ID, and tag) is converted into a feature vector using one hot encoding. This leaves 90,445 binary features called `ml-tag` dataset. I have used a Pearl script for the `.dat` to `.libfm` format. The conversion procedure is described at `http://www.libfm.org/libfm-1.42.manual.pdf` (section 2.2.1). The converted dataset has three files for the training, validation, and testing set, as follows:

- `ml-tag.train.libfm`
- `ml-tag.validation.libfm`
- `ml-tag.test.libfm`

For more information about this file format, refer to `http://www.libfm.org/`.

Using NFM for movie recommendations

I have reused and extended the NFM implementation using TensorFlow from this GitHub link at `https://github.com/hexiangnan/neural_factorization_machine`. This is a deep version of factorization machine and is more expressive than FM. The repository has three files namely `NeuralFM.py`, `FM.py`, and `LoadData.py`:

- `FM.py`: It is used to train the dataset. This is the original implementation of the Factorization Machines.
- `NeuralFM.py`: It is used to train the dataset. This is the original implementation of the Neural Factorization Machines but with some improvement and extension.
- `LoadData.py`: It is used to preprocess and load the dataset in the `libfm` format.

Model training

First, we will train the FM model with the following command. The command also includes the parameters needed to perform the training:

```
$ python3 FM.py --dataset ml-tag --epoch 20 --pretrain -1 --batch_size
4096 --lr 0.01 --keep 0.7
>>>
FM: dataset=ml-tag, factors=16, #epoch=20, batch=4096, lr=0.0100,
lambda=0.0e+00, keep=0.70, optimizer=AdagradOptimizer, batch_norm=1
#params: 1537566
Init:      train=1.0000, validation=1.0000 [5.7 s]
Epoch 1 [13.9 s] train=0.5413, validation=0.6005 [7.8 s]
```

```
Epoch 2 [14.2 s] train=0.4927, validation=0.5779 [8.3 s]
…
Epoch 19 [15.4 s]       train=0.3272, validation=0.5429 [8.1 s]
Epoch 20 [16.6 s]       train=0.3242, validation=0.5425 [7.8 s]
```

Once the training is finished, the trained model will be saved in the `pretrain` folder in your home directory:

```
>>>
Save model to file as pretrain
```

Additionally, I have tried to make the training and validation error visible for both the validation as well as the training loss, using the following code:

```
    # Plot loss over time
    plt.plot(epoch_list, train_err_list, 'r--', label='FM training
loss per epoch', linewidth=4)
    plt.title('FM training loss per epoch')
    plt.xlabel('Epoch')
    plt.ylabel('Training loss')
    plt.legend(loc='upper right')
    plt.show()

    # Plot accuracy over time
    plt.plot(epoch_list, valid_err_list, 'r--', label='FM validation
loss per epoch', linewidth=4)
    plt.title('FM validation loss per epoch')
    plt.xlabel('Epoch')
    plt.ylabel('Validation loss')
    plt.legend(loc='upper left')
    plt.show()
```

The preceding lines produce training versus validation loss per iteration in the FM model:

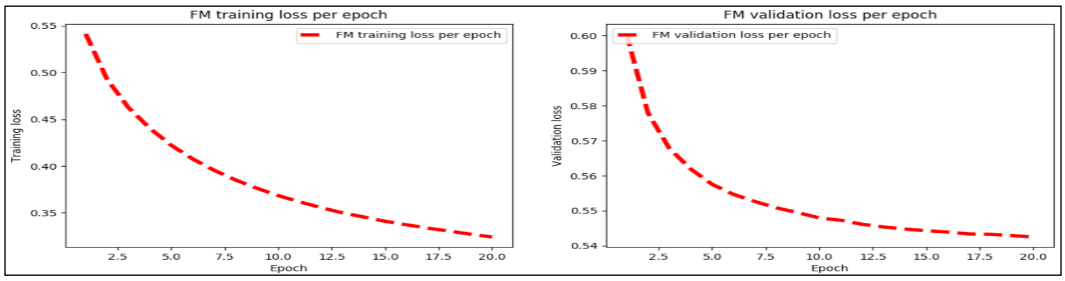


Figure 13: Training versus validation loss per iteration in the FM model

Now, if you see the preceding output logs, the best training (for both validation and training) occurs in iteration 20 that is, the last iteration. However, you can still iterate the training for higher iteration to get better that is, low RMSE value in the evaluation step:

```
Best Iter(validation)= 20      train = 0.3242, valid = 0.5425 [490.9
s]
```

Now, let's train the NFM model using the following command (but play with the parameters too):

```
$ python3 NeuralFM.py --dataset ml-tag --hidden_factor 64 --layers
[64] --keep_prob [0.8,0.5] --loss_type square_loss --activation relu
--pretrain 0 --optimizer AdagradOptimizer --lr 0.01 --batch_norm 1
--verbose 1 --early_stop 1 --epoch 20
>>>
Neural FM: dataset=ml-tag, hidden_factor=64, dropout_keep=[0.8,0.5],
layers=[64], loss_type=square_loss, pretrain=0, #epoch=20, batch=128,
lr=0.0100, lambda=0.0000, optimizer=AdagradOptimizer, batch_norm=1,
activation=relu, early_stop=1
#params: 5883150
Init:      train=0.9911, validation=0.9916, test=0.9920 [25.8 s]
Epoch 1 [60.0 s] train=0.6297, validation=0.6739, test=0.6721 [28.7 s]
Epoch 2 [60.4 s] train=0.5646, validation=0.6390, test=0.6373 [28.5 s]
…
Epoch 19 [53.4 s]      train=0.3504, validation=0.5607, test=0.5587
[25.7 s]
Epoch 20 [55.1 s]      train=0.3432, validation=0.5577, test=0.5556
[27.5 s]
```

Additionally, I have tried to make the training and validation error visible for both the validation as well as the training loss, using the following code:

```
    # Plot test accuracy over time
    plt.plot(epoch_list, test_err_list, 'r--', label='NFM test loss
per epoch', linewidth=4)
    plt.title('NFM test loss per epoch')
    plt.xlabel('Epoch')
    plt.ylabel('Test loss')
    plt.legend(loc='upper left')
    plt.show()
```

The preceding lines produces training versus validation loss per iteration in the NFM model:

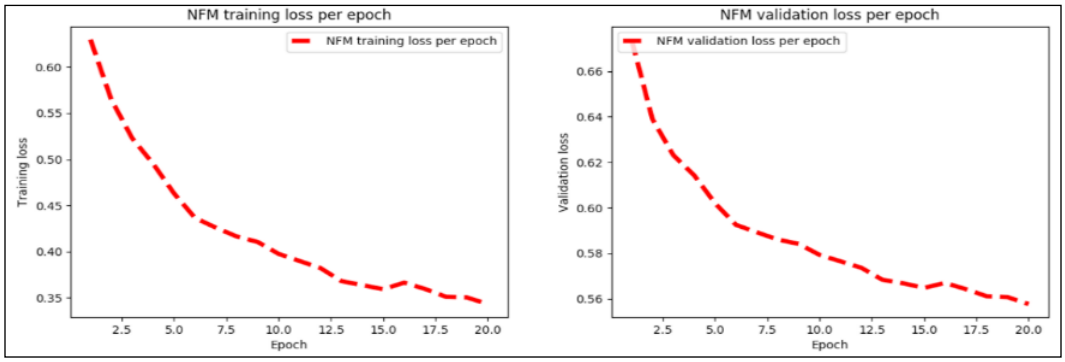


Figure 14: Training versus validation loss per iteration in the NFM model

Again for the NFM model, the best training (for both validation and training) occurs in iteration `20` that is, the last iteration. However, you can still iterate the training for higher iteration to get better that is, low RMSE value in the evaluation step:

```
Best Iter (validation) = 20   train = 0.3432, valid = 0.5577, test =
0.5556 [1702.5 s]
```

Model evaluation

Now, to evaluate the original FM model, execute the following command:

```
$ python3 FM.py --dataset ml-tag --epoch 20 --batch_size 4096 --lr 0.01
--keep 0.7 --process evaluate
```

```
Test RMSE: 0.5427
```



For the attentional factorization machines implementation on TensorFlow, interested readers can refer to the GitHub repo at `https://github.com/hexiangnan/attentional_factorization_machine`.

Now, to evaluate the `NeuralFM` model, just add the following line in the `main()` method in the `NeuralFM.py` script as follows:

```
# Model evaluation
print("RMSE: ")
print(model.evaluate(data.Test_data)) #evaluate on test set
>>>
RMSE:0.5578330373003925
```

RMSE is almost similar to FM. Now, let's look at how the test errors went per iteration:

```
# Plot test accuracy over time
plt.plot(epoch_list, test_err_list, 'r--', label='NFM test loss per
epoch', linewidth=4)
plt.title('NFM test loss per epoch')
plt.xlabel('Epoch')
plt.ylabel('Test loss')
plt.legend(loc='upper left')
plt.show()
```

The preceding code plots the test loss per iteration in the NFM model:

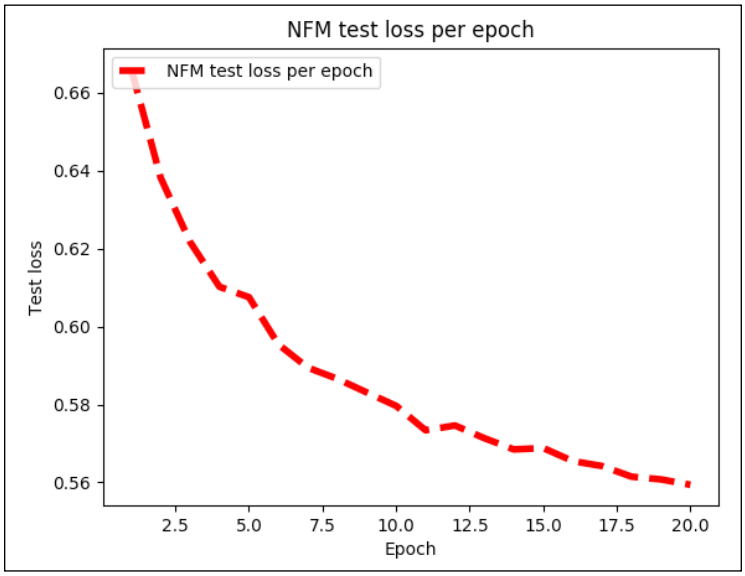


Figure 15: The test loss per iteration in the NFM model

Summary

In this chapter, we have discussed how to develop scalable recommendation systems with TensorFlow. We have seen some theoretical background of recommendation systems, such as gradient-based learning and back propagation and matrix factorization used in developing recommendation systems. Later in the chapter, we have also seen how to use MF, SVD, and k-means to develop a movie recommendation systems. Finally, we have seen how we can use Factorization Machines and its two other improved variations to develop more accurate recommendation systems that can handle large-scale sparse matrixes.

Reinforcement learning is an area of machine learning inspired by behavioral psychology, concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward. So, in short, RL is all about making the right actions at any given state.

Chapter 11, *Using Reinforcement Learning for Predictive Analytics*, is all about designing a machine learning system driven by criticisms and rewards. We will see how to apply reinforcement learning algorithms for the predictive model on real-life datasets.

11
Using Reinforcement Learning for Predictive Analytics

As a human being, we learn from past experiences. We haven't become so charming by accident. Years of positive compliments as well as negative criticism have all helped shape us into who we are today. We learn how to ride a bike by trying out different muscle movements until it just clicks. When you perform actions, you're sometimes rewarded immediately. This is all about **reinforcement learning** (**RL**).

This chapter is all about designing a machine learning system driven by criticisms and rewards. We will see how to apply reinforcement learning algorithms for the predictive model on real-life datasets.

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

- Reinforcement learning
- Reinforcement learning for predictive analytics
- Notation, policy, and utility in RL
- Developing a multiarmed bandit's predictive model
- Developing a stock price predictive model

Reinforcement learning

From a technical perspective, whereas supervised and unsupervised learning appears at opposite ends of the spectrum, RL exists somewhere in the middle. It's not supervised learning because the training data comes from the algorithm deciding between exploration and exploitation. And it's not unsupervised because the algorithm receives feedback from the environment. As long as you are in a situation where performing an action in a state produces a reward, you can use reinforcement learning to discover a good sequence of actions to take the maximum expected rewards.

The goal of an RL agent will be to maximize the total reward that it receives in the long run. The third main subelement is the `value` function.

While the rewards determine an immediate desirability of the states, the values indicate the long-term desirability of states, taking into account the states that may follow and the available rewards in these states. The `value` function is specified with respect to the chosen policy. During the learning phase, an agent tries actions that determine the states with the highest value, because these actions will get the best amount of reward in the long run.

Reinforcement learning in predictive analytics

Figure 1 shows a person making decisions to arrive at their destination. Moreover, suppose on your drive from home to work you always choose the same route. But one day your curiosity takes over and you decide to try a different path in hopes for a shorter commute. This dilemma of trying out new routes or sticking to the best-known route is an example of exploration versus exploitation:

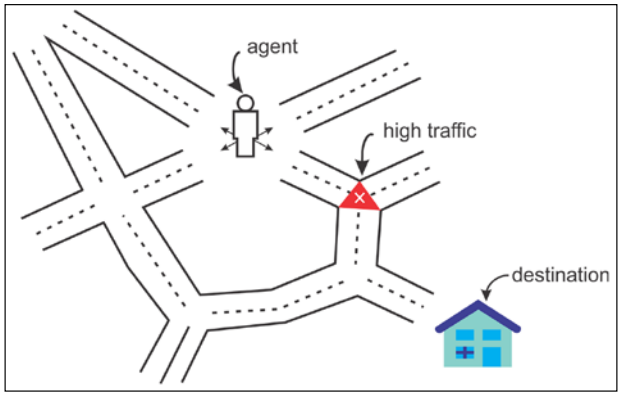


Figure 1: An agent always try to reach the destination passing through route

Reinforcement learning techniques are being used in many areas. A general idea that is being pursued right now is creating an algorithm that doesn't need anything apart from a description of its task. When this kind of performance is achieved, it will be applied virtually everywhere.

Notation, policy, and utility in RL

You may notice that reinforcement learning jargon involves anthropomorphizing the algorithm into taking actions in situations to receive rewards. In fact, the algorithm is often referred to as an agent that acts with the environment. You can just think of it like an intelligent hardware agent sensing with sensors and interacting with the environment using its actuators.

Therefore, it shouldn't be a surprise that much of RL theory is applied in robotics. Figure 2 demonstrates the interplay between states, actions, and rewards. If you start at state $\mathbf{s}_1$, you can perform action $\mathbf{a}_1$ to obtain a reward r ($\mathbf{s}_1$, $\mathbf{a}_1$). Actions are represented by arrows, and states are represented by circles:

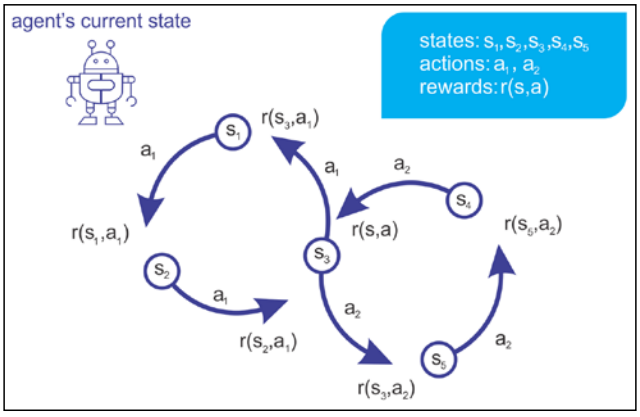


Figure 2: An agent is performing an action on a state produces a reward

A robot performs actions to change between different states. But how does it decide which action to take? Well, it's all about using different or a concrete policy.

Policy

In reinforcement learning lingo, we call the strategy a policy. The goal of reinforcement learning is to discover a good strategy. One of the most common ways to solve it is by observing the long-term consequences of actions in each state. The short-term consequence is easy to calculate: that's just the reward. Although performing an action yields an immediate reward, it's not always a good idea to greedily choose the action with the best reward. That's a lesson in life too because the most immediate best thing to do might not always be the most satisfying in the long run. The best possible policy is called the optimal policy, and it's often the holy grail of RL as shown in figure 3, which shows the optimal action given any state:

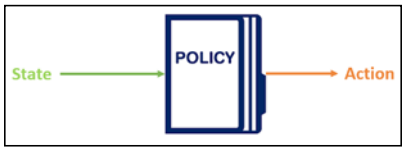


Figure 3: A policy defines an action to be taken in a given state

We've so far seen one type of policy where the agent always chooses the action with the greatest immediate reward, called a greedy policy. Another simple example of a policy is arbitrarily choosing an action, called random policy. If you come up with a policy to solve a reinforcement learning problem, it's often a good idea to double-check that your learned policy performs better than both the random and greedy policies.

In addition, we will also see how to develop another robust policy called policy gradients, where a neural network learns a policy for picking actions by adjusting its weights through gradient descent using feedback from the environment. We will see that although both the approaches are used, policy gradient is more direct and optimistic.

Utility

The long-term reward is called a utility. It turns out that, if we know the utility of performing an action at a state, then it's easy to solve reinforcement learning. For example, to decide which action to take, we simply select the action that produces the highest utility. However, uncovering these utility values is the harder part to be sorted out. The utility of performing an action *a* at a state *s* is written as a function *Q(s, a)*, called the utility function that predicts the expected immediate reward plus rewards following an optimal policy gave the state-action input which is shown in figure 4:

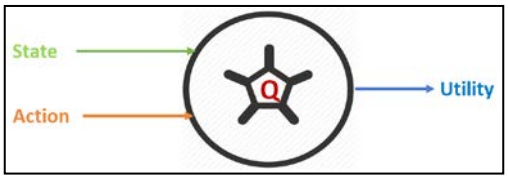


Figure 4: Using a utility function

Most reinforcement learning algorithms boil down to just three main steps: infer, do, and learn. During the first step, the algorithm selects the best action (*a*) given a state (*s*) using the knowledge it has so far. Next, it does the action to find out the reward (*r*) as well as the next state (*s'*). Then it improves its understanding of the world using the newly acquired knowledge *(s, r, a, s')*. However, this is just a naive way to calculate the utility; you would agree on this too.

Now, the question is what could be a more robust way to compute it? Here are two cents from my side. We can calculate the utility of a particular state-action pair *(s, a)* by recursively considering the utilities of future actions. The utility of your current action is influenced not just by the immediate reward, but also the next best action, as shown in the following formula:

$$Q(s,a) = r(s,a) + \gamma \max Q(s',a')$$

In the previous formula, s' denotes the next state, and a' denotes the next action. The reward of taking action *a* in state *s* is denoted by *r(s, a)*. Here, γ is a hyperparameter that you get to choose, called the discount factor. If γ is *0*, then the agent chooses the action that maximizes the immediate reward. Higher values of γ will make the agent put more importance in considering long-term consequences.

In practice, we have more hyperparameters to be considered. For example, if a vacuum cleaner robot is expected to learn to solve tasks quickly but not necessarily optimally, we might want to set a faster learning rate. Alternatively, if a robot is allowed more time to explore and exploit, we might tune down the learning rate. Let's call the learning rate *a*, and change our utility function as follows (note that when *a = 1*, both the equations are identical):

$$Q(s,a) \rightarrow Q(s,a) + \alpha\left(r(s,a) + \gamma \max Q(s',a') - Q(s,a)\right)$$

In summary, an RL problem can be solved if we know this *Q(s, a)* function. Here comes the machine learning strategy called neural networks, which are a way to approximate functions given enough training data. Also, TensorFlow is the perfect tool to deal with neural networks because it comes with many essential algorithms.

In the next two sections, we will see two examples of such implementation with TensorFlow. The first example is a naïve way of developing a multiarmed bandit agent for the predictive model. Then, the second example is a bit more advanced using neural network implementation for stock price prediction.

Developing a multiarmed bandit's predictive model

One of the simplest RL problems is called n-armed bandits. The thing is there are n-many slot machines but each has different fixed payout probability. The goal is to maximize the profit by always choosing the machine with the best payout.

As mentioned earlier, we will also see how to use policy gradient that produces explicit outputs. For our multiarmed bandits, we don't need to formalize these outputs on any particular state. To be simpler, we can design our network such that it will consist of just a set of weights that are corresponding to each of the possible arms to be pulled in the bandit. Then, we will represent how good an agent thinks to pull each arm to make maximum profit. A naive way is to initialize these weights to 1 so that the agent will be optimistic about each arm's potential reward.

To update the network, we can try choosing an arm with a greedy policy that we discussed earlier. Our policy is such that the agent receives a reward of either `1` or `-1` once it has issued an action. I know this is not a realistic imagination but most of the time the agent will choose an action randomly that corresponds to the largest expected value.

We will start developing a simple but effective bandit agent incrementally for solving multiarmed bandit problems. At first, there will be no state, that is, we will have a stateless agent. Then, we will see that using a stateless bandit agent to solve a complex problem is so biased that we cannot use it in real life.

Then we will increase the agent complexity by adding or converting the sample bandits into contextual bandits. The contextual bandits then will be a state full agent so can solve our predicting problem more efficiently. Finally, we will further increase the agent complexity by converting the textual bandits to full RL agent before deploying it:

1. Load the required library.

   Load the required library and packages/modules needed:

```
import tensorflow as tf
import tensorflow.contrib.slim as slim
import numpy as np
```

2. Defining bandits.

   For this example, I am using a four-armed bandit. The `getBandit` function that generates a random number from a normal distribution has the mean of 0. The lower the Bandit number, the more likely a positive reward will be awarded. As stated earlier, this is just a naïve but greedy way to train the agent so that it learns to choose a bandit that will generate not only the positive but also the maximum reward. Here I have listed the bandits so that Bandit 4 most often provides a positive reward:

```
def getBandit(bandit):
    ''
    This function creates the reward to the bandits on the basis
of randomly generated numbers. It then returns either a positive
or negative reward.
    ''
    random_number = np.random.randn(1)
    if random_number > bandit:
        return 1
    else:
        return -1
```

3. Developing an agent for the bandits.

   The following code creates a very simple neural agent consisting of a set of values for each of the `bandits`. Each value is estimated to be 1 based on the return value from the `bandits`. We use a policy gradient method to update the agent by moving the value for the selected action toward the received reward. At first, we need to reset the graph as follows:

```
tf.reset_default_graph()
```

   Then, the next two lines do the actual choosing by establishing the feed-forward part of the network:

```
weight_op = tf.Variable(tf.ones([num_bandits]))
action_op = tf.argmax(weight_op,0)
```

   Now, before starting the training process, we need to initiate the training process itself. Since we already know the reward, now it's time to feed them and choose an action in the network to compute the loss and use it to update the network:

```
reward_holder = tf.placeholder(shape=[1],dtype=tf.float32)
action_holder = tf.placeholder(shape=[1],dtype=tf.int32)
responsible_weight = tf.slice(weight_op,action_holder,[1])
```

We need to define the objective function that is `loss`:

```
loss = -(tf.log(responsible_weight)*reward_holder)
```

And then let's make the training process slow to make it exhaustive utilizing the learning rate:

```
LR = 0.001
```

We then use the gradient descent `optimizer` and instantiate the `training` operation:

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=LR)
training_op = optimizer.minimize(loss)
```

Now, it's time to define the training parameters such as a total number of iterations to train the agent, `reward` function, and a `random` action. The reward here sets the scoreboard for `bandits` to `0`, and by choosing a random action, we set the probability of taking a random action:

```
total_episodes = 10000
total_reward = np.zeros(num_bandits)
chance_of_random_action = 0.1
```

Finally, we initialize the global variables:

```
init_op = tf.global_variables_initializer()
```

4. Training the agent.

   We need to train the agent by taking actions to the environment and receiving rewards. We start by creating a TensorFlow session and launch the TensorFlow graph. Then, iterate the training process up to a total number of iterations. Then, we choose either a random act or one from the network. We then compute the reward from picking one of the `bandits`. Then, we make the training process consistent and update the network. Finally, we update the scoreboard:

```
with tf.Session() as sess:
    sess.run(init_op)
    i = 0
    while i < total_episodes:
        if np.random.rand(1) < chance_of_random_action:
            action = np.random.randint(num_bandits)
        else:
            action = sess.run(action_op)
                reward = getBandit(bandits[action])
            _,resp,ww = sess.run([training_op,responsible_
weight,weight_op], feed_dict={reward_holder:[reward],action_
holder:[action]})
```

```
        total_reward[action] += reward
        if i % 50 == 0:
            print("Running reward for all the " + str(num_bandits)
+ " bandits: " + str(total_reward))
        i+=1
```

Now let's evaluate the above model as follows:

```
print("The agent thinks bandit " + str(np.argmax(ww)+1) + " would
be the most efficient one.")
if np.argmax(ww) == np.argmax(-np.array(bandits)):
    print(" and it was right at the end!")
else:
    print(" and it was wrong at the end!")
>>>
```

The first iteration generates the following output:

```
Running reward for all the 4 bandits: [-1. 0. 0. 0.]
Running reward for all the 4 bandits: [ -1. -2. 14. 0.]
…
Running reward for all the 4 bandits: [ -15. -7. 340. 21.]
Running reward for all the 4 bandits: [ -15. -10. 364. 22.]
The agent thinks Bandit 3 would be the most efficient one and it
was wrong at the end!
```

The second iteration generates a different result as follows:

```
Running reward for all the 4 bandits: [ 1. 0. 0. 0.]
Running reward for all the 4 bandits: [ -1. 11. -3. 0.]
Running reward for all the 4 bandits: [ -2. 1. -2. 20.]
…
Running reward for all the 4 bandits: [ -7. -2. 8. 762.]
Running reward for all the 4 bandits: [ -8. -3. 8. 806.]
The agent thinks Bandit 4 would be the most efficient one and it
was right at the end!
```

Now that if you see the limitation of this agent being a stateless agent so randomly predicts which bandits to choose. In that situation, there are no environmental states, and the agent must simply learn to choose which action is best to take. To get rid of this problem, we can think of developing contextual bandits.

Using the contextual bandits, we can introduce and make the proper utilization of the state. The state consists of an explanation of the environment that the agent can use to make more intelligent and informed actions. The thing is that instead of using a single bandit we can chain multiple bandits together. So what would be the function of the state? Well, the state of the environment tells the agent to choose a bandit from the available list. On the other hand, the goal of the agent is to learn the best action for any number of bandits.

This way, the agent faces an issue since each bandit may have different reward probabilities for each arm and agent needs to learn how to perform an action on the state of the environment. Otherwise, the agent cannot achieve the maximum reward possible:

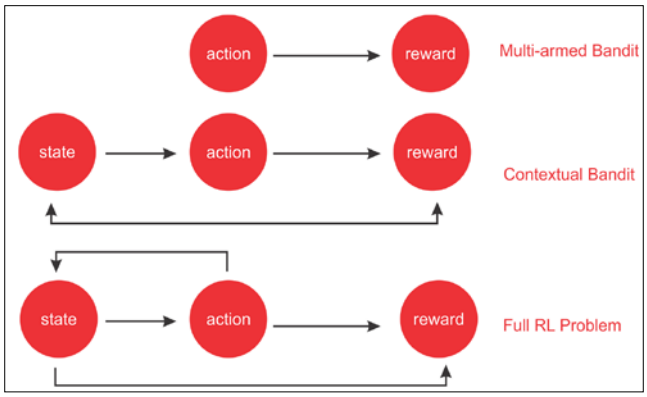


Figure 5: Stateless versus contextual bandits

As mentioned earlier, to get rid of this issue, we can build a single-layer neural network so that it can take a state and yield an action. Now, similar to the random bandits, we can use a policy-gradient update method too so that the network update is easier to take actions for maximizing the reward. This simplified way of posting an RL problem is referred to as the contextual bandit.

5. Developing contextual bandits.

   This example was adopted and extended based on "Simple Reinforcement Learning with TensorFlow Part 1.5: Contextual Bandits" By Arthur Juliani published at `https://medium.com/`.

   At first, let's define our contextual bandits. For this example, we will see how to use three four-armed bandits, that is, each Bandit has four arms that can be pulled to make an action. Since each bandit is contextual and has a state, so their arms have different success probabilities. This requires different actions to be performed to yield the best predictive result.

   Here, we define a class named `contextual_bandit()` consisting of a constructor and two user defined functions: `getBandit()` and `pullArm()`. The `getBandit()` function generates a random number from a normal distribution with a mean of `0`. The lower the Bandit number, the more likely a positive reward will be returned to be utilized. We want our agent to learn to choose the banditarm that will most often give a positive reward. Of course, it depends on the bandit presented. This constructor lists out all of our bandits. We assume the current state being armed `4`, `2`, `3`, and `1` that is the most optimal respectively.

Also, if you see carefully, most of the reinforcement learning algorithms follow similar implementation patterns. Thus, it's a good idea to create a class with the relevant methods to reference later, such as an abstract class or interface:

```
class contextualBandit():
    def __init__(self):
        self.state = 0
        self.bandits = np.array([[0.2,0,-0.0,-5], [0.1,-5,1,0.25],
[0.3,0.4,-5,0.5], [-5,5,5,5]])
        self.num_bandits = self.bandits.shape[0]
        self.num_actions = self.bandits.shape[1]
        def getBandit(self):
        '''
        This function returns a random state for each episode.
        '''
        self.state = np.random.randint(0, len(self.bandits))
        return self.state
        def pullArm(self,action):
        '''
        This function creates the reward to the bandits on the
basis of randomly generated numbers. It then returns either a
positive or negative reward that is action
        '''
        bandit = self.bandits[self.state, action]
        result = np.random.randn(1)
        if result > bandit:
            return 1
        else:
            return -1
```

6. Developing a policy-based agent.

   The following class `ContextualAgent` helps develop our simple, but very effective neural and contextual agent. We supply the current state as input and it then returns an action that is conditioned on the state of the environment. This is the most important step toward making a stateless agent a stateful one to be able to solve a full RL problem.

   Here, I tried to develop this agent such that it uses a single set of weights for choosing a particular arm given a bandit. The policy gradient method is used to update the agent by moving the value for a particular action toward achieving maximum reward:

```
class ContextualAgent():
    def __init__(self, lr, s_size,a_size):
        '''
```

```
        This function establishes the feed-forward part of the
network. The agent takes a state and produces an action -that is.
contextual agent
        '''
        self.state_in= tf.placeholder(shape=[1], dtype=tf.int32)
        state_in_OH = slim.one_hot_encoding(self.state_in, s_size)
        output = slim.fully_connected(state_in_OH, a_size,biases_
initializer=None, activation_fn=tf.nn.sigmoid, weights_
initializer=tf.ones_initializer())
        self.output = tf.reshape(output,[-1])
        self.chosen_action = tf.argmax(self.output,0)
        self.reward_holder = tf.placeholder(shape=[1], dtype=tf.
float32)
        self.action_holder = tf.placeholder(shape=[1], dtype=tf.
int32)
        self.responsible_weight = tf.slice(self.output, self.
action_holder,[1])
        self.loss = -(tf.log(self.responsible_weight)*self.reward_
holder)
        optimizer = tf.train.GradientDescentOptimizer(learning_
rate=lr)
        self.update = optimizer.minimize(self.loss)
```

7. Training the contextual bandit agent.

   At first, we clear the default TensorFlow graph:

   ```
   tf.reset_default_graph()
   ```

   Then, we define some parameters that will be used to train the agent:

   ```
   lrarning_rate = 0.001 # learning rate
   chance_of_random_action = 0.1 # Chance of a random action.
   max_iteration = 10000 #Max iteration to train the agent.
   ```

   Now, before starting the training, we need to load the bandits and then our agent:

   ```
   contextualBandit = contextualBandit() #Load the bandits.
   contextualAgent = ContextualAgent(lr=lrarning_rate, s_
   size=contextualBandit.num_bandits, a_size=contextualBandit.num_
   actions) #Load the agent.
   ```

   Now, to maximize the objective function toward total rewards, `weights` is used to evaluate to look into the network. We also set the scoreboard for `bandits` to `0` initially:

   ```
   weights = tf.trainable_variables()[0]
   total_reward = np.zeros([contextualBandit.num_
   bandits,contextualBandit.num_actions])
   ```

Then, we initialize all the variables using `global_variables_initializer()` function:

```
init_op = tf.global_variables_initializer()
```

Finally, we will start the training. The training is similar to the random one we have done in the preceding example. However, here the main objective of the training is to compute the mean reward for each of the bandits so that we can evaluate the agent's prediction accuracy later on by utilizing them:

```
with tf.Session() as sess:
    sess.run(init_op)
    i = 0
    while i < max_iteration:
        s = contextualBandit.getBandit() #Get a state from the
environment.
        #Choose a random action or one from our network.
        if np.random.rand(1) < chance_of_random_action:
            action = np.random.randint(contextualBandit.num_
actions)
        else:
            action = sess.run(contextualAgent.chosen_action,feed_
dict={contextualAgent.state_in:[s]})
        reward = contextualBandit.pullArm(action) #Get our reward
for taking an action given a bandit.
        #Update the network.
        feed_dict={contextualAgent.reward_
holder:[reward],contextualAgent.action_
holder:[action],contextualAgent.state_in:[s]}
        _,ww = sess.run([contextualAgent.update,weights], feed_
dict=feed_dict)
        #Update our running tally of scores.
        total_reward[s,action] += reward
        if i % 500 == 0:
            print("Mean reward for each of the " +
str(contextualBandit.num_bandits) + " bandits: " + str(np.
mean(total_reward,axis=1)))
        i+=1
>>>
Mean reward for each of the 4 bandits: [ 0. 0. -0.25 0. ]
Mean reward for each of the 4 bandits: [ 25.75 28.25 25.5 28.75]
…
Mean reward for each of the 4 bandits: [ 488.25 489. 473.5 440.5 ]
Mean reward for each of the 4 bandits: [ 518.75 520. 499.25
465.25]
Mean reward for each of the 4 bandits: [ 546.5 547.75 525.25
490.75]
```

8. Evaluating the agent.

   Now, that we have the mean reward for all the four bandits, it's time to utilize them to predict something interesting, that is, which bandit's arm will maximize the reward. Well, at first we can initialize some variables to estimate the prediction accuracy as well:

```
right_flag = 0
wrong_flag = 0
```

   Then let's start evaluating the agent's prediction performance:

```
for a in range(contextualBandit.num_bandits):
    print("The agent thinks action " + str(np.argmax(ww[a])+1) + "
for bandit " + str(a+1) + " would be the most efficient one.")
    if np.argmax(ww[a]) == np.argmin(contextualBandit.bandits[a]):
        right_flag += 1
        print(" and it was right at the end!")
    else:
        print(" and it was wrong at the end!")
        wrong_flag += 1
>>>
The agent thinks action 4 for Bandit 1 would be the most efficient
one and it was right at the end!
The agent thinks action 2 for Bandit 2 would be the most efficient
one and it was right at the end!
The agent thinks action 3 for Bandit 3 would be the most efficient
one and it was right at the end!
The agent thinks action 1 for Bandit 4 would be the most efficient
one and it was right at the end!
```

   As you can see, all the predictions made are right predictions. Now we can compute the accuracy as follows:

```
prediction_accuracy = (right_flag/right_flag+wrong_flag)
print("Prediction accuracy (%):", prediction_accuracy * 100)
>>>
Prediction accuracy (%): 100.0
```

Fantastic, well done! We have managed to design and develop a more robust bandit agent by means of a contextual agent that can accurately predict which arm, that is, the action of a bandit that would help to achieve the maximum reward, that is, profit.

In the next section, we will see another interesting but very useful application for stock price prediction, where we will see how to develop a policy-based Q Learning agent out of the box of the RL.

Developing a stock price predictive model

An emerging area for applying is the stock market trading, where a trader acts like a reinforcement agent since buying and selling (that is, action) particular stock changes the state of the trader by generating profit or loss, that is, reward. The following figure shows some of the most active stocks on July 15, 2017 (for an example):



Figure 6: `https://finance.yahoo.com/`

Now, we want to develop an intelligent agent that will predict stock prices such that a trader will buy at a low price and sell at a high price. However, this type of prediction is not so easy and is dependent on several parameters such as the current number of stocks, recent historical prices, and most importantly, on the available budget to be invested for buying and selling.

The states in this situation are a vector containing information about the current budget, current number of stocks, and a recent history of stock prices (the last 200 stock prices). So each state is a 202-dimensional vector. For simplicity, there are only three actions to be performed by a stock market agent: buy, sell, and hold.

So, we have the state and action, what else do you need? Policy, right? Yes, we should have a good policy, so based on that an action will be performed in a state. A simple policy can consist of the following rules:

- Buying (that is, action) a stock at the current stock price (that is, state) decreases the budget while incrementing the current stock count
- Selling a stock trades it in for money at the current share price
- Holding does neither, and performing this action simply waits for a particular time period and yields no reward

To find the stock prices, we can use the `yahoo_finance` library in Python. A general warning you might experience is "**HTTPError: HTTP Error 400: Bad Request**". But keep trying.

Now, let's try to get familiar with this module:

```
>>> from yahoo_finance import Share
>>> msoft = Share('MSFT')
>>> print(msoft.get_open())
72.24=
>>> print(msoft.get_price())
72.78
>>> print(msoft.get_trade_datetime())
2017-07-14 20:00:00 UTC+0000
>>>
```

So as of July 14, 2017, the stock price of Microsoft Inc. went higher, from 72.24 to 72.78, which means about a 7.5% increase. However, this small and just one-day data doesn't give us any significant information. But, at least we got to know the present state for this particular stock or instrument.

To install `yahoo_finance`, issue the following command:

```
$ sudo pip3 install yahoo_finance
```

Now it would be worth looking at the historical data. The following function helps us get the historical data for Microsoft Inc:

```
def get_prices(share_symbol, start_date, end_date, cache_filename):
    try:
        stock_prices = np.load(cache_filename)
    except IOError:
        share = Share(share_symbol)
        stock_hist = share.get_historical(start_date, end_date)
        stock_prices = [stock_price['Open'] for stock_price in stock_
hist]
        np.save(cache_filename, stock_prices)
    return stock_prices
```

The `get_prices()` method takes several parameters such as the share symbol of an instrument in the stock market, the opening date, and the end date. You will also like to specify and cache the historical data to avoid repeated downloading. Once you have downloaded the data, it's time to plot the data to get some insights.

The following function helps us to plot the price:

```
def plot_prices(prices):
    plt.title('Opening stock prices')
    plt.xlabel('day')
    plt.ylabel('price ($)')
    plt.plot(prices)
    plt.savefig('prices.png')
```

Now we can call these two functions by specifying a real argument as follows:

```
if __name__ == '__main__':
    prices = get_prices('MSFT', '2000-07-01', '2017-07-01',
'historical_stock_prices.npy')
    plot_prices(prices)
```

Here I have chosen a wide range for the historical data of 17 years to get a better insight. Now, let's take a look at the output of this data:

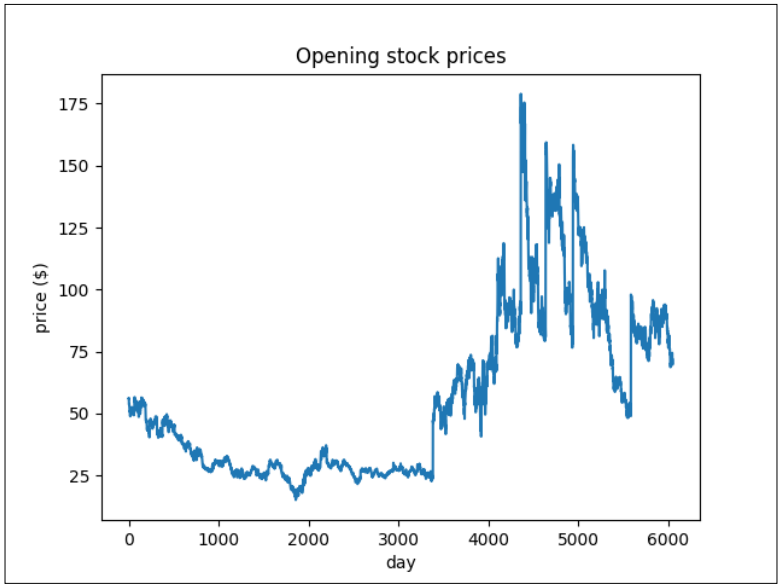


Figure 7: Historical stock price data of Microsoft Inc. from 2000 to 2017

The goal is to learn a policy that gains the maximum net worth from trading in the stock market. So what will a trading agent be achieving in the end? Figure 8 gives you some clue:

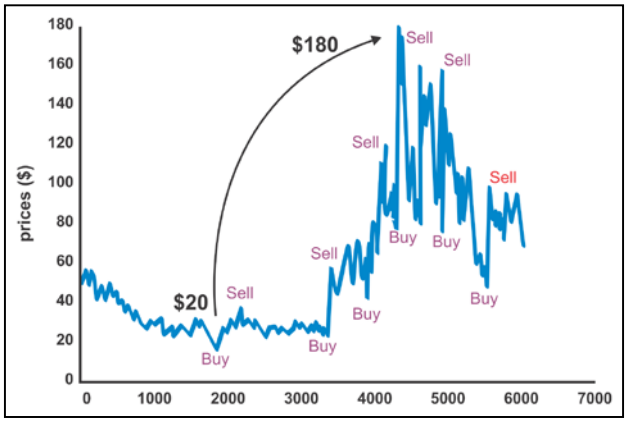


Figure 8: Some insight and a clue that shows, based on the current price, up to $160 profit can be made

Well, figure 8 shows that if the agent buys a certain instrument with price $20 and sells at a peak price say at $180, it will be able to make $160 reward, that is, profit. So, implementing such an intelligent agent using RL algorithms is a cool idea?

From the previous example, we have seen that for a successful RL agent, we need two operations well defined, which are as follows:

- How to select an action
- How to improve the utility Q-function

To be more specific, given a state, the decision policy will calculate the next action to take. On the other hand, improve Q-function from a new experience of taking an action.

Also, most reinforcement learning algorithms boil down to just three main steps: infer, perform, and learn. During the first step, the algorithm selects the best action (a) given a state (s) using the knowledge it has so far. Next, it performs the action to find out the reward (r) as well as the next state (s').

Then, it improves its understanding of the world using the newly acquired knowledge *(s, r, a, s')* as shown in the following figure:

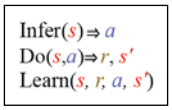


Figure 9: Steps to be performed for implementing an intelligent stock price prediction agent

Now, let's start implementing the decision policy based on which action will be taken for buying, selling, or holding a stock item. Again, we will do it an incremental way. At first, we will create a random decision policy and evaluate the agent's performance.

But before that, let's create an abstract class so that we can implement it accordingly:

```
class DecisionPolicy:
    def select_action(self, current_state, step):
        pass
    def update_q(self, state, action, reward, next_state):
        pass
```

The next task that can be performed is to inherit from this superclass to implement a random decision policy:

```
class RandomDecisionPolicy(DecisionPolicy):
    def __init__(self, actions):
        self.actions = actions
    def select_action(self, current_state, step):
        action = self.actions[random.randint(0, len(self.actions) -
1)]
        return action
```

The previous class did nothing except defining a function named `select_action()`, which will randomly pick an action without even looking at the state.

Now, if you would like to use this policy, you can run it on the real-world stock price data. This function takes care of exploration and exploitation at each interval of time, as shown in the following figure that form states S_1, S_2, and S_3. The policy suggests an action to be taken, which we may either choose to exploit or otherwise randomly explore another action. As we get rewards for performing an action, we can update the policy function over time:

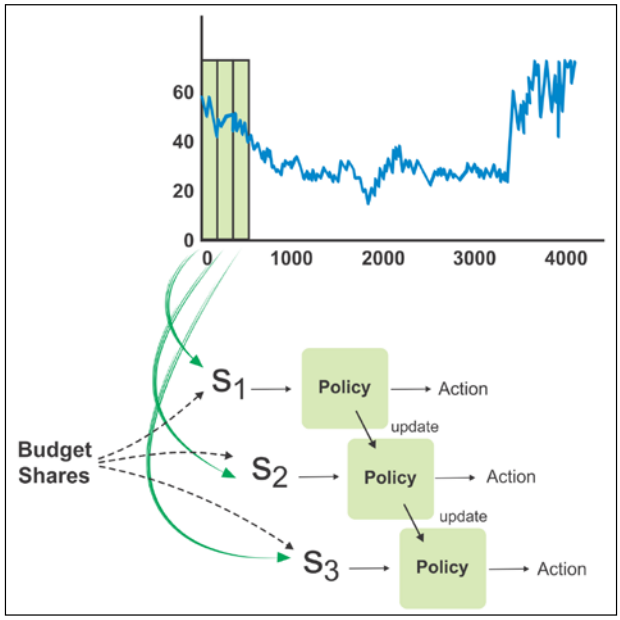


Figure 10: A rolling window of some size iterates through the stock prices over time

Fantastic, so we have the policy and now it's time to utilize this policy to make decisions and return the performance. Now, imagine a real scenario—suppose you're trading on Forex or ForTrade platform, then you can recall that you also need to compute the portfolio and the current profit or loss, that is, reward. Typically, these can be calculated as follows:

```
portfolio = budget + number of stocks * share value
reward = new_portfolio - current_portfolio
```

At first, we can initialize values that depend on computing the net worth of a portfolio, where the state is a `hist+2` dimensional vector. In our case, it would be 202 dimensional. Then we define the range of tuning the range up to:

Length of the prices selected by the user query – (history + 1), since we start from 0, we subtract 1 instead. Then, we should calculate the updated value of the portfolio and from the portfolio, we can calculate the value of the reward, that is, profit.

Also, we have already defined our random policy, so we can then select an action from the current policy. Then, we repeatedly update the portfolio values based on the action in each iteration and the new portfolio value after taking the action can be calculated. Then, we need to compute the reward from taking an action at a state. Nevertheless, we also need to update the policy after experiencing a new action. Finally, we compute the final portfolio worth:

```
def run_simulation(policy, initial_budget, initial_num_stocks, prices,
hist, debug=False):
    budget = initial_budget
    num_stocks = initial_num_stocks
    share_value = 0
    transitions = list()
    for i in range(len(prices) - hist - 1):
        if i % 100 == 0:
            print('progress {:.2f}%'.format(float(100*i) /
(len(prices) - hist - 1)))
        current_state = np.asmatrix(np.hstack((prices[i:i+hist],
budget, num_stocks)))
        current_portfolio = budget + num_stocks * share_value
        action = policy.select_action(current_state, i)
        share_value = float(prices[i + hist + 1])
        if action == 'Buy' and budget >= share_value:
            budget -= share_value
            num_stocks += 1
        elif action == 'Sell' and num_stocks > 0:
            budget += share_value
            num_stocks -= 1
        else:
            action = 'Hold'
        new_portfolio = budget + num_stocks * share_value
        reward = new_portfolio - current_portfolio
        next_state = np.asmatrix(np.hstack((prices[i+1:i+hist+1],
budget, num_stocks)))
        transitions.append((current_state, action, reward, next_
state))
        policy.update_q(current_state, action, reward, next_state)
    portfolio = budget + num_stocks * share_value
    if debug:
        print('${}\t{} shares'.format(budget, num_stocks))
    return portfolio
```

The previous simulation predicts a somewhat good result; however, it produces random results too often. Thus, to obtain a more robust measurement of success, let's run the simulation a couple of times and average the results. Doing so may take a while to complete, say 100 times, but the results will be more reliable:

```
def run_simulations(policy, budget, num_stocks, prices, hist):
    num_tries = 100
    final_portfolios = list()
    for i in range(num_tries):
        final_portfolio = run_simulation(policy, budget, num_stocks,
prices, hist)
        final_portfolios.append(final_portfolio)
    avg, std = np.mean(final_portfolios), np.std(final_portfolios)
    return avg, std
```

The previous function computes the average portfolio and the standard deviation by iterating the previous simulation function 100 times. Now, it's time to evaluate the previous agent. As already stated, there will be three possible actions to be taken by the stock trading agent such as buy, sell, and hold. We have a state vector of 202 dimension and budget only $1000. Then, the evaluation goes as follows:

```
actions = ['Buy', 'Sell', 'Hold']
    hist = 200
    policy = RandomDecisionPolicy(actions)
    budget = 1000.0
    num_stocks = 0
    avg,std=run_simulations(policy,budget,num_stocks,prices, hist)
    print(avg, std)
>>>
1512.87102405 682.427384814
```

The first one is the mean and the second one is the standard deviation of the final portfolio. So, our stock prediction agent predicts that as a trader you/we could make a profit about $513. Not bad. However, the problem is that since we have utilized a random decision policy, the result is not so reliable. To be more specific, the second execution will definitely produce a different result:

```
>>>
1518.12039077 603.15350649
```

Therefore, we should develop a more robust decision policy. Here comes the use of neural network-based QLearning for decision policy. Next, we will see a new hyperparameter epsilon to keep the solution from getting stuck when applying the same action over and over. The lesser its value, the more often it will randomly explore new actions:

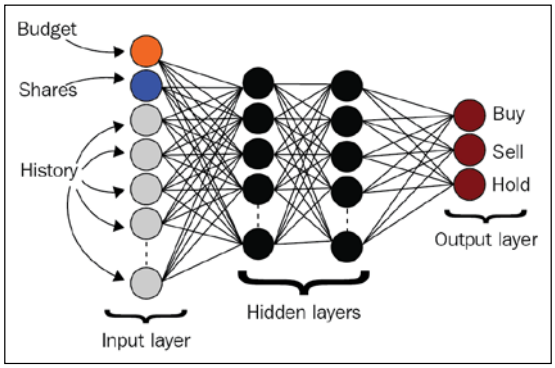


Figure 11: The input is the state space vector with three outputs, one for each output's Q-value

Next, I am going to write a class containing their functions:

- `Constructor`: This helps to set the hyperparameters from the Q-function. It also helps to set the number of hidden nodes in the neural networks. Once we have these two, it helps to define the input and output tensors. It then defines the structure of the neural network. Further, it defines the operations to compute the utility. Then, it uses an optimizer to update model parameters to minimize the loss and sets up the session and initializes variables.
- `select_action`: This function exploits the best option with probability 1-epsilon.
- `update_q`: This updates the Q-function by updating its model parameters.

Refer to the following code:

```
class QLearningDecisionPolicy(DecisionPolicy):
    def __init__(self, actions, input_dim):
        self.epsilon = 0.9
        self.gamma = 0.001
        self.actions = actions
        output_dim = len(actions)
        h1_dim = 200
        self.x = tf.placeholder(tf.float32, [None, input_dim])
        self.y = tf.placeholder(tf.float32, [output_dim])
        W1 = tf.Variable(tf.random_normal([input_dim, h1_dim]))
```

```
            b1 = tf.Variable(tf.constant(0.1, shape=[h1_dim]))
            h1 = tf.nn.relu(tf.matmul(self.x, W1) + b1)
            W2 = tf.Variable(tf.random_normal([h1_dim, output_dim]))
            b2 = tf.Variable(tf.constant(0.1, shape=[output_dim]))
            self.q = tf.nn.relu(tf.matmul(h1, W2) + b2)
            loss = tf.square(self.y - self.q)
            self.train_op = tf.train.GradientDescentOptimizer(0.01).
    minimize(loss)
            self.sess = tf.Session()
            self.sess.run(tf.initialize_all_variables())
        def select_action(self, current_state, step):
            threshold = min(self.epsilon, step / 1000.)
            if random.random() < threshold:
                # Exploit best option with probability epsilon
                action_q_vals = self.sess.run(self.q, feed_dict={self.x:
    current_state})
                action_idx = np.argmax(action_q_vals)
                action = self.actions[action_idx]
            else:
                # Random option with probability 1 - epsilon
                action = self.actions[random.randint(0, len(self.actions)
    - 1)]
            return action
        def update_q(self, state, action, reward, next_state):
            action_q_vals = self.sess.run(self.q, feed_dict={self.x:
    state})
            next_action_q_vals = self.sess.run(self.q, feed_dict={self.x:
    next_state})
            next_action_idx = np.argmax(next_action_q_vals)
            action_q_vals[0, next_action_idx] = reward + self.gamma *
    next_action_q_vals[0, next_action_idx]
            action_q_vals = np.squeeze(np.asarray(action_q_vals))
            self.sess.run(self.train_op, feed_dict={self.x: state, self.y:
    action_q_vals})
```

Summary

In this chapter, we have discussed a wonderful field of machine learning called reinforcement learning with TensorFlow. We have discussed it from the theoretical as well as practical point of view. Reinforcement learning is the natural tool when a problem can be framed by states that change due to actions that can be taken by an agent to discover rewards. There are three primary steps in implementing the algorithm: infer the best action from the current state, perform the action, and learn from the results.

We have seen how to implement RL agents for making predictions by knowing the `action`, `state`, `policy`, and `utility` functions. We have seen how to develop RL-based agents using random policy as well as neural network-based QLearning policy. QLearning is an approach to solve reinforcement learning, where you develop an algorithm to approximate the utility function (`Q`-function). Once a good enough approximation is found, you can start inferring best actions to take from each state. In particular, we have seen two step-by-step examples that show how we could develop a multiarmed bandit agent and a stock price prediction agent with very good accuracy. But, be advised that the actual stock market is a much more complicated beast, and the techniques used in this chapter generalize too many situations.

This is more or less the end of our little journey with TensorFlow. Throughout the chapters, I tried to provide you with several examples of how to use TensorFlow efficiently for developing predictive models for predictive analytics. During the writing of this book, I had to keep many constraints in my mind, for example, the page count, API availability, and my expertise. But I tried to make the book more or less simpler and also tried to avoid details of the theory since you can read them from openly available books, blogs, and the TensorFlow website itself.

Nevertheless, I have also used some APIs from the TensorFlow `contribs` package. Naturally, some of them might be updated or even deleted from the code base. But don't worry I will keep the code of this book updated on my GitHub repo at `https://github.com/PacktPublishing/Predictive-Analytics-with-TensorFlow`. Feel free to open a new issue or any pull request for the betterment of this book. So it would be better to keep tuned.

I would like to say thanks for buying and enjoying this book!

Index

Symbols

1m MovieLens dataset
reference link 416

A

accelerated computing developer program
URL 83
activation functions 258
Adjusted Rand Index (ARI) 257
alternating least squares (ALS) 414
Analog to Digital Converter (ADC) 172
Area Under Curve (AUC) 256
Artificial Neural Networks 245-248
attentional factorization machine (AFM) 454
axon 245

B

Bag-of-words. *See* **BOW**
Basic Linear Algebra Subroutines (BLAS) 82
basic probability
for predictive modeling 54
Bayes' rule 58, 59
Bazel
URL, for installation 85
bias neuron 247
biological neurons 245
BOW
dataset description 202, 203
dataset exploration 202
issues, defining 202
spam prediction, LR used with TensorFlow 203-214
used, for predictive analytics (PA) 201
used, for spam prediction with TensorFlow 203-214
BRNN
about 359, 360
used, for image classification 364-371

C

continuous bag-of-words (CBOW)
about 225
model building 227-235
reusing, for predicting sentiment 235-240
used, for model building 227
used, for word embedding 227
central limit theorem (CLT)
about 44-46
data distribution 46
skewness 46
Chi-square independence test 53
Chi-square test 52
clustering 158-160
clustering audio files
predictive models 170-182
CNN-based predictive model
CNN, used for predictive analytics (PA) about reviews 302-317
for sentiment analysis 300, 301
movie and product review datasets, exploring 301
CNN hyperparameters
tuning 296-299
CNN model
architecture design 318-327
dataset description 317, 333
for emotion recognition 317

testing, on image 328-333
used, for predictive analytics (PA) 333
CNN predictive model
for image classification 334-355
collaborative filtering approach
about 412
dataset description 416
exploratory analysis, of dataset 418-423
for movie recommendations 414
utility matrix 414, 415
collaborative filtering (CF) 446
conditional entropy 62
conditional probability
about 56
chain rule 57
content-based filtering approach 413
continuous skip-gram 226
contrastive divergence 276
conversion procedure
reference link 456
convolutional neural networks (CNN)
about 288, 289
architecture 289, 290
convolutional operations
about 290
applying, in TensorFlow 291, 292
convolutional operations, parameters
data_format 292
filter 292
input 291
padding 292
strides 292
use_cudnn_on_gpu 292
Cornell University (CU) 301
correlation 49, 50
covariance 49, 50
CUDA toolkit
reference link 82
cuDNN v5.1 library
URL, for downloading 82
curse of dimensionality 7

D

data distribution 46
data, in TensorFlow
reference link 102
data model
data type 97-99
feeds 102
fetches 101
in TensorFlow 93
placeholders 102
rank 96
shape 97
tensors 94-96
variables 100
dataset description
about 261, 262, 378, 416
bank-additional.csv 261
bank-additional-full.csv 261
bank.csv 261
bank-full.csv 261
movies data 416
ratings data 416
URL, for downloading 378
user data 417
datasets, for DSCI
about 425
reference link 162
data type 97-99
data value chain
for creating decisions 70, 71
DBN
construction 277
unsupervised pretraining 277-279
Decision trees (DTs) 151
deep belief networks
about 273, 274
DBN, construction 277
reference link 280
Restricted Boltzmann Machines 274-276
used, for predictive analytics (PA) 279-285
deep learning
for predictive analytics (PA) 244, 245
Deep Neural Networks
about 248
architecture 248-250
performance analysis 253-257
dendrites 245
Distributed TensorFlow
reference link 297
DNN
drawbacks 288, 289

E

entropy
  about 61
  conditional entropy 62
  joint entropy 62
  Shannon entropy 61
estimators 196
estimator transformer 197, 198
expectation 43, 44
exploratory analysis
  of dataset 418-423

F

factorization machines
  about 446, 447
  dataset description 449
  FM model, implementing 452
  formulation 448, 449
  for recommendation systems 445
  improving, for predictive analytics (PA) 454
  neural factorization machines 455
  preprocessing 450, 451
  problem definition 448, 449
Fandango
  reference link 119
Fast Fourier Transforms (FFT) 81
feedforward neural network (FFNN) 249
fetches 101
fine-tuning DNN hyperparameters
  about 257
  activation functions 258
  number of hidden layers 257
  number of neurons per hidden layer 258
  regularization 259
  weight and biases initialization 258
forwarding propagation step 251

G

Gated Recurrent Unit (GRU) 358
graphs 21
GRU cell 363, 364

H

hidden layers 250
hybrid recommendation systems
  collaborative filtering 414
  content-based filtering 414
hypothesis testing
  about 51
  Chi-square independence test 53
  Chi-square test 52

I

information gain 62
information theory
  using 63, 64
  using, in predictive modeling 60
  using, in Python 64-66
input neurons 247
interquartile range (IQR) 50, 51

J

joint entropy 62
Jupyter Notebook
  reference link 99

K

Kaggle
  URL 71
K-means
  betweenness 167
  cluster assignment step 161
  totwithiness 167
  update step 161
  used, for predicting neighborhoods 162-169
  used, for predictive analytics (PA) 160
  withiness 167
  working 160, 161
kNN
  kNN-based predictive model, implementing 183-190
  used, for predictive analytics (PA) 182
  working principles 182, 183

L

Latent Drichilet Allocation (LDA) 195
latent factors (LFs) 414
least squares fitting (LSF) 7
libcupti-dev library
  installing 83
linear algebra
  about 7, 8
  Linear algebra package (LAPACK) 8
  NumPy 8
  Pandas 8
  programming 8
  SciPy 8
linear independence 29, 31
linear regression
  about 105-117
  problem statement 118, 119
  source code 111, 113
  used, for movie rating prediction 119-131
linear threshold unit (LTU) 246
Logistic Regression (LR) 115
log-likelihood function 252
Long Short-Term Memory (LSTM) 358
LSTM model evaluation 408-410
LSTM model training 389-406
LSTM networks 360-362
LSTM predictive model
  about 382-384
  for sentiment analysis 388
  LSTM model evaluation 408-410
  LSTM model training 389-406
  network design 388
  TensorBoard, visualizing 406, 407

M

machine learning (ML) 157
marginal probability 56
Markov Chain Monte Carlo (MCMC) 276
Markov Random Fields (MRF) 273
mathematical explanation
  reference link 24
matrices
  about 21, 25
  determinant of matrix, finding 27
  eigenvalues 29
  eigenvectors 29
  matrix addition 25
  multiplying 26
  simultaneous linear equations, solving 28
  transpose of a matrix, finding 27
matrix factorization (FM) 413
matrix subtraction 25
Mean Squared Error (MSE) 187
model-based collaborative filtering 414
model evaluation 384-388
MovieLens data
  reference link 427
movie recommendation engine
  implementing 424
  model, training with ratings 424-433
  movie rating prediction, by users 439, 440
  movies, clustering 435-438
  recommendation system, evaluating 442-445
  saved model, inferencing 433
  top K movies, finding 440
  top K movies, predicting 441
  user-item table, generating 434, 435
  user-user similarity, computing 442
movie review dataset
  references 227
multiarmed bandit's predictive model
  developing 468-476
multilayer perceptron
  about 250
  MLP, training 251
  MLP, used 252, 253
  preprocessing 263, 264
  TensorFlow implementation, of MLP 265-272
multilayer perceptrons
  dataset description 261, 262
  used, for predictive analytics (PA) 260
mutual information 61

N

network design
  about 388
  embedding layer 388
  RNN layer 388
  softmax or sigmoid layer 388

neural factorization machine (NFM)
about 454, 455
dataset description 456
model evaluation 459
model training 456-459
used, for movie recommendations 456
Neural Networks (NNs) 182
N-grams 200
NLP analytics pipelines
about 194, 195
text analytics, used 195, 196
nonparametric model
versus parametric model 41
nonparametric predictive models 42
notation
in RL 465
number of hidden layers 257
number of neurons per hidden layer 258
numerical linear algebra (NLA) 8
NVIDIA CUDA
installing 82
NVIDIA cuDNN v5.1+
installing 82
NVIDIA Graph Analytics Library (nvGRAPH) 82

P

padding operations
about 293, 294
subsampling operations, applying in TensorFlow 294-296
parametric model
versus nonparametric model 41
parametric predictive models 42
perceptron 246
placeholder
about 92
data type 92
name 93
shape 92
policy
in RL 465, 466
pooling layer 293, 294
population 42, 43
predictive analytics (PA)
about 1-3
BOW, used 201
Clustering with mean-NN classification 300
deep belief networks, used 279-285
deep learning 244, 245
K-means, used 160
kNN, used 182
Naive Bayes 300
N-grams 300
One-hot (sequence) vectors 300
predictive model, working principles 3-7
supervised learning 116, 117
Support Vector Machine (SVM) 300
predictive analytics tools
in Python 37, 38
predictive model
dataset, description 378
developing, for time series data 378
exploratory analysis 379-381
for clustering audio files 170-182
LSTM predictive model 382-384
model evaluation 384-388
preprocessing 379-381
predictive modeling
information theory, using in 60
principal component analysis 31, 32
probability 54
probability density functions (PDF) 56
probability distributions
about 55, 56
conditional independence 58
independence 58
probability mass functions (PMF) 56
Python
about 13
classes 20, 21
data types 14
dictionary, used 17
functions 19, 20
information theory, using in 64-67
installing 9
installing, on Linux 11
installing, on macOS 12, 13
installing, on PIP 12
installing, on Windows 9-11
list, used 15
obtaining 9
packages, installing 13

predictive analytics tools 37, 38
reference link 9
set, used 18, 19
string, used 14, 15
tuple, used 16
upgrading, on PIP 12
Python programming
reference link 21

R

random numbers
generating 55
random sampling 43
random variables 54
rank 96
recommendation systems
about 411
cold start 412
collaborative filtering approaches 412
content-based filtering approach 413
factorization machines 445
hybrid recommendation systems 413
in factorization machines 447, 448
model-based collaborative filtering 414
scalability 412
sparsity 413
rectified linear unit (ReLU) 248
regularization
about 259
dropout 259
L2 regularization 259
max-norm constraints 259
reinforcement learning (RL)
about 464
in predictive analytics 464
requests package
reference link 185
Restricted Boltzmann Machines (RBM) 273-276
RNN
implementing, for spam prediction 371-377
RNN architecture
about 358, 359
BRNN 359, 360
contextual information 358, 359
GRU cell 363, 364
LSTM networks 360-362
Root Mean Squared Error (RMSE) 256
R script
reference link 169

S

sample
about 42, 43
expectation 43, 44
random sampling 43
seed
setting 55
self-information
about 60
mutual information 61
sentiment analysis
CNN-based predictive model 300, 301
LSTM predictive model 388
Shannon entropy 61
shape 97
signals 246
singular value decomposition (SVD)
about 33, 34, 413
used, for data compression in predictive model 34-37
skewness 46
Spam dataset
reference link 202
span 29, 31
squared error function 252
Staked Auto-Encoders (SAEs) 249
standard deviation (SD) 47, 48
standard transformer 196
statistical model
about 41
nonparametric model, versus parametric model 41
nonparametric predictive models 42
parametric predictive models 42
statistics
central limit theorem (CLT) 44-46
hypothesis testing 51
interquartile range (IQR) 50, 51
population 42, 43
quartiles 50, 51
sample 42, 43

standard deviation (SD) 47, 48
used, in predictive modeling 40
variance 47, 48
stock price predictive model
developing 477-485
StopWordsRemover 199
supervised learning
for predictive analytics (PA) 116, 117
Support Vector Machines (SVMs) 115, 182, 247
synaptic terminals 246

T

telodendrian 246
TensorBoard
about 103
visualizing 406, 407
working 104
TensorFlow
computational graph 87-90
configuring 78
convolutional operations, applying 291, 292
data model 93
Dataset API 102
easy debugging 77
easy use 78
extensibility 78
faster computing 77
feeding 102
flexibility 77
GPU computing, transparency 78
installation, testing 87
installing 78, 83
installing, from source 85, 86
installing, on Linux 79
installing, with native pip 83
installing, with virtualenv 84, 85
normal 88
nVidia driver, installing 80
overview 76
placeholders 89
portability 77
preloaded data 102
production scale 78
programming model 90-93
Python, installing 80
reading, from files 102
session 89
special 88
supported 78
tensors 89
tf.Operation objects 88
tf.Tensor object 88
unified API 77
variables 89
wide adoption 78
with CPU support 79
with GPU support 79
TensorFlow-based implementation of FM
reference link 449
TensorFlow contrib
reference link 141
using 141-146
TensorFlow Implementation, of Neural Factorization Machine
reference link 456
TensorFlow, on macOS
URL, for installation 79
TensorFlow, on Windows
URL, for installation 79
TensorFlow Python package
reference link 84
tensors
about 21, 93-96
reference link 94
text analytics
event extraction 196
named entity recognition 196
sentiment analysis 195
term frequency - inverse document frequency (TF-IDF) 195
topic modelling 195
using 195, 196
TF-IDF model
for predictive analytics (PA) 214
IDF, computing 215-218
implementing, for spam prediction 218-224
TF, computing 215-218
TFIDF, computing 215-218
tf.Tensor
reference link 93
time series data
predictive model, developing 378

titanic dataset
about 70, 131, 132
comparative analysis 155, 156
data value chain, for creating decisions 70, 71
ensemble method, for survival prediction 151-155
example 71-76
exploratory analysis 132-137
feature engineering 137-139
linear SVM, for survival prediction 146-150
logistic regression, for survival prediction 140
TensorFlow contrib, using 141-146
transformers
about 196
estimator transformer 197, 198
N-grams 200
standard transformer 196
StopWordsRemover 199

U

UCI Machine Learning repository
reference link 261
unsupervised learning 158-160
unsupervised pretraining 277-279
utility
in RL 465-467
utility matrix 414, 415

V

variables 100
variance 47, 48
vectors 21-25

W

weight and biases initialization
about 258
biases, initializing 258
small random numbers 258
zero initialization, avoiding 258
within-cluster sums of squares (WCSS) 256
Word2vec
CBOW, used, for model building 227
CBOW, used, for word embedding 227
continuous bag-of-words 225
continuous skip-gram 226
used, for sentiment analysis 225