



Machine Learning using Python

Manaranjan Pradhan | U Dinesh Kumar



भारतीय प्रबंध संस्थान बैंगलूर
INDIAN INSTITUTE OF MANAGEMENT
BANGALORE

WILEY

Machine Learning using Python

|

|

|

|

Machine Learning using Python

Manaranjan Pradhan
Consultant
Indian Institute of Management Bangalore

U Dinesh Kumar
Professor
Indian Institute of Management Bangalore

WILEY

Machine Learning using Python

Copyright © 2019 by Wiley India Pvt. Ltd., 4436/7, Ansari Road, Daryaganj, New Delhi-110002.

Cover Image: © Getty Images

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, electronic, mechanical, photocopying, recording or scanning without the written permission of the publisher.

Limits of Liability: While the publisher and the author have used their best efforts in preparing this book, Wiley and the author make no representation or warranties with respect to the accuracy or completeness of the contents of this book, and specifically disclaim any implied warranties of merchantability or fitness for any particular purpose. There are no warranties which extend beyond the descriptions contained in this paragraph. No warranty may be created or extended by sales representatives or written sales materials. The accuracy and completeness of the information provided herein and the opinions stated herein are not guaranteed or warranted to produce any particular results, and the advice and strategies contained herein may not be suitable for every individual. Neither Wiley India nor the author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

Disclaimer: The contents of this book have been checked for accuracy. Since deviations cannot be precluded entirely, Wiley or its author cannot guarantee full agreement. As the book is intended for educational purpose, Wiley or its author shall not be responsible for any errors, omissions or damages arising out of the use of the information contained in the book. This publication is designed to provide accurate and authoritative information with regard to the subject matter covered. It is sold on the understanding that the Publisher is not engaged in rendering professional services.

Trademarks: All brand names and product names used in this book are trademarks, registered trademarks, or trade names of their respective holders. All logos are standard trademarks of the companies. Wiley is not associated with any product or vendor mentioned in this book.

Other Wiley Editorial Offices:

John Wiley & Sons, Inc. 111 River Street, Hoboken, NJ 07030, USA

Wiley-VCH Verlag GmbH, Pappellaee 3, D-69469 Weinheim, Germany

John Wiley & Sons Australia Ltd, 42 McDougall Street, Milton, Queensland 4064, Australia

John Wiley & Sons (Asia) Pte Ltd, 1 Fusionopolis Walk #07-01 Solaris, South Tower, Singapore 138628

John Wiley & Sons Canada Ltd, 22 Worcester Road, Etobicoke, Ontario, Canada, M9W 1L1

First Edition: 2019

ISBN: 978-81-265-7990-7

ISBN: 978-81-265-8855-8 (ebk)

www.wileyindia.com

Printed at:

Dedication

To my parents, Janaradan and Nayana Pradhan.

– Manaranjan Pradhan

Haritha Saranga, Prathusha Dinesh and to the one who said:

“Learn from the water: Loud splashes the Brook but the Oceans Depth are Calm.”

– U Dinesh Kumar



Preface

Artificial Intelligence (AI) has emerged as one of the decisive expertise with applications across various industry domains. Machine learning (ML), a subset of AI, is an important set of algorithms used for solving several business and social problems. The objective of this book is to discuss machine learning model development using Python. AI and ML are important skills that every graduate in engineering and management discipline will require to advance in their career. Companies across different industry sectors such as retail, finance, insurance, manufacturing, healthcare are planning to build and offer AI-based solutions to stay competitive in the market. Companies across different domains will need a large pool of experts in these fields to build products and solutions for them. An estimate by IDC (International Data Corporation) states that spending by organizations on AI and ML will grow from \$12B in 2017 to \$57.6B in 2021 (Columbus, 2018)¹. Almost 61% of organizations have reported that Machine Learning and Artificial Intelligence will be among their top data initiatives in 2019. Machine Learning is going to be most in-demand AI skill for data scientists (Zafarino, 2018)². An article published in *The Economic Times* in 2018 states that India's demand for data scientist grew by 400%, but in contrast the supply has only increased by 19% (Pramanik, 2018)³.

Python is the leading programming language used by several organizations for creating end-to-end solutions using ML. Several universities and institutions have started undergraduate and post-graduate programs in business analytics, and one of the core offerings is skill development in machine learning. But there is a shortage of quality teaching material that provides insights into application of machine learning using real-world examples. This book is an attempt to create hands-on content for structured learning. This is a practitioner's book that both students and industry professionals can use to understand and master skills in machine learning using Python.

We have been teaching Business Analytics, Artificial Intelligence and Machine Learning at the Indian Institute of Management, Bangalore, for MBA students and several other leading corporates in India for many years. This book is a result of learnings on how to make the students understand machine learning in an efficient manner. For better understanding, we have reproduced many concepts and examples in this book from the book titled *Business Analytics – The Science of Data-Driven Decision*, by U Dinesh Kumar published by Wiley in 2017. We recommend the readers to read this book along with the book by U Dinesh Kumar (co-author of this book).

We have acknowledged the sources of data used for developing ML models using Python in respective chapters. We are thankful to the original creators of the dataset.

1 L Columbus, "Roundup of Machine Learning Forecasts and Market Estimates", Forbes, February 2018.

2 S Zafarino, "The Outlook for Machine Learning in Tech: ML and AI Skills in High Demand", CIO, July 27, 2018.

3 A Pramanik, "India's Demand for Data Scientists Grow over 400%: Report", The Economic Times, July 10 2018.

ORGANIZATION OF CHAPTERS

The book consists of 10 chapters. The sequence of the chapters is designed to create strong foundation for the learners. The first few chapters provide the foundations in Python and ML and the later chapters build on the concepts learnt in the previous chapters. We suggest readers to read the chapters in sequence for a structured learning.

Chapter Number	Topic	Description
Chapter 1	Introduction to Machine Learning using Python	This chapter discusses concepts such as artificial intelligence, machine learning and deep learning and a framework for building machine learning models. The chapter also explains why Python has been chosen as the language for building machine learning models. Also, provides a quick overview of Python language and how to get started.
Chapter 2	Descriptive Analytics	Introduces the concept of DataFrame and working with DataFrames such as selecting, filtering, grouping, joining, etc. Also introduces statistical plots such as histogram, distribution plot, scatter plot, heatmap and so on and how to derive insights from these plots.
Chapter 3	Probability Distributions and Hypothesis Tests	Provides overview of probability distributions such as normal distribution, exponential distribution, Poisson distribution, and their applications to real-world problems. We also discuss hypotheses tests such as <i>t</i> -test, paired <i>t</i> -test and ANOVA in this chapter.
Chapter 4	Regression	This chapter explains how to build regression models. Provides in-depth understanding of model evaluation, validation and accuracy measures such as RMSE, R-squared.
Chapter 5	Classification Problems	This chapter discusses classification problems, formulating classification problems and applying classification models such as logistic regression and decision trees. Provides in-depth understanding of evaluating the models using confusion matrix, ROC AUC score, precision, recall, and finding optimal cut-off for model deployment.
Chapter 6	Advanced Machine Learning	Understanding cost function and usage of optimization techniques such as gradient descent in machine learning algorithms. Deep dive into application of advanced models such as KNN, Random Forest, Bagging, and Boosting. Also provides insight into hyperparameter search using grid search techniques.
Chapter 7	Clustering	Creating clusters or segments using clustering techniques like <i>K</i> -Means and Hierarchical Clustering. Introduces the concept of distance measures such as Euclidean distance, cosine similarity and Jaccard coefficient.
Chapter 8	Forecasting	Deep dive into time-series forecasting models such as simple moving average, exponential moving average and ARIMA models. Discusses the concepts of time-series stationarity and accuracy measures such as RMSE and MAPE.
Chapter 9	Recommender Systems	Discusses how a customer's purchase behaviour can be used to predict or recommend what he/she is likely to buy next. Provides overview of building recommender systems using association rules, collaborative filtering, and matrix factorization techniques.
Chapter 10	Text Analytics	Discusses steps for preparing text data (natural languages) using techniques like TF and TF-IDF vectorization, stop word removal, stemming, or lemmatization. Discusses the concepts through an example of sentiment classification using Naïve–Bayes model.

BOOK RESOURCES

The datasets and codes for the book are available at [https://www.wileyindia.com/catalog/product/view/
id/6915/s/machine-learning-using-python/](https://www.wileyindia.com/catalog/product/view/id/6915/s/machine-learning-using-python/). The learners should practice the examples given in each chapter, while reading to reinforce the concepts.

**Manaranjan Pradhan
U Dinesh Kumar**



Acknowledgements

It is a great privilege to be part of a reputed institute like the Indian Institute of Management Bangalore (IIMB). This association has given us opportunity to interact with many great minds and scholars from Industry and Academia, which helped us shape the structure and content of this book. We would like to thank IIMB for the opportunities and for the permission to publish this book under IIMB brand.

This textbook could not have been possible without the help of our colleagues. We would like to extend our thanks to Aayushi Kalra, Bharat Paturi, Dhimant Ganatra, Dinesh Kumar Vaitheeswaran, Gaurav Kumar, Manupriya Agrawal, Purvi Tiwari, Rahul Kumar, Satyabala Hariharan, Shailaja Grover, Sharada Sringeswara, and Sunil for their assistance in preparation of this book. Their feedback and review comments were extremely valuable in improving structure, content, and quality of the book.

We would also like to thank Meenakshi Sehrawat of Wiley for editing and providing several valuable suggestions to improve the quality of the book.

**Manaranjan Pradhan
Dinesh Kumar**

On a personal note, I would like to extend my gratitude to Prof. Dinesh Kumar for his vision, guidance, and persistence in writing this book and choosing me as a co-author. He has been a great teacher and mentor to me and I hope to continue learning from him in future.

Manaranjan Pradhan



Table of Contents

Preface	vii
Acknowledgements	xi

1 | INTRODUCTION TO MACHINE LEARNING

1.1 Introduction to Analytics and Machine Learning	1
1.2 Why Machine Learning?	3
1.3 Framework for Developing Machine Learning Models	3
1.4 Why Python?	5
1.5 Python Stack for Data Science	7
1.6 Getting Started with Anaconda Platform	8
1.7 Introduction to Python	12
1.7.1 Declaring Variables	12
1.7.2 Conditional Statements	13
1.7.3 Generating Sequence Numbers	14
1.7.4 Control Flow Statements	15
1.7.5 Functions	16
1.7.6 Working with Collections	17
1.7.6.1 List	17
1.7.6.2 Tuples	19
1.7.6.3 Set	21
1.7.6.4 Dictionary	22
1.7.7 Dealing with Strings	23
1.7.8 Functional Programming	24
1.7.8.1 Example 1: Map	24
1.7.8.2 Example 2: Filter	25
1.7.9 Modules and Packages	25
1.7.10 Other features	26
Further Reading	27
References	27

2 | DESCRIPTIVE ANALYTICS

2.1 Working with DataFrames in Python	29
2.1.1 IPL Dataset Description using DataFrame in Python	30
2.1.2 Loading Dataset into Pandas DataFrame	31
2.1.3 Displaying First Few Records of the DataFrame	33

2.1.4	Finding Summary of the DataFrame	33
2.1.5	Slicing and Indexing of DataFrame	36
2.1.6	Value Counts and Cross Tabulations	38
2.1.7	Sorting DataFrame by Column Values	39
2.1.8	Creating New Columns	40
2.1.9	Grouping and Aggregating	41
2.1.10	Joining DataFrames	42
2.1.11	Re-Naming Columns	43
2.1.12	Applying Operations to Multiple Columns	43
2.1.13	Filtering Records Based on Conditions	44
2.1.14	Removing a Column or a Row from a Dataset	45
2.2	Handling Missing Values	45
2.3	Exploration of Data using Visualization	48
2.3.1	Drawing Plots	49
2.3.2	Bar Chart	49
2.3.3	Histogram	50
2.3.4	Distribution or Density Plot	51
2.3.5	Box Plot	52
2.3.6	Comparing Distributions	54
2.3.7	Scatter Plot	56
2.3.8	Pair Plot	57
2.3.9	Correlation and Heatmap	58
Conclusion		59
Exercises		60
References		61

3 | PROBABILITY DISTRIBUTIONS AND HYPOTHESIS TESTS

3.1	Overview	63
3.2	Probability Theory – Terminology	63
3.2.1	Random Experiment	63
3.2.2	Sample Space	64
3.2.3	Event	64
3.3	Random Variables	64
3.4	Binomial Distribution	65
3.4.1	Example of Binomial Distribution	66
3.5	Poisson Distribution	68
3.5.1	Example of Poisson Distribution	68
3.6	Exponential Distribution	69
3.6.1	Example of Exponential Distribution	70
3.7	Normal Distribution	72
3.7.1	Example of Normal Distribution	72
3.7.2	Mean and Variance	77
3.7.3	Confidence Interval	78
3.7.4	Cumulative Probability Distribution	79
3.7.5	Other Important Distributions	80

3.8	Central Limit Theorem	80
3.9	Hypothesis Test	81
3.9.1	Z-test	82
3.9.2	One-Sample <i>t</i> -Test	83
3.9.3	Two-Sample <i>t</i> -Test	85
3.9.4	Paired Sample <i>t</i> -Test	88
3.9.5	Chi-Square Goodness of Fit Test	89
3.10	Analysis of Variance (ANOVA)	90
3.10.1	Example of One-Way ANOVA	91
	Conclusion	92
	Exercises	93
	References	94

4 | LINEAR REGRESSION

4.1	Simple Linear Regression	95
4.2	Steps in Building a Regression Model	96
4.3	Building Simple Linear Regression Model	97
4.3.1	Creating Feature Set (<i>X</i>) and Outcome Variable (<i>Y</i>)	101
4.3.2	Splitting the Dataset into Training and Validation Sets	102
4.3.3	Fitting the Model	102
4.3.3.1	<i>Printing Estimated Parameters and Interpreting Them</i>	102
4.3.3.2	<i>Complete Code for Building Regression Model</i>	103
4.4	Model Diagnostics	103
4.4.1	Co-efficient of Determination (<i>R</i> -Squared or <i>R</i> ²)	103
4.4.2	Hypothesis Test for the Regression Co-efficient	104
4.4.3	Analysis of Variance (ANOVA) in Regression Analysis	105
4.4.4	Regression Model Summary Using Python	105
4.4.5	Residual Analysis	106
4.4.5.1	<i>Check for Normal Distribution of Residual</i>	106
4.4.5.2	<i>Test of Homoscedasticity</i>	107
4.4.6	Outlier Analysis	108
4.4.6.1	<i>Z-Score</i>	108
4.4.6.2	<i>Cook's Distance</i>	109
4.4.6.3	<i>Leverage Values</i>	110
4.4.7	Making Prediction and Measuring Accuracy	110
4.4.7.1	<i>Predicting using the Validation Set</i>	111
4.4.7.2	<i>Finding R-Squared and RMSE</i>	111
4.4.7.3	<i>Calculating Prediction Intervals</i>	111
4.5	Multiple Linear Regression	112
4.5.1	Predicting the SOLD PRICE (Auction Price) of Players	113
4.5.2	Developing Multiple Linear Regression Model Using Python	114
4.5.2.1	<i>Loading the Dataset</i>	114
4.5.2.2	<i>Displaying the First Five Records</i>	115

4.5.3	Categorical Encoding Features	116
4.5.4	Splitting the Dataset into Train and Validation Sets	117
4.5.5	Building the Model on a Training Dataset	118
4.5.6	Multi-Collinearity and Handling Multi-Collinearity	119
4.5.6.1	<i>Variance Inflation Factor (VIF)</i>	120
4.5.6.2	<i>Checking Correlation of Columns with Large VIFs</i>	121
4.5.6.3	<i>Building a New Model after Removing Multi-collinearity</i>	124
4.5.7	Residual Analysis in Multiple Linear Regression	126
4.5.7.1	<i>Test for Normality of Residuals (P-P Plot)</i>	126
4.5.7.2	<i>Residual Plot for Homoscedasticity and Model Specification</i>	127
4.5.8	Detecting Influencers	127
4.5.9	Transforming Response Variable	129
4.5.10	Making Predictions on the Validation Set	130
4.5.10.1	<i>Measuring RMSE</i>	131
4.5.10.2	<i>Measuring R-squared Value</i>	131
4.5.10.3	<i>Auto-correlation Between Error Terms</i>	131
Conclusion		131
Exercises		132
References		134

5 CLASSIFICATION PROBLEMS

5.1	Classification Overview	135
5.2	Binary Logistic Regression	136
5.3	Credit Classification	137
5.3.1	Encoding Categorical Features	141
5.3.2	Splitting Dataset into Training and Test Sets	143
5.3.3	Building Logistic Regression Model	143
5.3.4	Printing Model Summary	143
5.3.5	Model Diagnostics	145
5.3.6	Predicting on Test Data	147
5.3.7	Creating a Confusion Matrix	148
5.3.8	Measuring Accuracies	150
5.3.9	Receiver Operating Characteristic (ROC) and Area Under the Curve (AUC)	151
5.3.10	Finding Optimal Classification Cut-off	153
5.3.10.1	<i>Youden's Index</i>	153
5.3.10.2	<i>Cost-Based Approach</i>	155
5.4	Gain Chart and Lift Chart	156
5.4.1	Loading and Preparing the Dataset	158
5.4.2	Building the Logistic Regression Model	160
5.5	Classification Tree (Decision Tree Learning)	166
5.5.1	Splitting the Dataset	167
5.5.2	Building Decision Tree Classifier using Gini Criteria	168
5.5.3	Measuring Test Accuracy	168
5.5.4	Displaying the Tree	168

5.5.5	Understanding Gini Impurity	169
5.5.6	Building Decision Tree using Entropy Criteria	171
5.5.7	Finding Optimal Criteria and Max Depth	173
5.5.8	Benefits of Decision Tree	174
Conclusion		175
Exercises		175
References		177

6 | ADVANCED MACHINE LEARNING

6.1	Overview	179
6.1.1	How Machines Learn?	180
6.2	Gradient Descent Algorithm	180
6.2.1	Developing a Gradient Descent Algorithm for Linear Regression Model	182
6.2.1.1	<i>Loading the Dataset</i>	182
6.2.1.2	<i>Set X and Y Variables</i>	183
6.2.1.3	<i>Standardize X and Y</i>	183
6.2.1.4	<i>Implementing the Gradient Descent Algorithm</i>	183
6.2.1.5	<i>Finding the Optimal Bias and Weights</i>	186
6.2.1.6	<i>Plotting the Cost Function against the Iterations</i>	188
6.3	Scikit-Learn Library for Machine Learning	190
6.3.1	Steps for Building Machine Learning Models	191
6.3.1.1	<i>Splitting Dataset into Train and Test Datasets</i>	191
6.3.1.2	<i>Building Linear Regression Model with Train Dataset</i>	191
6.3.1.3	<i>Making Prediction on Test Set</i>	193
6.3.1.4	<i>Measuring Accuracy</i>	193
6.3.2	Bias-Variance Trade-off	194
6.3.3	K-Fold Cross-Validation	200
6.4	Advanced Regression Models	201
6.4.1	Building Linear Regression Model	201
6.4.1.1	<i>Loading IPL Dataset</i>	201
6.4.1.2	<i>Standardization of X and Y</i>	203
6.4.1.3	<i>Split the Dataset into Train and Test</i>	203
6.4.1.4	<i>Build the Model</i>	204
6.4.1.5	<i>Plotting the Coefficient Values</i>	205
6.4.1.6	<i>Calculate RMSE</i>	206
6.4.2	Applying Regularization	206
6.4.2.1	<i>Ridge Regression</i>	208
6.4.2.2	<i>LASSO Regression</i>	208
6.4.2.3	<i>Elastic Net Regression</i>	209
6.5	Advanced Machine Learning Algorithms	210
6.5.1	Dealing with Imbalanced Datasets	211
6.5.2	Logistic Regression Model	214
6.5.2.1	<i>Building the Model</i>	214
6.5.2.2	<i>Confusion Matrix</i>	214

6.5.2.3 Classification Report	216
6.5.2.4 Receiver Operating Characteristic Curve (ROC) and Area under ROC (AUC) Score	216
6.5.3 K-Nearest Neighbors (KNN) Algorithm	219
6.5.3.1 KNN Accuracy	221
6.5.3.2 GridSearch for Optimal Parameters	223
6.5.4 Ensemble Methods	225
6.5.5 Random Forest	226
6.5.5.1 Building Random Forest Model	226
6.5.5.2 Grid Search for Optimal Parameters	227
6.5.5.3 Building the Final Model with Optimal Parameter Values	229
6.5.5.4 ROC AUC Score	229
6.5.5.5 Drawing the Confusion Matrix	229
6.5.5.6 Finding Important Features	231
6.5.6 Boosting	233
6.5.6.1 AdaBoost	233
6.5.6.2 Gradient Boosting	234
Conclusion	239
Exercises	239
References	243

7 | CLUSTERING

7.1 Overview	245
7.2 How Does Clustering Work?	246
7.2.1 Finding Similarities Using Distances	247
7.2.1.1 Euclidean Distance	247
7.2.1.2 Other Distance Metrics	248
7.3 K-Means Clustering	248
7.3.1 Plotting Customers with Their Segments	249
7.3.2 Normalizing Features	250
7.3.3 Cluster Centers and Interpreting the Clusters	251
7.4 Creating Product Segments Using Clustering	252
7.4.1 Beer Dataset	252
7.4.2 How Many Clusters Exist?	253
7.4.2.1 Using Dendrogram	253
7.4.2.2 Finding Optimal Number of Clusters Using Elbow Curve Method	255
7.4.2.3 Normalizing the Features	256
7.4.3 Creating Clusters	256
7.4.4 Interpreting the Clusters	256
7.5 Hierarchical Clustering	258
7.5.1 Compare the Clusters Created by K-Means and Hierarchical Clustering	258
Conclusion	259
Exercises	260
References	261

8 | FORECASTING

8.1	Forecasting Overview	263
8.2	Components of Time-Series Data	264
8.3	Moving Average	265
8.3.1	Loading and Visualizing the Time-Series Dataset	265
8.3.2	Forecasting Using Moving Average	266
8.3.3	Calculating Forecast Accuracy	268
8.3.3.1	<i>Mean Absolute Percentage Error</i>	268
8.3.3.2	<i>Root Mean Square Error</i>	269
8.3.4	Exponential Smoothing	269
8.4	Decomposing Time Series	271
8.5	Auto-Regressive Integrated Moving Average Models	273
8.5.1	Auto-Regressive (AR) Models	273
8.5.1.1	<i>ACF</i>	273
8.5.1.2	<i>PACF</i>	273
8.5.1.3	<i>Building AR Model</i>	276
8.5.1.4	<i>Forecast and Measure Accuracy</i>	277
8.5.2	Moving Average (MA) Processes	277
8.5.3	ARMA Model	278
8.5.4	ARIMA Model	279
8.5.4.1	<i>What is Stationary Data?</i>	280
8.5.4.2	<i>Dicky-Fuller Test</i>	282
8.5.4.3	<i>Differencing</i>	283
8.5.4.4	<i>Forecast and Measure Accuracy</i>	287
Conclusion		287
Exercises		288
References		289

9 | RECOMMENDER SYSTEMS

9.1	Overview	291
9.1.1	Datasets	291
9.2	Association Rules (Association Rule Mining)	292
9.2.1	Metrics	293
9.2.1.1	<i>Support</i>	293
9.2.1.2	<i>Confidence</i>	294
9.2.1.3	<i>Lift</i>	294
9.2.2	Applying Association Rules	294
9.2.2.1	<i>Loading the Dataset</i>	294
9.2.2.2	<i>Encoding the Transactions</i>	295
9.2.2.3	<i>Generating Association Rules</i>	296
9.2.2.4	<i>Top Ten Rules</i>	298
9.2.2.5	<i>Pros and Cons of Association Rule Mining</i>	299
9.3	Collaborative Filtering	299
9.3.1	How to Find Similarity between Users?	299

9.3.2	User-Based Similarity	300
9.3.2.1	<i>Loading the Dataset</i>	301
9.3.2.2	<i>Calculating Cosine Similarity between Users</i>	303
9.3.2.3	<i>Filtering Similar Users</i>	304
9.3.2.4	<i>Loading the Movies Dataset</i>	305
9.3.2.5	<i>Finding Common Movies of Similar Users</i>	305
9.3.2.6	<i>Challenges with User-Based Similarity</i>	306
9.3.3	Item-Based Similarity	307
9.3.3.1	<i>Calculating Cosine Similarity between Movies</i>	307
9.3.3.2	<i>Finding Most Similar Movies</i>	308
9.4	Using Surprise Library	309
9.4.1	User-Based Similarity Algorithm	310
9.4.2	Finding the Best Model	311
9.4.3	Making Predictions	312
9.5	Matrix Factorization	313
Conclusion		314
Exercises		314
References		316

10 | TEXT ANALYTICS

10.1	Overview	317
10.2	Sentiment Classification	317
10.2.1	<i>Loading the Dataset</i>	318
10.2.2	<i>Exploring the Dataset</i>	319
10.2.3	Text Pre-processing	320
10.2.3.1	<i>Bag-of-Words (BoW) Model</i>	320
10.2.3.2	<i>Creating Count Vectors for sentiment_train Dataset</i>	322
10.2.3.3	<i>Displaying Document Vectors</i>	324
10.2.3.4	<i>Removing Low-frequency Words</i>	325
10.2.3.5	<i>Removing Stop Words</i>	327
10.2.3.6	<i>Creating Count Vectors</i>	327
10.2.3.7	<i>Distribution of Words Across Different Sentiment</i>	330
10.3	Naïve-Bayes Model for Sentiment Classification	331
10.3.1	<i>Split the Dataset</i>	332
10.3.2	<i>Build Naïve-Bayes Model</i>	333
10.3.3	<i>Make Prediction on Test Case</i>	333
10.3.4	<i>Finding Model Accuracy</i>	333
10.4	Using TF-IDF Vectorizer	334
10.5	Challenges of Text Analytics	335
10.5.1	<i>Using n-Grams</i>	335
10.5.2	<i>Build the Model Using n-Grams</i>	336
Conclusion		337
Exercises		337
References		338
Index		339

CHAPTER

1

Introduction to Machine Learning

LEARNING OBJECTIVES

After completing this chapter, you will be able to

- Understand how machine learning is used to solve problems in real world.
- Understand types of machine learning algorithms and framework for building machine learning models.
- Learn why Python has been widely adopted as a platform for building machine learning models.
- Learn what are the key libraries in Python and their purpose.
- Learn how to get started with setting up the Anaconda platform.
- Learn the basic features of Python language to get started with machine learning tasks.

1.1 | INTRODUCTION TO ANALYTICS AND MACHINE LEARNING

Analytics is a collection of techniques and tools used for creating value from data. Techniques include concepts such as artificial intelligence (AI), machine learning (ML), and deep learning (DL) algorithms. AI, ML, and DL are defined as follows:

1. **Artificial Intelligence:** Algorithms and systems that exhibit human-like intelligence.
2. **Machine Learning:** Subset of AI that can learn to perform a task with extracted data and/or models.
3. **Deep Learning:** Subset of machine learning that imitate the functioning of human brain to solve problems.

The relationship between AI, ML, and DL can be visualized as shown in Figure 1.1

The relationship between AI, ML, and DL shown in Figure 1.1 is not accepted by all. There is another school of thought that believes that AI and ML are different (ML is not a subset of AI) with some overlap. The important point is that all of them are algorithms, which are nothing but set of instructions used for solving business and social problems.

Machine learning is a set of algorithms that have the capability to learn to perform tasks such as prediction and classification effectively using data. Learning is achieved using additional data and/or additional models. An algorithm can be called a learning algorithm when it improves on a performance metric while performing a task, for example, accuracy of classification such as fraud, customer churn, and so on. The focus of this book is machine learning algorithms. In the next few sections we discuss various machine learning algorithms and how to solve a problem using ML algorithms.

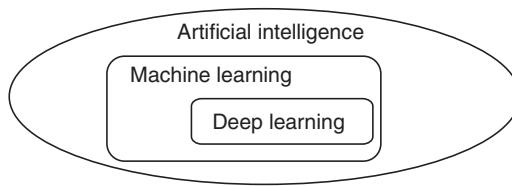


FIGURE 1.1 Relationship between artificial intelligence, machine learning, and deep learning.

Machine learning algorithms are classified into four categories as defined below:

1. **Supervised Learning Algorithms:** These algorithms require the knowledge of both the outcome variable (dependent variable) and the features (independent variable or input variables). The algorithm learns (i.e., estimates the values of the model parameters or feature weights) by defining a loss function which is usually a function of the difference between the predicted value and actual value of the outcome variable. Algorithms such as linear regression, logistic regression, discriminant analysis are examples of supervised learning algorithms. In the case of multiple linear regression, the regression parameters are estimated by minimizing the sum of squared errors, which is given by $\sum_{i=1}^n (y_i - \hat{y}_i)^2$, where y_i is the actual value of the outcome variable, \hat{y}_i is the predicted value of the outcome variable, and n is the total number of records in the data. Here the predicted value is a linear or a non-linear function of the features (or independent variables) in the data. The prediction is achieved (by estimating feature weights) with the knowledge of the actual values of the outcome variables, thus called supervised learning algorithms. That is, the supervision is achieved using the knowledge of outcome variable values.
2. **Unsupervised Learning Algorithms:** These algorithms are set of algorithms which do not have the knowledge of the outcome variable in the dataset. The algorithms must find the possible values of the outcome variable. Algorithms such as clustering, principal component analysis are examples of unsupervised learning algorithms. Since the values of outcome variable are unknown in the training data, supervision using that knowledge is not possible.
3. **Reinforcement Learning Algorithms:** In many datasets, there could be uncertainty around both input as well as the output variables. For example, consider the case of spell check in various text editors. If a person types “buutiful” in Microsoft Word, the spell check in Microsoft Word will immediately identify this as a spelling mistake and give options such as “beautiful”, “bountiful”, and “dutiful”. Here the prediction is not one single value, but a set of values. Another definition is: Reinforcement learning algorithms are algorithms that have to take sequential actions (decisions) to maximize a cumulative reward. Techniques such as Markov chain and Markov decision process are examples of reinforcement learning algorithms.
4. **Evolutionary Learning Algorithms:** Evolutional algorithms are algorithms that imitate natural evolution to solve a problem. Techniques such as genetic algorithm and ant colony optimization fall under the category of evolutionary learning algorithms.

In this book, we will be discussing several supervised and unsupervised learning algorithms.

1.2 | WHY MACHINE LEARNING?

Organizations across the world use several performance measures such as return on investment (ROI), market share, customer retention, sales growth, customer satisfaction, and so on for quantifying, monitoring, benchmarking, and improving. Organizations would like to understand the association between key performance indicators (KPIs) and factors that have a significant impact on the KPIs for effective management. Knowledge of the relationship between KPIs and factors would provide the decision maker with appropriate actionable items (U D Kumar, 2017). Machine learning algorithms can be used for identifying the factors that influence the key performance indicators, which can be further used for decision making and value creation. Organizations such as Amazon, Apple, Capital One, General Electric, Google, IBM, Facebook, Procter and Gamble and so on use ML algorithms to create new products and solutions. ML can create significant value for organizations if used properly. MacKenzie et al. (2013) reported that Amazon's recommender systems resulted in a sales increase of 35%.

A typical ML algorithm uses the following steps:

1. Identify the problem or opportunity for value creation.
2. Identify sources of data (primary as well secondary data sources) and create a data lake (integrated data set from different sources).
3. Pre-process the data for issues such as missing and incorrect data. Generate derived variables (feature engineering) and transform the data if necessary. Prepare the data for ML model building.
4. Divide the datasets into subsets of training and validation datasets.
5. Build ML models and identify the best model(s) using model performance in validation data.
6. Implement Solution/Decision/Develop Product.

1.3 | FRAMEWORK FOR DEVELOPING MACHINE LEARNING MODELS

The framework for ML algorithm development can be divided into five integrated stages: problem and opportunity identification, collection of relevant data, data pre-processing, ML model building, and model deployment. The various activities carried out during these different stages are described in Figure 1.2. The success of ML projects will depend on how innovatively the data is used by the organization as compared to the mechanical use of ML tools. Although there are several routine ML projects such as customer segmentation, clustering, forecasting, and so on, highly successful companies blend innovation with ML algorithms.

The success of ML projects will depend on the following activities:

1. **Feature Extraction:** Feature extraction is a process of extracting features from different sources. For a given problem, it is important to identify the features or independent variables that may be necessary for building the ML algorithm. Organizations store data captured by them in enterprise resource planning (ERP) systems, but there is no guarantee that the organization would have identified all important features while designing the ERP system. It is also possible that the problem being addressed using the ML algorithm may require data that is not captured by the organization. For example, consider a company that is interested in predicting the warranty cost for the vehicle manufactured by them. The number of warranty claims may depend on weather conditions such as rainfall, humidity, and so on. In many cases, feature extraction itself can be an iterative process.

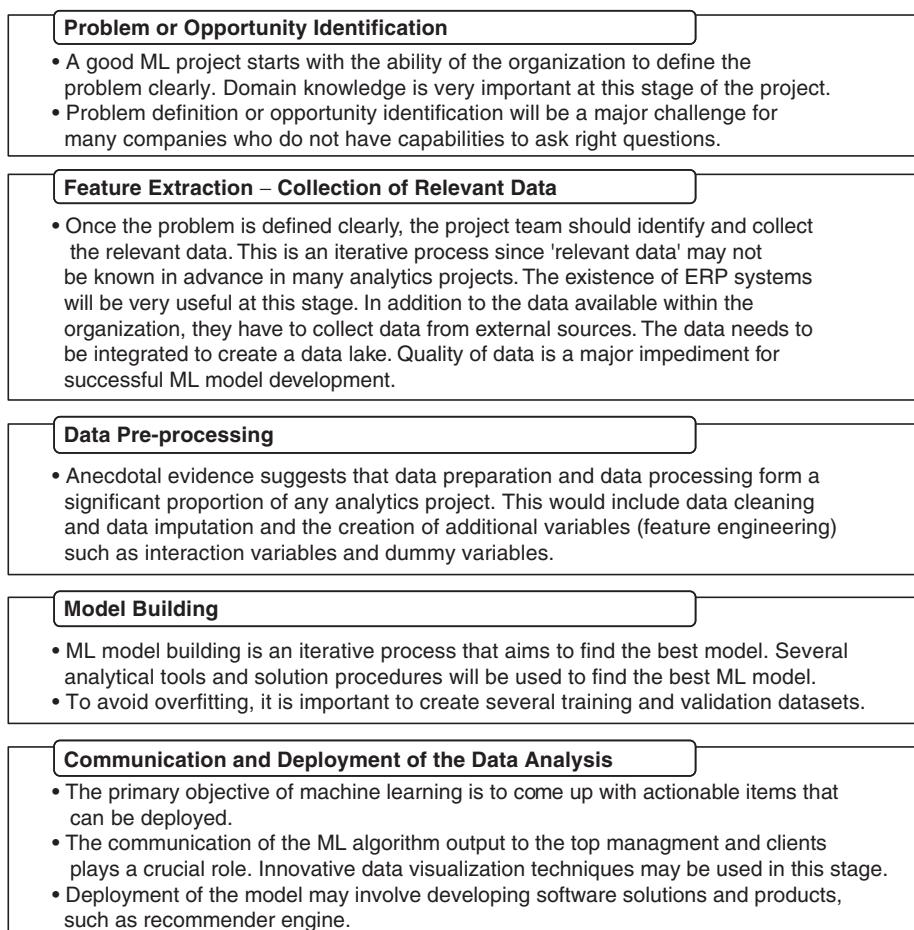


FIGURE 1.2 Framework of ML model development.

2. **Feature Engineering:** Once the data is made available (after feature extraction), an important step in machine learning is feature engineering. The model developer should decide how he/she would like to use the data that has been captured by deriving new features. For example, if X_1 and X_2 are two features that are captured in the original data. We can derive new features by taking ratio (X_1/X_2) and product (X_1X_2). There are many other innovative ways of deriving new features such as binning continuous variable, centring the data (deviation from the mean), and so on. The success of the ML model may depend on feature engineering.
3. **Model Building and Feature Selection:** During model building, the objective is to identify the model that is more suitable for the given problem context. The selected model may not be always the most accurate model, as accurate model may take more time to compute and may require expensive infrastructure. The final model for deployment will be based on multiple criteria such as accuracy, computing speed, cost of deployment, and so on. As a part of model building, we will also go through feature selection which identifies important features that have significant relationship with the outcome variable.

4. **Model Deployment:** Once the final model is chosen, then the organization must decide the strategy for model deployment. Model deployment can be in the form of simple business rules, chatbots, real-time actions, robots, and so on.

In the next few sections we will discuss about why Python has become one of most widely adopted language for machine learning, what features and libraries are available in Python, and how to get started with Python language.

1.4 | WHY PYTHON?

Python is an interpreted, high-level, general-purpose programming language (Downey, 2012). One of the key design philosophy of Python is code readability. Ease of use and high productivity have made Python very popular. Based on the number of question views on StackOverflow (Kauflin, 2017), as shown in Figure 1.3, Python seems to have gained attention and popularity significantly compared to other languages since 2012.

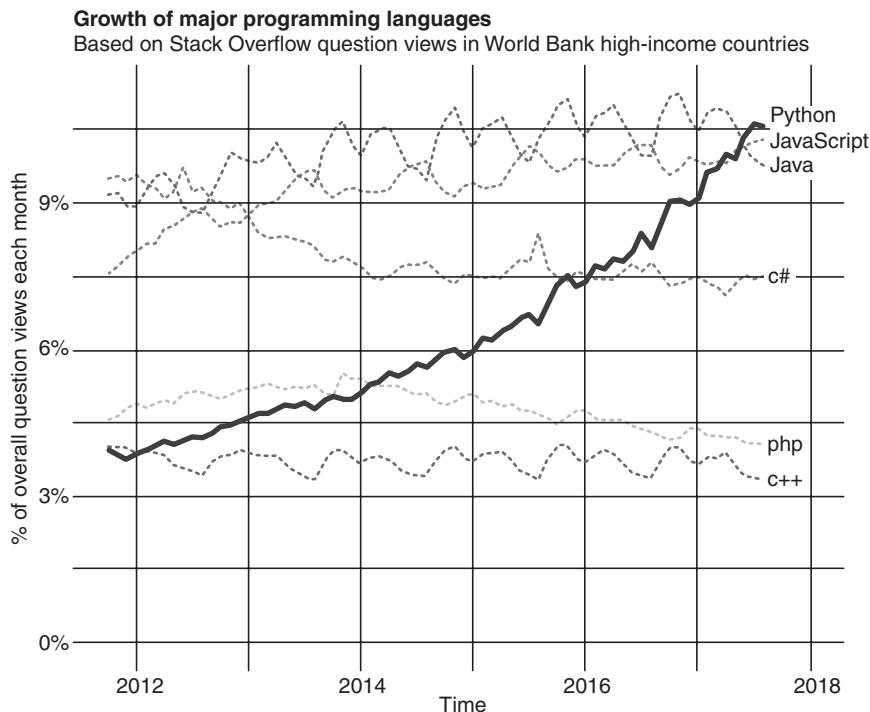


FIGURE 1.3 Overall question views for Python.

Source: <https://www.forbes.com/sites/jeffkauflin/2017/05/12/the-five-most-in-demand-coding-languages/#211777b2b3f5>

Python has an amazing ecosystem and is excellent for developing prototypes quickly. It has a comprehensive set of core libraries for data analysis and visualization. Python, unlike R, is not built only for data analysis, it is a general-purpose language. Python can be used to build web applications, enterprise applications, and is easier to integrate with existing systems in an enterprise for data collection and preparation.

Data science projects need extraction of data from various sources, data cleaning, data imputation beside model building, validation, and making predictions. Enterprises typically want to build an End-to-End integrated systems and Python is a powerful platform to build these systems.

Data analysis is mostly an iterative process, where lots of exploration needs to be done in an ad-hoc manner. Python being an interpreted language provides an interactive interface for accomplishing this.

Python's strong community continuously evolves its data science libraries and keeps it cutting edge. It has libraries for linear algebra computations, statistical analysis, machine learning, visualization, optimization, stochastic models, etc. We will discuss the different libraries in the subsequent section in detail.

Python has a shallow learning curve and it is one of the easiest languages to learn to come up to speed.

The following link provides a list of enterprises using Python for various applications ranging from web programming to complex scalable applications:

<https://www.python.org/about/success/>

An article published on forbes.com puts Python as top 5 languages with highest demand in the industry (link to the article is provided below):

<https://www.forbes.com/sites/jeffkauflin/2017/05/12/the-five-most-in-demand-coding-languages/#6e2dc575b3f5>

A search on number of job posts on various languages such as Python, R, Java, Scala, and Julia with terms like “data science” or “machine learning” on www.indeed.com site, give the following trend results (Figure 1.4). It is very clear that Python has become the language with most demand since 2016 and it is growing very rapidly.

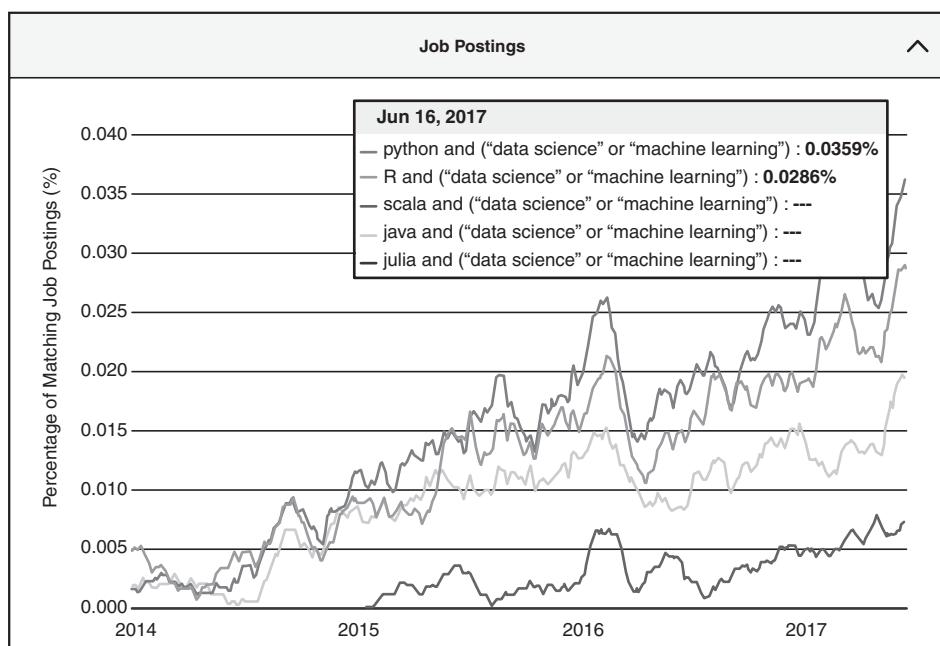


FIGURE 1.4 Trend on job postings.

Source: <http://makemeanalyst.com/most-popular-languages-for-data-science-and-analytics-2017/>

1.5 | PYTHON STACK FOR DATA SCIENCE

Python community has developed several libraries that cater to specific areas of data science applications. For example, there are libraries for statistical computations, machine learning, DataFrame operations, visualization, scientific computation using arrays and matrices, etc.

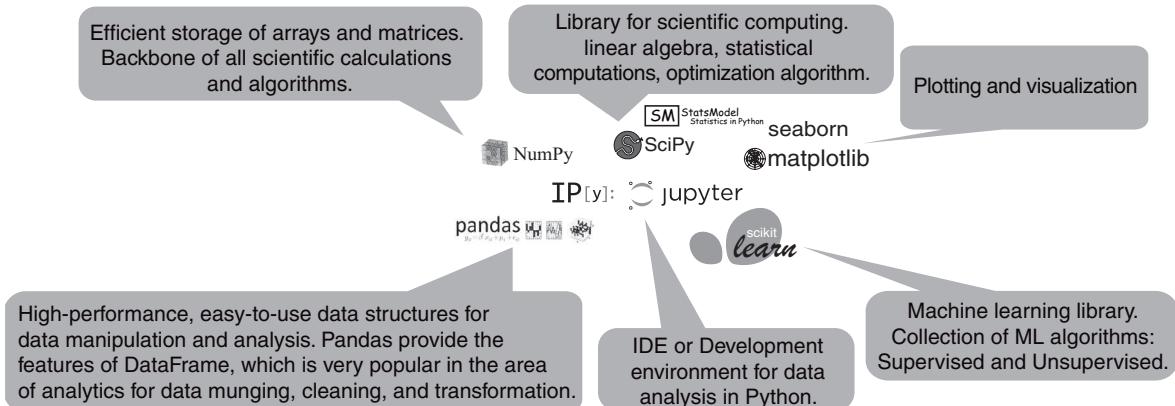


FIGURE 1.5 Python core libraries for data science applications.

Figure 1.5 shows the important Python libraries that are used for developing data science or machine learning models. Table 1.1 also provides details of these libraries and the website for referring to documentations. We will use these throughout the book.

TABLE 1.1 Core Python Libraries for Data Analysis

Areas of Application	Library	Description	Documentation Website
Statistical Computations	SciPy	SciPy contains modules for optimization and computation. It provides libraries for several statistical distributions and statistical tests.	www.scipy.org
Statistical Modelling	StatsModels	StatsModels is a Python module that provides classes and functions for various statistical analyses.	www.statsmodels.org/stable/index.html
Mathematical Computations	NumPy	NumPy is the fundamental package for scientific computing involving large arrays and matrices. It provides useful mathematical computation capabilities.	www.numpy.org
Data Structure Operations (Dataframes)	Pandas	Pandas provides high-performance, easy-to-use data structures called DataFrame for exploration and analysis. DataFrames are the key data structures that feed into most of the statistical and machine learning models.	pandas.pydata.org
Visualization	Matplotlib	It is a 2D plotting library.	matplotlib.org

(Continued)

TABLE 1.1 Continued

Areas of Application	Library	Description	Documentation Website
More elegant Visualization	Seaborn	According to seaborn.pydata.org , Seaborn is a Python visualization library based on matplotlib. It provides a high-level interface for drawing attractive statistical graphics.	seaborn.pydata.org
Machine Learning Algorithm	Scikit-learn (aka sklearn)	Scikit-learn provides a range of supervised and unsupervised learning algorithms.	scikit-learn.org
IDE (Integrated Development Environment)	Jupyter Notebook	According to jupyter.org , the Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations, and explanatory text.	jupyter.org

1.6 | GETTING STARTED WITH ANACONDA PLATFORM

We recommend using Anaconda platform for data science. The Anaconda distribution simplifies the installation process by including almost everything we need for working on data science tasks. It contains the core Python language, as well as all the essential libraries including NumPy, Pandas, SciPy, Matplotlib, sklearn, and Jupyter notebook. It has distributions for all Operating Systems (OS) environments (e.g. Windows, MAC, and Linux). Again, we recommend using Python 3.5+ environment for Anaconda. All the codes in the book are written using Anaconda 5.0 for Python 3.5+.

Follow the steps below for installation:

Step 1: Go to Anaconda Site

Go to <https://www.anaconda.com/distribution/> using your browser window.

Step 2: Download Anaconda Installer for your Environment

Select your OS environment and choose Python 3.7 version to download the installation files as shown in Figure 1.6.

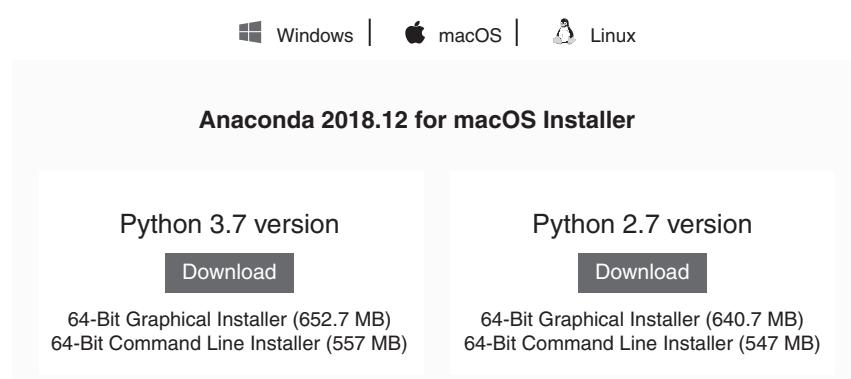


FIGURE 1.6 Anaconda distribution site for downloading the installer.

Source: www.anaconda.com

Step 3: Install Anaconda

Double click on the downloaded file and follow the on-screen installation instructions, leaving options as set by default. This will take a while and complete the installation process.

Step 4: Start Jupyter Notebook

Open the command terminal window as per your OS environment and type the following command, as shown in Figure 1.7.

```
jupyter notebook --ip=*
```



```
Manaranjan-MacBook-Pro:~ manaranjan$ jupyter notebook --ip=*
[I 10:20:14.928 NotebookApp] [nb_conda_kernels] enabled, 2 kernels found
[W 10:20:15.636 NotebookApp] WARNING: The notebook server is listening on all IP addresses and not using encryption. This is not recommended.
[I 10:20:15.805 NotebookApp] [nb_anacondacloud] enabled
[I 10:20:15.811 NotebookApp] [nb_conda] enabled
[I 10:20:15.893 NotebookApp] ✓ nbpresent HTM export ENABLED
[W 10:20:15.893 NotebookApp] ✗ nbpresent PDF export DISABLED: No module named 'nbbrowserpdf'
[I 10:20:15.900 NotebookApp] Serving notebooks from local directory: /Users/manaranjan
[I 10:20:15.900 NotebookApp] 0 active kernels
[I 10:20:15.900 NotebookApp] The Jupyter Notebook is running at: http://[all ip addresses on your system]:8888/?token=ae80c575b3993c6dcf5ba8d0b
f5f577da1c019e1380e4eeef
[I 10:20:15.900 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 10:20:15.901 NotebookApp]

Copy/paste this URL into your browser when you connect for the first time,
to login with a token:
http://localhost:8888/?token=ae80c575b3993c6dcf5ba8d0b5f577da1c019e1380e4eeef
[I 10:20:16.320 NotebookApp] Accepting one-time-token-authenticated connection from ::1
[W 10:20:16.537 NotebookApp] 404 GET /apple-touch-icon-precomposed.png (::1) 7.52ms referer=None
[W 10:20:16.541 NotebookApp] 404 GET /apple-touch-icon.png (::1) 1.01ms referer=None
```

FIGURE 1.7 Screenshot of starting the jupyter notebook.

This should start the Jupyter notebook and open a browser window in your default browser software as shown in Figure 1.8.

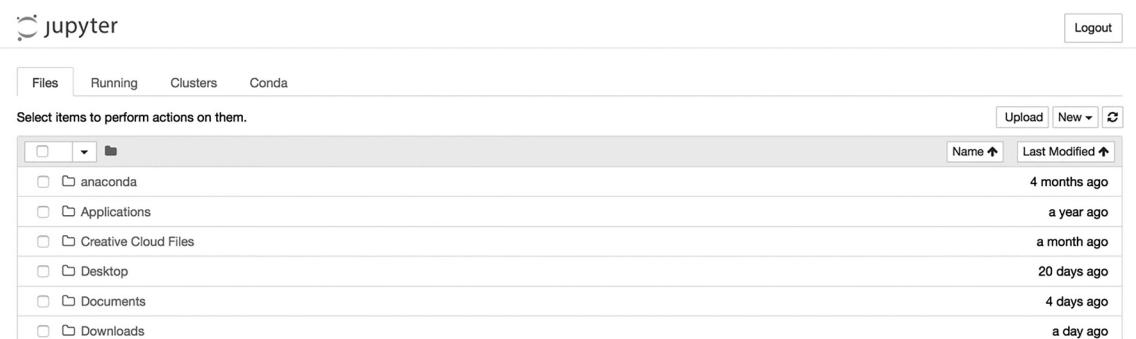


FIGURE 1.8 Screenshot of the file system explorer of Jupyter notebook open in the browser.

The reader can also start browser window using the URL highlighted below. The URL also contains the password token as shown in Figure 1.9.

```
[I 10:20:15.900 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 10:20:15.901 NotebookApp]

Copy/paste this URL into your browser when you connect for the first time,
to login with a token:
[ http://localhost:8888/?token=ae80c575b3993c6dcf5ba8d0bf5f577da1c019e1380e4eef ]
[I 10:20:16.320 NotebookApp] Accepting one-time-token-authenticated connection from ::1
[W 10:20:16.537 NotebookApp] 404 GET /apple-touch-icon-precomposed.png (::1) 7.52ms referer=None
[W 10:20:16.541 NotebookApp] 404 GET /apple-touch-icon.png (::1) 1.01ms referer=None
```

FIGURE 1.9 Screenshot of using notebook URL.

Step 5: Create a New Python Program

On the browser window, select “New” for a menu. Clicking on the “Folder” will create a directory in the current directory.

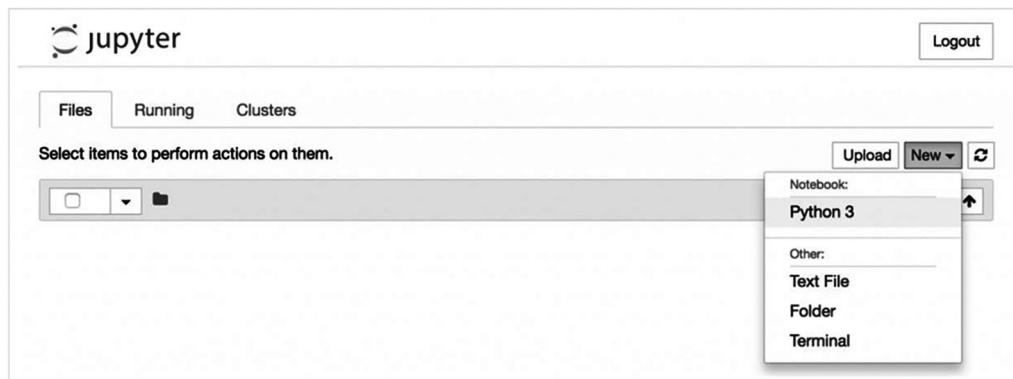


FIGURE 1.10 Screenshot of creating a new Python program.

To create a Python program, click on “Python 3”. It will open a new window, which will be the program editor for the new Python program as shown in Figure 1.10.

Step 6: Rename the Program

By default, the program name will be “Untitled”. Click on it to rename the program and name as per your requirement. For example, we have renamed it to “My First Program” as shown in Figure 1.11.

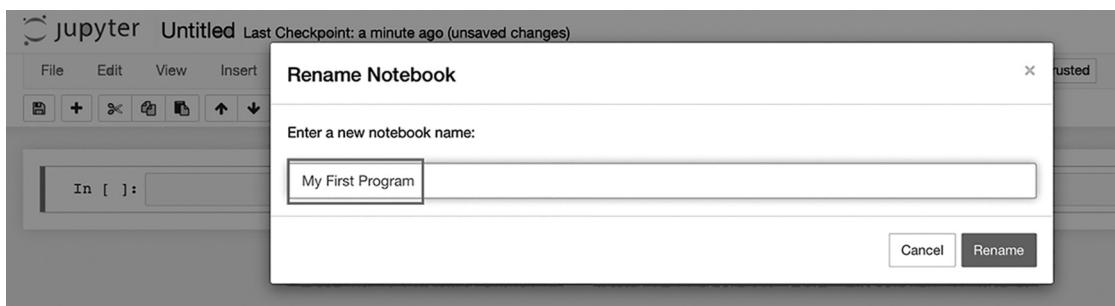


FIGURE 1.11 Screenshot of renaming the python program file.

Step 7: Write and Execute Code

Write Python code in the cell and then press SHIFT+ENTER to execute the cell as shown in Figure 1.12.

The screenshot shows a Jupyter Notebook interface titled "My First Program". The top menu bar includes File, Edit, View, Insert, Cell, Kernel, Widgets, and Help. Below the menu is a toolbar with various icons. The main area has two code cells. The first cell, labeled "In [3]:", contains the assignment statement `a = 5`. The second cell, labeled "In [4]:", contains the print statement `print(a)`, which outputs the value `5`. A message at the bottom of the notebook says "Press SHIFT + ENTER to execute the code line".

FIGURE 1.12 Screenshots of Python code and execution output.

Step 8: Basic Commands for Working with Jupyter Notebook

Click on “User Interface Tour” for a quick tour of Jupyter notebook features. Or click on “Keyboard Shortcuts” for basic editor commands as shown in Figure 1.13.

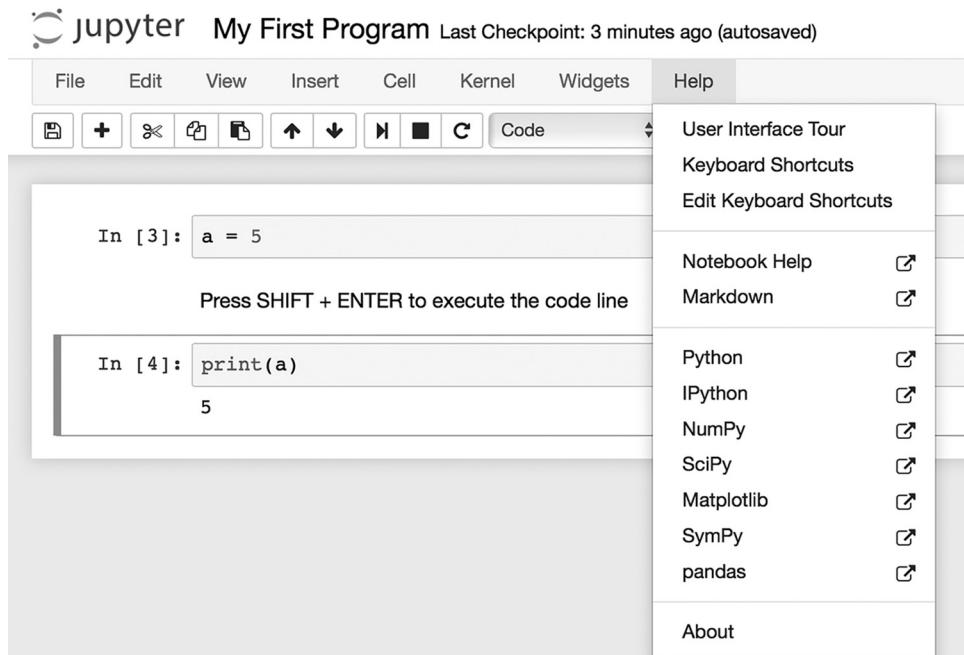


FIGURE 1.13 Screenshot of basic commands in Jupyter notebook.

The above steps should get you started with setting up the environment. Now we are all set to explore ML using Python applying real-world datasets and data science techniques in the subsequent chapters.

1.7 | INTRODUCTION TO PYTHON

In this section we will provide an overview of the Python programming language and its features. The focus will be on features that are important from data analysis and machine learning perspective. Here is a list of features that will be discussed in this section.

1. Declaring variables
2. Conditional Statements
3. Control flow statements
4. Collections
5. Functions
6. Functional Programming
7. Modules and packages

So, let us get started!

1.7.1 | Declaring Variables

In Python, a variable can be declared and then assigned a value without specifying data type. Python automatically infers the variable type from values assigned to it. Also, the variable need not be assigned a value of same type during its lifetime. A variable initialized with a value of one type (e.g., integer) can later be re-assigned as value of a different type (e.g., string). This implicit conversion can increase productivity and code reusability.

Python supports the following variable types:

1. *int* – Integer type.
2. *float* – Floating point numbers.
3. *bool* – Booleans are subtypes of integers and assigned value using literals *True* and *False*.
4. *str* – Textual data.

Below is an example of creating a set of variables and assigning values of different data types, that is, integer, float, Boolean, and string types.

```
var1 = 2
var2 = 5.0
var3 = True
var4 = "Machine Learning"
```

We print values of all the variables using *print()* method. The signature of print method is

*print(*objects, sep=' ', end='\n').*

where

1. *objects* – object to be printed. It can take variable number of objects, that is, one or more objects to be printed.
2. *sep* – objects are separated by sep. Default value: ‘ ’ (space).
3. *end* – end is printed at last. By default it is new line character.

Now print the variable values along with a string message.

```
print("Value of var1 :", var1)
print("Value of var2 :", var2)
print("Value of var3 :", var3)
print("Value of var4 :", var4)
```

```
Value of var1 : 2
Value of var2 : 5.0
Value of var3 : True
Value of var4 : Machine Learning
```

Check the data type of each variable. Python provides a method *type()*, which takes a variable as an argument and prints its data type.

```
type(var1)
```

```
int
```

```
type(var2)
```

```
float
```

```
type(var3)
```

```
bool
```

```
type(var4)
```

```
str
```

As we can see, the variables' data types are appropriately chosen based on the values assigned to them.

1.7.2 | Conditional Statements

Python supports *if-elif-else* for writing conditional statements.

The condition should be terminated by : (colon) and the code block following that must be indented. Indentation in Python is not optional. It is a syntax of Python. The conditional statement need not be enclosed with a bracket.

An example of a simple **if** condition is shown below.

```
# Checking a condition if the variable value is more than 1
if var1 > 1:
    print( "Bigger than 1" )
```

Bigger than 1

Note: A line starting with hash (#) is a comment line in Python.

The following is an example of a complex *if-elif-else* statement to check if the variable *x* is larger than *y* or smaller or same.

```
x = 10
y = 12

# if x is greater than y
if x > y:
    print ("x > y")
# if x is lesser than y
elif x < y:
    print ("x < y")
else:
    print ("x = y")
```

x < y

Ternary operators are operators that evaluate something based on a condition being true or false. It enables to test a condition in a single line instead of the multi-line *if-else*. For example, assigning *True* or *False* to variable based on condition check as below.

```
# Initialize
x = 5

# True if x is more than 10 or assign False using ternary operator
isGreater = True if x > 10 else False

print(isGreater)
```

False

1.7.3 | Generating Sequence Numbers

Sometimes it may be necessary to create a sequence of numbers. *range()* function generates a sequence of numbers. It takes the following three parameters.

1. start: Starting number of the sequence.
2. stop: Generate numbers up to, but not including, this number.
3. step: Difference between each number in the sequence. It is optional and default value is 1.

To generate a sequence of numbers from 1 to 5 use the following code:

```
# Initializing the sequence of numbers starting from 1
# and ending (not including) with 6
numbers = range(1, 6)

numbers

range(1, 6)
```

1.7.4 | Control Flow Statements

A **for loop** can be used to print the sequence of numbers by iterating through it as follows.

```
# Iterate through the collection
for i in numbers:
    print (i)
```

```
1
2
3
4
5
```

The *for* loop automatically advances the iterator through the range after every loop and completes after the iterator reaches the end. The above loop prints values from 1 to 5 as *numbers* variable contains range from 1 to 6 (not including 6).

A while loop is used to keep executing a loop until a condition is false. We can print a list of integer from 1 to 5 using *while loop* as follows:

```
# Initialize the value of i
i = 1

# check the value of i to check if the loop will be continued or not
while i < 5:

    print(i)
    # Increment the value of i.
    i = i+1

# print after the value of i
print('Done')
```

```
1
2
3
4
Done
```

In *while* loop, the state of *i* has to be managed explicitly. There is a good chance of going into an infinite loop if the state of *i* is not incremented.

1.7.5 | Functions

Functions are the most important part of a language.

1. Functions can be created using *def* keyword.
2. The function signature should contain the function name followed by the input parameters enclosed in brackets and must end with a colon (:).
3. Parameters are optional if initialized in the definition. The code block inside the method should be indented.
4. The function ends with a return statement. No return statement implies the function returns *None*, which is same as *void* return type in languages such as C, C++, or Java.

The data types of parameters and return types are inferred at runtime.

The following is an example of a function which takes two parameters and returns the addition of their values using “+” operator.

```
def addElements(a, b):  
    return a + b
```

Few examples of method invocation are shown below.

Example: Invoking the function with two integer values.

```
result = addElements(2, 3)  
result
```

5

Example: Invoking the function with two float values.

```
result = addElements(2.3, 4.5)  
result
```

6.8

Example: Invoking the function with two strings.

```
result = addElements("python", "workshop")  
result  
  
'pythonworkshop'
```

It can be observed that the data type of parameters and return types are automatically determined based on values passed to the function. If two strings are passed, they are concatenated as the + operator is also overloaded for string concatenation.

The default value for the parameters can be defined in function signatures. This makes the parameter optional.

Example: Defining the method *addElements()* with parameter *b* initialized to 4.

```
def addElements(a, b = 4):
    return a + b
```

Example: Invoking the function with only one parameter, that is, *a*.

```
addElements(2)
```

6

So, the above method assigns 2 to variable *a* and adds that to the variable *b*, whose default value is 4.

Example: Invoking the function with both the parameters.

```
addElements(2, 5)
```

7

In the last example, the default value of *b* is overridden with the value passed. Hence the sum of *a* and *b* is 7.

1.7.6 | Working with Collections

Collections are useful containers or data structures to store and manipulate list of homogeneous or heterogeneous elements. We will discuss the following collections in this section:

1. List
2. Tuple
3. Set
4. Dictionary

1.7.6.1 List

Lists are like arrays, but can contain heterogeneous items, that is, a single list can contain items of type integer, float, string, or objects. It is also not a unique set of items, that is, the values can repeat. Lists are mutable and generally initialized with a list of values specified inside square brackets or an empty list.

```
## Create an empty list
emptyList = []
```

As an example let us create a list of batsmen in Indian cricket team in the order of batting.

```
batsmen = ['Rohit', 'Dhawan', 'Kohli', 'Rahane', 'Rayudu', 'Dhoni']
```

The list index starts with 0. An item in the list can be accessed using index as follows:

```
batsmen[0]
```

```
'Rohit'
```

A slice of the list can be obtained using an index range separated by a colon (:). A range [0:2] means starting with index 0 until index 2, but not including 2.

```
## Slicing a list  
batsmen[0:2]
```

```
['Rohit', 'Dhawan']
```

To find the last batsman, an index value of -1 can be used.

```
## Accessing the last element  
batsmen[-1]
```

```
'Dhoni'
```

To find out number of elements in the list, the list can be passed to a function called **len()**.

```
# how many elements in the list  
len(batsmen)
```

```
6
```

Two separate lists can be concatenated into one list using + operator.

```
bowlers = ['Bumrah', 'Shami', 'Bhuvi', 'Kuldeep', 'Chahal']
```

```
all_players = batsmen + bowlers
```

```
all_players
```

```
['Rohit',  
'Dhawan',  
'Kohli',  
'Rahane',  
'Rayudu',  
'Dhoni',
```



```
'Bumrah',  
'Shami',  
'Bhuvi',  
'Kuldeep',  
'Chahal']
```

Finding if an item exists in a list or not, the *in* operator can be used. It returns True if exists, else returns False.

```
'Bumrah' in bowlers
```

True

```
'Rayudu' in bowlers
```

False

Finding the index of an item in the list.

```
all_players.index('Dhoni')
```

5

The items in a list can be arranged in reverse order by calling *reverse()* function on the list.

```
all_players.reverse()
```

```
all_players
```

```
['Chahal',  
'Kuldeep',  
'Bhuvi',  
'Shami',  
'Bumrah',  
'Dhoni',  
'Rayudu',  
'Rahane',  
'Kohli',  
'Dhawan',  
'Rohit']
```

1.7.6.2 Tuples

Tuple is also a list, but it is immutable. Once a tuple has been created it cannot be modified. For example, create a tuple which can contain the name of a cricketer and the year of his one-day international (ODI) debut.



```
odiDebut = ('Kohli', 2008)
```

```
odiDebut
```

```
('Kohli', 2008)
```

Tuple element's index also starts with 0.

```
odiDebut[0]
```

```
'Kohli'
```

It is not allowed to change the tuple elements. For example, if we try to change the year in the tuple, it will give an error.

```
tup1[1] = 2009
```

```
-----  
TypeError                                     Traceback (most recent call last)  
<ipython-input-217-9195c07b537c> in <module>()  
----> 1 tup1[1] = 2009
```

```
TypeError: 'tuple' object does not support item assignment
```

The above statement results in an error saying the tuple does not support item assignment, that is, the tuple variable is immutable.

An existing list can be converted into tuple using *tuple* type cast. We convert the list *all_players* into tuple so that it cannot be modified anymore.

```
players = tuple(all_players)
```

```
players
```

```
('Chahal',  
'Kuldeep',  
'Bhuvi',  
'Shami',  
'Bumrah',  
'Dhoni',  
'Rayudu',  
'Rahane',  
'Kohli',  
'Dhawan',  
'Rohit')
```

1.7.6.3 Set

A set is a collection of unique elements, that is, the values cannot repeat. A set can be initialized with a list of items enclosed with curly brackets.

```
setOfNumbers = { 6, 1, 1, 2, 4, 5 }
```

The set automatically removes duplicates and contains only unique list of numbers.

```
setOfNumbers
```

```
{1, 2, 4, 5, 6}
```

The set supports operations such as union, intersection, and difference.

To understand these operations, let us create two sets with list of batsmen who played for India in 2011 and 2015 world cup teams.

```
wc2011 = {"Dhoni", "Sehwag", "Tendulkar", "Gambhir",
           "Kohli", "Raina", "Yuvraj", "Yusuf"}
wc2015 = {"Dhoni", "Dhawan", "Rohit", "Rahane",
           "Kohli", "Raina", "Rayudu", "Jadeja"}
```

To find the list of all batsmen who played in **either** 2011 or 2015 world cup, we can take union of the above two sets.

```
wc2011.union(wc2015)
```

```
{'Dhawan',
 'Dhoni',
 'Gambhir',
 'Jadeja',
 'Kohli',
 'Rahane',
 'Raina',
 'Rayudu',
 'Rohit',
 'Sehwag',
 'Tendulkar',
 'Yusuf',
 'Yuvraj'}
```

To find the list of all batsmen who played for **both** 2011 and 2015 world cup, we can take intersection of these two sets wc2011 and wc2015.

```
wc2011.intersection(wc2015)
```

```
{'Dhoni', 'Kohli', 'Raina'}
```

If we need to find the new batsmen who were not part of 2011 world cup and played in 2015 world cup, we take difference of wc2015 from wc2011.

```
wc2015.difference( wc2011 )
```

```
{'Dhawan', 'Jadeja', 'Rahane', 'Rayudu', 'Rohit'}
```

1.7.6.4 Dictionary

Dictionary is a list of key and value pairs. All the keys in a dictionary are unique. For example, a dictionary that contains the ICC ODI World Cup winner from 1975 till 2011, where the key is year of tournament and the value is the name of the winning country.

```
wcWinners = {1975: "West Indies",
              1979: "West Indies",
              1983: "India",
              1987: "Australia",
              1991: "Pakistan",
              1996: "Sri Lanka",
              1999: "Australia",
              2003: "Australia",
              2007: "Australia",
              2011: "India"}
```

The value of a specific dictionary element can be accessed by key. For example, to find the winning country in a specific year.

```
wcWinners[1983]
```

```
'India'
```

For a list of all winning countries use the following code:

```
wcWinners.values()
```

```
dict_values(['Australia', 'Australia', 'Australia',
            'Pakistan', 'West Indies', 'India', 'West Indies',
            'Sri Lanka', 'Australia', 'Australia', 'India'])
```

The above list had repeated names of certain countries as they have won multiple times. To find unique list of countries, the above list can be converted to a set.

```
set(wcWinners.values())
{'Australia', 'India', 'Pakistan', 'Sri Lanka', 'West Indies'}
```

To add a new key-value pair to the dictionary, use the following code:

```
wcWinners[2015] = 'Australia'
```

```
wcWinners
```

```
{1975: 'West Indies',
1979: 'West Indies',
1983: 'India',
1987: 'Australia',
1991: 'Pakistan',
1996: 'Sri Lanka',
1999: 'Australia',
2003: 'Australia',
2007: 'Australia',
2011: 'India',
2015: 'Australia'}
```

1.7.7 | Dealing with Strings

A string in Python can be initialized with single or double quotes.

```
string0 = 'python'
string1 = "machine learning"
```

If multiline, the string can be initialized with triple quotes as below.

```
string2 = """This is a multiline string"""
```

To convert a string to upper or lower case use the following codes:

```
# Converting to upper case string0.upper()
# Similarly string.lower() can be used to convert to lower case.
# string0.lower()
```

```
'PYTHON'
```

For splitting the string into a list of words or tokens separated by space use the following:

```
tokens = string1.split(' ')
tokens

['machine', 'learning']
```

1.7.8 | Functional Programming

Functional programming supports functions being passed as parameters to another function like variables. This allows to create higher order functions. One core benefit of functional programming in data analysis is applying transformations or filters to a set of records or columns more efficiently than using plain looping.

1.7.8.1 Example 1: Map

Let us say we have a list named *intList* which contains integers as defined below.

```
intList = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

We want to create another list named *squareList*, which contains the squared value of all elements in *intList*. Typical approach to accomplish this is to write a *for loop* as below.

```
# Create an empty list.
squareList = []

# Loop through the intList, square every item and append to result
# list squareList.
for x in intList:
    squareList.append(pow( x, 2 ) )

print(squareList)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

The above code is quite verbose and not efficient. The loop transforms items in sequential fashion and has no scope of parallel processing. Using functional programming approach, this can be written more efficiently as described below in steps.

Step 1: Define a function *square_me()* that takes an integer and returns the square value of it.

```
def square_me( x ):
    return x * x
```

Step 2: The function *square_me* and the list of integers can be passed to a higher order function *map()*. *map()* iterates through the list and transforms each element using the function.

```
squareList = map(square_me, intList)
```

Now print the result as a list.

```
list(squareList)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

The square function *square_me()* we used is just one line of code and can actually be written as an anonymous function. Anonymous function is a function without a name and is defined using *lambda* keyword.

Write the above map using anonymous function.

```
squareList = map(lambda x: x*x, intList)
list(squareList)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

1.7.8.2 Example 2: Filter

Similarly, filters can also be applied using functional programming. For example, in case we want to select only the even numbers from the numbers in the list *intList*. It will filter only those numbers which are divisible by 2 and can be achieved using higher order function *filter()*.

filter() takes a function as an argument, which should act like a filter and return True or False. If returns False, the element will be filtered out. To verify if an integer is even, we can use filter $x \% 2 == 0$.

```
evenInts = filter(lambda x : x % 2 == 0, intList)
```

```
list(evenInts)
```

```
[2, 4, 6, 8]
```

Functional programming is an important aspect of Python programming and will be used extensively during data analysis. During data analysis, we will deal with mostly a collection of records. So, to accomplish tasks like transformations or filters, functional programming can be very handy in the place of plain looping.

1.7.9 | Modules and Packages

In Python, a module is a file that consists of functions, classes, and variables. A set of modules under a namespace (mostly a directory) is called a package. The modules and packages can be imported to

another module using *import* statement. For example, to use mathematical functions, Python's *math* module can be imported.

```
import math

# Taking square root of a value
math.sqrt(16)
```

4.0

Using *from...import* syntax, a specific module or object (e.g., class or function) can be imported from a package. For example, to import only *sample()* function from *random* module the following import style can be used.

```
from random import sample
```

Another example is generating a random set of numbers in a range, that is, *range(0, 10)*. *random.sample()* takes the range function and number of random numbers to be generated as parameters. The code below helps to generate 3 random numbers between 0 and 10.

```
sample(range(0, 11), 3)
```

[8, 7, 2]

1.7.10 | Other Features

It may be necessary to return multiple values from a function. This can be achieved by returning a tuple. This is an important feature we will be using in the subsequent chapters.

For example, define a function that returns the mean and the median of a list of numbers generated randomly.

```
import random

randomList = random.sample(range(0, 100), 20)
randomList
```

[67, 13, 56, 19, 65, 8, 2, 28, 75, 35, 32, 16, 43, 57,
11, 81, 64, 46, 3, 6]

```
from statistics import mean, median

def getMeanAndMedian(listNum):
    return mean(listNum), median(listNum)
```

`getMeanAndMedian()` returns a tuple with two elements and is stored into two separate variables during invocation.

```
mean, median = getMeanAndMedian(randomList)
```

Print the mean and median values from the list.

```
print("Mean:", mean, "Median:", median)
```

Mean: 36.35 Median: 33.5

FURTHER READING

In this chapter, we have given you a crash course on Python language and its features. This quick overview is enough to get you started with Machine Learning using Python. For more detailed overview and deep dive into the programming language and its features, refer to the resources listed below that are dedicated to Python programming.

1. <https://www.python.org/doc/>
2. <https://docs.python.org/3/tutorial/index.html>

REFERENCES

1. Downey A B (2012). *Think Python*, O'Reilly, California, USA.
2. Kauflin J (2017). “The Five Most In-Demand Coding Languages”, *Forbes*, May 12, 2017. Available at <https://www.forbes.com/sites/jeffkauflin/2017/05/12/the-five-most-in-demand-coding-languages/#54b7ea48b3f5>
3. MacKenzie I, Meyer C, and Noble S (2013), “How Retailers can keep up with Customers”, McKinsey & Company Insights, October 2013. Available at <http://www.mckinsey.com/industries/retail/our-insights/how-retailers-cankeep-up-with-consumers>. Accessed on 20 March 2017.
4. Robinson D (2017), “The Incredible Growth of Python”, *Stack Overflow Bloc*. Available at <https://stackoverflow.blog/2017/09/06/incredible-growth-python/>
5. U Dinesh Kumar (2017). *Business Analytics: The Science of Data-Driven Decision Making*, Wiley India Pvt. Ltd., India.



CHAPTER

2

Descriptive Analytics

LEARNING OBJECTIVES

After completing this chapter, you will be able to

- Understand the concept of descriptive analytics.
- Learn to load structured data onto DataFrame and perform exploratory data analysis.
- Learn data preparation activities such as filtering, grouping, ordering, and joining of datasets.
- Learn to handle data with missing values.
- Learn to prepare plots such as bar plot, histogram, distribution plot, box plot, scatter plot, pair plot, and heat maps to find insights.

2.1 | WORKING WITH DATAFRAMES IN PYTHON

Descriptive analytics is a component of analytics and is the science of describing the past data; it thus captures “what happened” in a given context. The primary objective of descriptive analytics is comprehension of data using data summarization, basic statistical measures and visualization. Data visualization is an integral component of business intelligence (BI). Visuals are used as part of dashboards that are generated by companies for understanding the performance of the company using various key performance indicators.

Data scientists deal with structured data in most of their data analysis activities and are very familiar with the concept of structured query language (SQL) table. SQL tables represent data in terms of rows and columns and make it convenient to explore and apply transformations. The similar structure of presenting data is supported in Python through DataFrames, which can be imagined as in-memory SQL tables, that is data in tabular format. DataFrames are widely used and very popular in the world of R and are inherited into Python by **Pandas** library.

A DataFrame is very efficient two-dimensional data structure as shown in Figure 2.1. It is flat in structure and is arranged in rows and columns. Rows and columns can be indexed or named.

Pandas library support methods to explore, analyze, and prepare data. It can be used for performing activities such as load, filter, sort, group, join datasets and also for dealing with missing data. To demonstrate usage of DataFrame, we will use IPL dataset (described in the next section) to load data into DataFrame and perform descriptive analytics.

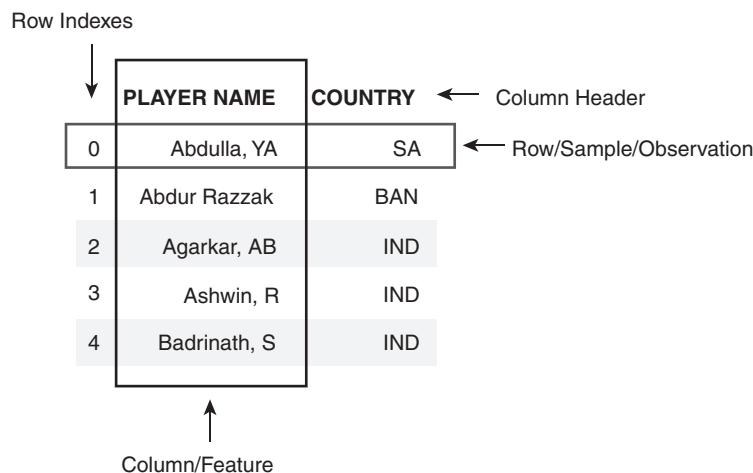


FIGURE 2.1 Structure of a DataFrame.

2.1.1 | IPL Dataset Description using DataFrame in Python

The Indian Premier League (IPL) is a professional league for Twenty20 (T20) cricket championships that was started in 2008 in India. It was initiated by the Board of Control for Cricket in India (BCCI) with eight franchises comprising players from across the world. The first IPL auction was held in 2008 for ownership of the teams for 10 years, with a base price of USD 50 million. The franchises acquire players through an English auction that is conducted every year. However, there are several rules imposed by the IPL. For example, there is a maximum cap on the money a franchise can spend on buying players.

The performance of the players could be measured through several metrics. Although the IPL follows the Twenty20 format of the game, it is possible that the performance of the players in the other formats of the game such as Test and One-Day matches can influence player pricing. A few players have excellent records in Test matches, but their records in Twenty20 matches are not very impressive. The dataset consists of the performance of 130 players measured through various performance metrics such as batting strike rate, economy rate and so on in the year 2013. The list of features is provided in Table 2.1.

TABLE 2.1 IPL auction price data description

Data Code	Data Type	Description
AGE	Categorical	Age of the player at the time of auction classified into 3 categories. Category 1 (L25) means the player is less than 25 years old, 2 means that age is between 25 and 35 years (B25–35) and category 3 means that the age is more than 35 (A35).
RUNS-S	Continuous	Number of runs scored by a player
RUNS-C	Continuous	Number of runs conceded by a player

(Continued)

TABLE 2.1 (Continued)

Data Code	Data Type	Description
HS	Continuous	Highest score by the batsman in IPL
AVE-B	Continuous	Average runs scored by the batsman in IPL
AVE-BL	Continuous	Bowling average (Number of runs conceded / number of wickets taken) in IPL
SR-B	Continuous	Batting strike rate (ratio of the number of runs scored to the number of balls faced) in IPL
SR-BL	Continuous	Bowling strike rate (ratio of the number of balls bowled to the number of wickets taken) in IPL
SIXERS	Continuous	Number of six runs scored by a player in IPL
WKTS	Continuous	Number of wickets taken by a player in IPL
ECON	Continuous	Economy rate of a bowler (number of runs conceded by the bowler per over) in IPL
CAPTAINCY EXP	Categorical	Captained either a T20 team or a national team
ODI-SR-B	Continuous	Batting strike rate in One-Day Internationals
ODI-SR-BL	Continuous	Bowling strike rate in One-Day Internationals
ODI-RUNS-S	Continuous	Runs scored in One-Day Internationals
ODI-WKTS	Continuous	Wickets taken in One-Day Internationals
T-RUNS-S	Continuous	Runs scored in Test matches
T-WKTS	Continuous	Wickets taken in Test matches
PLAYER-SKILL	Categorical	Player's primary skill (batsman, bowler, or all-rounder)
COUNTRY	Categorical	Country of origin of the player (AUS: Australia; IND: India; PAK: Pakistan; SA: South Africa; SL: Sri Lanka; NZ: New Zealand; WI: West Indies; OTH: Other countries)
YEAR-A	Categorical	Year of Auction in IPL
IPL TEAM	Categorical	Team(s) for which the player had played in the IPL (CSK: Chennai Super Kings, DC: Deccan Chargers, DD: Delhi Daredevils, KXl: Kings XI Punjab, KKR: Kolkata Knight Riders; MI: Mumbai Indians; PWI: Pune Warriors India; RR: Rajasthan Royals; RCB: Royal Challengers Bangalore). A + sign was used to indicate that the player had played for more than one team. For example, CSK+ would mean that the player had played for CSK as well as for one or more other teams.

2.1.2 | Loading Dataset into Pandas DataFrame

Pandas is an open-source, Berkeley Software Distribution (BSD)-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language (source: <https://pandas.pydata.org/>). Pandas is very popular and most widely used library for data exploration and preparation. We will be using Pandas DataFrame throughout this book.

To use the Pandas library, we need to import *pandas* module into the environment using the *import* keyword. After importing, required *pandas* methods can be invoked using the format ***pandas.<method name>***.

In Python, *longer library names* can be avoided by assigning an alias name while importing. For example, in the code below, *pandas* is imported as alias *pd*. After the execution of this code, the required methods can be invoked using ***pd.<method name>***.

```
import pandas as pd
```

Pandas library has provided different methods for loading datasets with many different formats onto DataFrames. For example:

1. `read_csv` to read comma separated values.
 2. `read_json` to read data with json format.
 3. `read_fwf` to read data with fixed width format.
 4. `read_excel` to read excel files.
 5. `read_table` to read database tables.

To find out all available methods, we can type `pd.read_` and then press *TAB* in Jupyter notebook cell. It will list all available *read* methods. To find out what parameters a method takes and what data format it supports, we can type method name followed by a question mark (?) and press *SHIFT + ENTER*. For example, the command

pd.read_csv? (Press SHIFT + ENTER)

pops out detailed documentation at the bottom of the Jupyter notebook page as shown in Figure 2.2.

```
Signature: pd.read_csv(filepath_or_buffer, sep=',', delimiter=None, header='infer', names=None, index_col=None, usecols=None, squeeze=False, prefix=None, mangle_dupe_cols=True, dtype=None, engine=None, converters=None, true_values=None, false_values=None, skipinitialspace=False, skiprows=None, nrows=None, na_values=None, keep_default_na=True, na_filter=True, verbose=False, skip_blank_lines=True, parse_dates=False, infer_datetime_format=False, keep_date_col=False, date_parser=None, dayfirst=False, iterator=False, chunksize=None, compression='infer', thousands=None, decimal=',', lineterminator=None, quotechar='"', quoting=0, escapechar=None, comment=None, encoding=None, dialect=None, tupleize_cols=None, error_bad_lines=True, warn_bad_lines=True, skipfooter=0, skip_footer=0, doublequote=True, delimeter_whitespace=False, as_recarray=None, compact_ints=None, use_unsigned=None, low_memory=True, buffer_lines=None, memory_map=False, float_precision=None)
Docstring:
Read CSV (comma-separated) file into DataFrame
```

FIGURE 2.2 Sample documentation for Pandas method `read_csv`.

As per the documentation shown in Figure 2.2 for `pd.read_csv`

1. `read_csv` takes the file name as a parameter.
 2. `read_csv` uses *comma* as separator. If any other separator, parameter *sep* to be set to appropriate character.
 3. The first line in the dataset is expected to be header. If not, the *header* parameter needs to be set to *None*.

IPL dataset is stored in comma separated values (csv) format, so we will use `pd.read_csv` method to read and load it onto a DataFrame. The dataset contains header information in the first line.

```
ipl auction df = pd.read_csv('IPL IMB381IPL2013.csv')
```

To find out the type of variable `ipl_auction_df`, we can pass the variable to `type()` function of python.

```
type(ip1 auction df)
```

```
pandas.core.frame.DataFrame
```

That is, `ipl_auction_df` is of type DataFrame. Now we can use pandas features such as selecting, filtering, aggregating, joining, slicing/dicing of data to prepare and explore the dataset.

2.1.3 | Displaying First Few Records of the DataFrame

To display the first few rows from the DataFrame, use function `head(n)`. The parameter `n` is the number of records to display.

In this example, we will only print a maximum of 7 columns as the total width exceeds the page width and the display is distorted. Setting pandas option `display.max_columns` will limit the number of columns to be printed/displayed.

```
pd.set_option('display.max_columns', 7)
```

```
ipl_auction_df.head(5)
```

	Sl. No.	Player Name	Age	...	Auction Year	Base Price	Sold Price
0	1	Abdulla, YA	2	...	2009	50000	50000
1	2	Abdur Razzak	2	...	2008	50000	50000
2	3	Agarkar, AB	2	...	2008	200000	350000
3	4	Ashwin, R	1	...	2011	100000	850000
4	5	Badrinath, S	2	...	2011	100000	800000

The values in the first column – 0 to 4 – are row indexes and the words in first row are the dataset header.

2.1.4 | Finding Summary of the DataFrame

To display all the column names, use `columns` attribute of the DataFrame `ipl_auction_df`.

```
list(ipl_auction_df.columns)
```

```
['SL.NO.',  
 'PLAYER NAME',  
 'AGE',  
 'COUNTRY',  
 'TEAM',  
 'PLAYING ROLE',  
 'T-RUNS',  
 'T-WKTS',  
 'ODI-RUNS-S',  
 'ODI-SR-B',  
 'ODI-WKTS',  
 'ODI-SR-BL',  
 'CAPTAINCY EXP',
```

```
'RUNS-S',
'HS',
'AVE',
'SR-B',
'SIXERS',
'RUNS-C',
'WKTS',
'AVE-BL',
'ECON',
'SR-BL',
'AUCTION YEAR',
'BASE PRICE',
'SOLD PRICE']
```

The other way to print a DataFrame with a large number of columns (features) is to transpose the DataFrame and display the columns as rows and rows as columns. The row indexes will be shown as column headers and column names will be shown as row indexes.

```
ipl_auction_df.head(5).transpose()
```

	0	1	2	3	4
SI. NO.	1	2	3	4	5
PLAYER NAME	Abdulla, YA	Abdur Razzak	Agarkar, AB	Ashwin, R	Badrinath, S
AGE	2	2	2	1	2
COUNTRY	SA	BAN	IND	IND	IND
TEAM	KXIP	RCB	KKR	CSK	CSK
PLAYING ROLE	Allrounder	Bowler	Bowler	Bowler	Batsman
T-RUNS	0	214	571	284	63
T-WKTS	0	18	58	31	0
ODI-RUNS-S	0	657	1269	241	79
ODI-SR-B	0	71.41	80.62	84.56	45.93
ODI-WKTS	0	185	288	51	0
ODI-SR-BL	0	37.6	32.9	36.8	0
CAPTAINCY EXP	0	0	0	0	0
RUNS-S	0	0	167	58	1317
HS	0	0	39	11	71
AVE	0	0	18.56	5.8	32.93
SR-B	0	0	121.01	76.32	120.71

(Continued)

	0	1	2	3	4
SIXERS	0	0	5	0	28
RUNS-C	307	29	1059	1125	0
WKTS	15	0	29	49	0
AVE-BL	20.47	0	36.52	22.96	0
ECON	8.9	14.5	8.81	6.23	0
SR-BL	13.93	0	24.9	22.14	0
AUCTION YEAR	2009	2008	2008	2011	2011
BASE PRICE	50000	50000	200000	100000	100000
SOLD PRICE	50000	50000	350000	850000	800000

The dimension or size of the DataFrame can be retrieved through the *shape* attribute of the DataFrame, which returns a tuple. The first value of the tuple is the number of rows and the second value is the number of columns.

```
ipl_auction_df.shape
```

```
(130, 26)
```

In this case, IPL dataset contains 130 records and 26 columns.

More detailed summary about the dataset such as the number of records, columns names, number of actual populated values in each column, datatypes of columns and total computer memory consumed by the DataFrame can be retrieved using *info()* method of the DataFrame.

```
ipl_auction_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 130 entries, 0 to 129
Data columns (total 26 columns):
Sl.NO.          130    non-null   int64
PLAYER NAME     130    non-null   object
AGE             130    non-null   int64
COUNTRY         130    non-null   Object
TEAM            130    non-null   Object
PLAYING ROLE   130    non-null   Object
T-RUNS          130    non-null   int64
T-WKTS          130    non-null   int64
ODI-RUNS-S     130    non-null   int64
ODI-SR-B       130    non-null   float64
ODI-WKTS        130    non-null   int64
ODI-SR-BL      130    non-null   float64
CAPTAINCY EXP  130    non-null   int64
RUNS-S          130    non-null   int64
```

```

HS           130    non-null   int64
AVE          130    non-null   float64
SR-B         130    non-null   float64
SIXERS       130    non-null   int64
RUNS-C       130    non-null   int64
WKTS         130    non-null   int64
AVE-BL        130    non-null   float64
ECON          130    non-null   float64
SR-BL         130    non-null   float64
AUCTION YEAR 130    non-null   int64
BASE PRICE    130    non-null   int64
SOLD PRICE    130    non-null   int64
dtypes: float64(7), int64(15), object(4)
memory usage: 26.5+ KB

```

It shows a value of 130 against every column, which indicates none of the columns (features) has any missing values. Pandas automatically infer the data type of the columns by analyzing the values of each column. If the type cannot be inferred or contains texts or literals, the column is inferred as an *object*. Continuous variables are typically inferred as either *int64* or *float64*, whereas categorical variables (strings or literals) are inferred as objects. (The variable types are discussed in detail in Chapter 3.)

As shown by the output of *info()* method, in this dataset pandas found 7 columns to be of *float* type, 15 columns to be of *integer* type and remaining 4 columns as *object* type. The DataFrame consumes a total of 26.5 KB memory.

2.1.5 | Slicing and Indexing of DataFrame

Sometimes only a subset of rows and columns are analyzed from the complete dataset. To select few rows and columns, the DataFrame can be accessed or sliced by indexes or names. The row and column indexes always start with value 0.

Assume that we are interested in displaying the first 5 rows of the DataFrame. The index range takes two values separated by a colon. For example, [0:5] means start with row with index 0 and end with row with index 5, but not including 5. [0:5] is same as [:5]. By default, the indexing always starts with 0.

```
ipl_auction_df[0:5]
```

Sl. No.	Player Name	Age	...	Auction Year	Base Price	Sold Price
0	1	Abdulla, YA	2	...	2009	50000
1	2	Abdur Razzak	2	...	2008	50000
2	3	Agarkar, AB	2	...	2008	200000
3	4	Ashwin, R	1	...	2011	100000
4	5	Badrinath, S	2	...	2011	800000

Negative indexing is an excellent feature in Python and can be used to select records from the bottom of the DataFrame. For example, [-5:] can be used to select the last five records.

```
ipl_auction_df[-5:]
```

Sl. No.	Player Name	Age	...	Auction Year	Base Price	Sold Price
125	126	Yadav, AS	2	...	2010	50000
126	127	Younis Khan	2	...	2008	225000
127	128	Yuvraj Singh	2	...	2011	400000
128	129	Zaheer Khan	2	...	2008	200000
129	130	Zoysa, DNT	2	...	2008	100000

Specific columns of a DataFrame can also be selected or sliced by column names. For example, to select only the player names of the first five records, we can use the following code:

```
ipl_auction_df['PLAYER NAME'][0:5]
```

```
0      Abdulla, YA
1      Abdur Razzak
2      Agarkar, AB
3      Ashwin, R
4      Badrinath, S
Name: PLAYER NAME, dtype: object
```

To select two columns, for example, player name and the country name of the first five records, pass a list of column names to the DataFrame, as shown below:

```
ipl_auction_df[['PLAYER NAME', 'COUNTRY']][0:5]
```

	Player Name	Country
0	Abdulla, YA	SA
1	Abdur Razzak	BAN
2	Agarkar, AB	IND
3	Ashwin, R	IND
4	Badrinath, S	IND

Specific rows and columns can also be selected using row and column indexes. For example, to select first five records starting from row index 4 and columns ranging from column index 1 (second column) to column index 4, pass as below. [4:9, 1:4] takes the row index ranges first and column ranges as second parameter. It should be passed to *iloc method of DataFrame*.

```
ipl_auction_df.iloc[4:9, 1:4]
```

	Player Name	Age	Country
4	Badrinath, S	2	IND
5	Bailey, GJ	2	AUS
6	Balaji, L	2	IND
7	Bollinger, DE	2	AUS
8	Botha, J	2	SA

In the subsequent sections, we will discuss many built-in functions of pandas to explore the dataset further. But rather than exploring each function one by one, to make our learning interesting, we will seek specific insights from the dataset, and figure out how to accomplish that using pandas.

2.1.6 | Value Counts and Cross Tabulations

value_counts() provides the occurrences of each unique value in a column. For example, we would like to know how many players from different countries have played in the IPL, then the method *value_counts()* can be used. It should primarily be used for categorical variables.

```
ipl_auction_df.COUNTRY.value_counts()
IND      53
AUS     22
SA      16
SL      12
PAK      9
NZ       7
WI       6
ENG      3
ZIM      1
BAN      1
Name: COUNTRY, dtype: int64
```

As expected, most players auctioned are from India, followed by Australia and then South Africa.

Passing parameter *normalize=True* to the *value_counts()* will calculate the percentage of occurrences of each unique value.

```
ipl_auction_df.COUNTRY.value_counts(normalize=True) *100
IND      40.769231
AUS     16.923077
SA      12.307692
SL      9.230769
```

```
PAK      6.923077
NZ       5.384615
WI       4.615385
ENG      2.307692
ZIM      0.769231
BAN      0.769231
Name: COUNTRY, dtype: float64
```

Cross-tabulation features will help find occurrences for the combination of values for two columns. For example, cross tabulation between *PLAYING ROLE* and *AGE* will give number of players in each age category for each playing role. Ages of the players at the time of auction are classified into three categories. Category 1 means the player is less than 25 years old, category 2 means that the age is between 25 and 35 years, and category 3 means that the age is more than 35. To find out such occurrences across the combination of two categories, we can use *crosstab()* method as shown below:

```
pd.crosstab( ipl_auction_df['AGE'], ipl_auction_df['PLAYING ROLE'] )
```

		Allrounder	Batsman	Bowler	W. Keeper
		Playing Role	Age		
Age	1	4	5	7	0
	2	25	21	29	11
	3	6	13	8	1

Most of the players auctioned are from the age category 2. In category 1, there are more bowlers than other playing roles and in category 3, there are more batsman than other playing roles.

2.1.7 | Sorting DataFrame by Column Values

sort_values() takes the column names, based on which the records need to be sorted. By default, sorting is done in ascending order. The following line of code selects two columns *PLAYER NAME* and *SOLD PRICE* from the DataFrame and then sorts the rows by *SOLD PRICE*.

```
ipl_auction_df[['PLAYER NAME', 'SOLD PRICE']].sort_values
('SOLD PRICE')[0:5]
```

	Player Name	Sold Price
73	Noffke, AA	20000
46	Kamran Khan	24000
0	Abdulla, YA	50000
1	Abdur Razzak	50000
118	Van der Merwe	50000

To sort the DataFrame records in descending order, pass *False* to *ascending* parameter.

```
ipl_auction_df[['PLAYER NAME', 'SOLD PRICE']].sort_values('SOLD PRICE', ascending = False) [0:5]
```

	Player Name	Sold Price
93	Sehwag, V	1800000
127	Yuvraj Singh	1800000
50	Kohli, V	1800000
111	Tendulkar, SR	1800000
113	Tiware, SS	1600000

2.1.8 | Creating New Columns

We can create new columns (new features) by applying basic arithmetic operations on existing columns. For example, let us say we would like to create a new feature which is the difference between the sold price and the base price (called by new feature name premium). We will create a new column called *premium* and populate by taking difference between the values of columns *SOLD PRICE* and *BASE PRICE*.

```
ipl_auction_df['premium'] = ipl_auction_df['SOLD PRICE'] -  
                           ipl_auction_df['BASE PRICE']
```

```
ipl_auction_df[['PLAYER NAME', 'BASE PRICE', 'SOLD PRICE',  
'premium']] [0:5]
```

	Player Name	Base Price	Sold Price	Premium
0	Abdulla, YA	50000	50000	0
1	Abdur Razzak	50000	50000	0
2	Agarkar, AB	200000	350000	150000
3	Ashwin, R	100000	850000	750000
4	Badrinath, S	100000	800000	700000

To find which players got the maximum premium offering on their base price, the DataFrame can be sorted by the values of column *premium* in descending order. *sort_values()* method sorts a DataFrame by a column whose name is passed as a parameter along with order type. The default order type is ascending. For sorting in descending order, the parameter *ascending* needs to be set to *False*.

```
ipl_auction_df[['PLAYER NAME',  
                'BASE PRICE',  
                'SOLD PRICE', 'premium']].sort_values('premium',  
                                           ascending = False) [0:5]
```

	Player Name	Base Price	Sold Price	Premium
50	Kohli, V	150000	1800000	1650000
113	Tiwary, SS	100000	1600000	1500000
127	Yuvraj Singh	400000	1800000	1400000
111	Tendulkar, SR	400000	1800000	1400000
93	Sehwag, V	400000	1800000	1400000

The result shows Virat Kohli was auctioned with maximum premium on the base price set.

2.1.9 | Grouping and Aggregating

Sometimes, it may be required to group records based on column values and then apply aggregated operations such as mean, maximum, minimum, etc. For example, to find average *SOLD PRICE* for each age category, group all records by *AGE* and then apply *mean()* on *SOLD PRICE* column.

```
ipl_auction_df.groupby('AGE')[ 'SOLD PRICE' ].mean()
```

```
AGE
1    720250.000000
2    484534.883721
3    520178.571429
Name: SOLD PRICE, dtype: float64
```

The average price is highest for age category 1 and lowest for age category 2.

The above operation returns a *pd.Series* data structure. To create a DataFrame, we can call *reset_index()* as shown below on the returned data structure.

```
soldprice_by_age = ipl_auction_df.groupby('AGE')[ 'SOLD PRICE' ].
                      mean().reset_index()
print(soldprice_by_age)
```

	Age	Sold Price
0	1	720250.000000
1	2	484534.883721
2	3	520178.571429

Data can be grouped using multiple columns. For example, to find average *SOLD PRICE* for players for each *AGE* and *PLAYING ROLE* category, multiple columns can be passed to *groupby()* method as follows:

```
soldprice_by_age_role = ipl_auction_df.groupby(['AGE', 'PLAYING
ROLE'])['SOLD PRICE'].mean().reset_index()
print(soldprice_by_age_role)
```

	Age	Playing Role	Sold Price
0	1	Allrounder	587500.000
1	1	Batsman	1110000.000
2	1	Bowler	517714.286
3	2	Allrounder	449400.000
4	2	Batsman	654761.905
5	2	Bowler	397931.034
6	2	W. Keeper	467727.273
7	3	Allrounder	766666.667
8	3	Batsman	457692.308
9	3	Bowler	414375.000
10	3	W. Keeper	700000.000

2.1.10 | Joining DataFrames

We may need to combine columns from multiple DataFrames into one single DataFrame. In this case both DataFrames need to have a common column. Then the remaining column values can be retrieved from both the DataFrames and joined to create a row in the resulting DataFrame. To merge two DataFrames, pandas method `merge()` can be called from one of the DataFrames and the other DataFrame can be passed as a parameter to it. It also takes a parameter `on`, which is the common column in both the DataFrames and should be present in both the DataFrames. DataFrames can be joined based on multiple common columns. The join type can be of inner, outer, left or right joins and should be specified in the `how` parameter. For understanding different joins, please refer to the examples given at https://www.w3schools.com/sql/sql_join.asp

For example, to compare the average *SOLD PRICE* for different *AGE* categories with the different age and *PLAYING ROLE* categories, we need to merge the DataFrames `soldprice_by_age` and `soldprice_by_age_role`. The common column is *AGE* and this needs outer join.

We will join the above two DataFrames created and then compare the difference in auction prices.

```
soldprice_comparison = soldprice_by_age_role.merge(soldprice_by_age,
                                                    on = 'AGE',
                                                    how = 'outer')
```

```
soldprice_comparison
```

	Age	Playing Role	Sold Price_x	Sold Price_y
0	1	Allrounder	587500.000	720250.000
1	1	Batsman	1110000.000	720250.000
2	1	Bowler	517714.286	720250.000

	Age	Playing Role	Sold Price_x	Sold Price_y
3	2	Allrounder	449400.000	484534.884
4	2	Batsman	654761.905	484534.884
5	2	Bowler	397931.034	484534.884
6	2	W. Keeper	467727.273	484534.884
7	3	Allrounder	766666.667	520178.571
8	3	Batsman	457692.308	520178.571
9	3	Bowler	414375.000	520178.571
10	3	W. Keeper	700000.000	520178.571

Because the column name SOLD PRICE is same in both the DataFrames, it automatically renames them to _x and _y. SOLD PRICE_x comes from the left table (*soldprice_by_age_role*) and SOLD PRICE_y comes from the right table (*soldprice_by_age*).

2.1.11 | Re-Naming Columns

The existing columns of a DataFrame can be renamed using *rename()* method. For renaming multiple columns simultaneously, the method can take a dictionary as a parameter, where the keys should be existing column names and the values should be the new names to be assigned.

```
soldprice_comparison.rename( columns = { 'SOLD PRICE_x': 'SOLD_PRICE_AGE_ROLE', 'SOLD PRICE_y': 'SOLD_PRICE_AGE' }, inplace = True )
```

```
soldprice_comparison.head(5)
```

	Age	Playing Role	Sold_price_age_role	Sold_price_age
0	1	Allrounder	587500.000	720250.000
1	1	Batsman	1110000.000	720250.000
2	1	Bowler	517714.286	720250.000
3	2	Allrounder	449400.000	484534.884
4	2	Batsman	654761.905	484534.884

2.1.12 | Applying Operations to Multiple Columns

Consider as an example a situation where we would like to find whether players carry a premium if they belong to a specific AGE and PLAYING ROLE category. The premium (we will call it change) is calculated in percentage terms and calculated as follows:

$$\text{Change} = \frac{(\text{Average SOLD PRICE for all player in an AGE and PLAYING ROLE category} - \text{Average SOLD PRICE for all player in an AGE category})}{\text{Average SOLD PRICE for all player in an AGE category}}$$

To accomplish this, we need to iterate through each row in the DataFrame and then apply the above calculations to the columns. The resulting value should be added as a new column to the existing DataFrame. The function `apply()` can apply a function along any axis of the DataFrame.

```
soldprice_comparison['change'] = soldprice_comparison.apply(lambda
    rec:(rec.SOLD_PRICE_AGE_ROLE -
    rec.SOLD_PRICE_AGE) / rec.SOLD_
    PRICE_AGE, axis = 1)
```

```
soldprice_comparison
```

	Age	Playing Role	Sold_price_age_role	Sold_price_age	Change
0	1	Allrounder	587500.000	720250.000	-0.184
1	1	Batsman	1110000.000	720250.000	0.541
2	1	Bowler	517714.286	720250.000	-0.281
3	2	Allrounder	449400.000	484534.884	-0.073
4	2	Batsman	654761.905	484534.884	0.351
5	2	Bowler	397931.034	484534.884	-0.179
6	2	W. Keeper	467727.273	484534.884	-0.035
7	3	Allrounder	766666.667	520178.571	0.474
8	3	Batsman	457692.308	520178.571	-0.120
9	3	Bowler	414375.000	520178.571	-0.203
10	3	W. Keeper	700000.000	520178.571	0.346

2.1.13 | Filtering Records Based on Conditions

Assume that we would like to filter certain records such as the players who have hit more than 80 sixers in the IPL tournament. DataFrame records can be filtered using a condition as indexing mechanism. Those records for which the condition returns `True` are selected to be part of the resulting DataFrame.

```
ipl_auction_df[ipl_auction_df['SIXERS'] > 80 ][['PLAYER NAME',
'SIXERS']]
```

	Player Name	Sixers
26	Gayle, CH	129
28	Gilchrist, AC	86
82	Pathan, YK	81
88	Raina, SK	97
97	Sharma, RG	82

2.1.14 | Removing a Column or a Row from a Dataset

To remove a column or a row, we can use `drop()` method on the DataFrame. It takes a parameter `axis` to specify if a column or a row needs to be dropped.

1. To drop a column, pass the column name and axis as 1.
2. To drop a row, pass the row index and axis as 0.

We will drop the column `SL.NO.` in the IPL dataset as we do not need this column for our exploration. Most of these DataFrame operations always result in creating new DataFrames. But creating new DataFrame each time can result in over consumption of computer memory. To avoid creating new DataFrames and make changes to the existing DataFrame, there is another parameter `inplace` available, which can be set to `True`.

```
ipl_auction_df.drop('SL.NO.', inplace = True, axis = 1)
```

```
ipl_auction_df.columns
```

```
Index ([ 'PLAYER NAME', 'AGE', 'COUNTRY', 'TEAM', 'PLAYING ROLE',
        'T-RUNS', 'T-WKTS', 'ODI-RUNS-S', 'ODI-SR-B', 'ODI-WKTS',
        'ODI-SR-BL', 'CAPTAINCY EXP', 'RUNS-S', 'HS', 'AVE',
        'SR-B', 'SIXERS', 'RUNS-C', 'WKTS', 'AVE-BL', 'ECON', 'SR-BL',
        'AUCTION YEAR', 'BASE PRICE', 'SOLD PRICE', 'premium'],
       dtype = 'object')
```

2.2 | HANDLING MISSING VALUES

In real world, the datasets are not clean and may have missing values. We must know how to find and deal with these missing values. One of the strategies to deal with missing values is to remove them from the dataset. However, whenever possible, we would like to impute the data, which is a process of filling the missing values. Data imputation techniques will be discussed in the later chapter. Here we will discuss how to find missing values and remove them. We will use an example dataset called `autos-mpg.data`¹, which contains information about different cars and their characteristics such as

1. mpg – miles per gallon
2. cylinders – Number of cylinders (values between 4 and 8)
3. displacement – Engine displacement (cu. inches)
4. horsepower – Engine horsepower
5. weight – Vehicle weight (lbs.)
6. acceleration – Time to accelerate from 0 to 60 mph (sec.)
7. year – Model year (modulo 100)
8. origin – Origin of car (1. American, 2. European, 3. Japanese)
9. name – Vehicle name

¹ The dataset can be downloaded from <https://archive.ics.uci.edu/ml/datasets/auto+mpg> (<https://archive.ics.uci.edu/ml/datasets/auto+mpg>).

The dataset contains *SPACE* separated values and has no header. To read the dataset onto the DataFrame,

Use `pd.read_csv` method, Set *separator* to `\s+` and Set *header* to None

```
autos = pd.read_csv('auto-mpg.data', sep = '\s+', header = None)
autos.head(5)
```

	0	1	2	...	6	7	8
0	18.0	8	307.0	...	70	1	Chevrolet chevelle malibu
1	15.0	8	350.0	...	70	1	buick skylark 320
2	18.0	8	318.0	...	70	1	plymouth satellite
3	16.0	8	304.0	...	70	1	amc rebel sst
4	17.0	8	302.0	...	70	1	ford torino

5 rows × 9 columns

As the dataset does not have a header, the columns are not named. We can name the columns by assigning a list of names to the DataFrame header. The column names are given in the dataset description page at the link <https://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/auto-mpg.names>

1. mpg: continuous
2. cylinders: multi-valued discrete
3. displacement: continuous
4. horsepower: continuous
5. weight: continuous
6. acceleration: continuous
7. model year: multi-valued discrete
8. origin: multi-valued discrete
9. car name: string (unique for each instance)

We will create a list of the above names and assign to the DataFrame's *columns* attribute as follows:

```
autos.columns = ['mpg', 'cylinders', 'displacement',
                 'horsepower', 'weight', 'acceleration',
                 'year', 'origin', 'name']
autos.head(5)
```

	mpg	cylinders	displacement	...	year	origin	name
0	18.0	8	307.0	...	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	...	70	1	buick skylark 320
2	18.0	8	318.0	...	70	1	plymouth satellite
3	16.0	8	304.0	...	70	1	amc rebel sst
4	17.0	8	302.0	...	70	1	ford torino

5 rows × 9 columns

The schema of the DataFrame is as follows:

```
autos.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 398 entries, 0 to 397
Data columns (total 9 columns):
mpg            398 non-null float64
cylinders      398 non-null int64
displacement   398 non-null float64
horsepower     398 non-null object
weight          398 non-null float64
acceleration   398 non-null float64
year            398 non-null int64
origin          398 non-null int64
name             398 non-null object
dtypes: float64(4), int64(3), object(2)
memory usage: 28.1+ KB
```

Here the column *horsepower* has been inferred as *object*, whereas it should have been inferred as *float64*. This may be because some of the rows contain non-numeric values in the *horsepower* column. One option to deal with this is to force convert the *horsepower* column into numeric, which should convert the non-numeric values into null values. *pd.to_numeric()* can be used to convert any column with non-numeric datatype to a column with numeric datatype. It takes a parameter *errors*, which specifies how to deal with non-numeric values. The following are the possible parameter values (source: *pandas.pydata.org*):

errors : {'ignore', 'raise', 'coerce'}, default 'raise'

1. If 'raise', then invalid parsing will raise an exception.
2. If 'coerce', then invalid parsing will be set as NaN.
3. If 'ignore', then invalid parsing will return the input.

In our case, we will use *errors = 'coerce'* to convert any non-numeric values to null *NaN*.

```
autos[“horsepower”] = pd.to_numeric(autos[“horsepower”],
errors = ‘corece’) autos.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 398 entries, 0 to 397
Data columns (total 9 columns):
mpg            398 non-null float64
cylinders      398 non-null int64
displacement   398 non-null float64
horsepower     392 non-null float64
weight          398 non-null float64
acceleration   398 non-null float64
```

```

year           398 non-null   int64
origin         398 non-null   int64
name           398 non-null   object
dtypes: float64(5), int64(3), object(1)
memory usage: 28.1+ KB

```

The column *horsepower* has been converted into *float64* type. We can verify if some of the rows contain null values in *horsepower* column. This can be done by using *isnull()* method on the DataFrame column as shown in the code below.

```
autos[autos.horsepower.isnull()]
```

	mpg	cylinders	displacement	...	year	origin	name
32	25.0	4	98.0	...	71	1	ford pinto
126	21.0	6	200.0	...	74	1	ford maverick
330	40.9	4	85.0	...	80	2	renault lecar deluxe
336	23.6	4	140.0	...	80	1	ford mustang cobra
354	34.5	4	100.0	...	81	2	renault 18i
374	23.0	4	151.0	...	82	1	ame concord di

6 rows × 9 columns

There are 6 rows which contain null values. These rows can be dropped from the DataFrame using *dropna()* method. *Dropna()* method removes all rows with NaN values. In this dataset, only one feature – horse power – contains NaNs. Hence subset argument as *dropna()* method should be used.

```
autos = autos.dropna(subset = ['horsepower'])
```

We can verify if the rows with null values have been removed by applying the filtering condition again.

```
autos[autos.horsepower.isnull()]
```

mpg	cylinders	displacement	...	year	origin	name
0 rows × 9 columns						

Filtering with *isnull()* shows no rows with null values anymore. Similarly, *fillna()* can take a default value to replace the null values in a column. For example, *fillna(0)* replaces all null values in a column with value 0.

2.3 | EXPLORATION OF DATA USING VISUALIZATION

Data visualization is useful to gain insights and understand what happened in the past in a given context. It is also helpful for feature engineering. In this section we will be discussing various plots that we can draw using Python.

2.3.1 | Drawing Plots

Matplotlib is a Python 2D plotting library and most widely used library for data visualization. It provides extensive set of plotting APIs to create various plots such as scattered, bar, box, and distribution plots with custom styling and annotation. Detailed documentation for *matplotlib* can be found at <https://matplotlib.org/>. Seaborn is also a Python data visualization library based on matplotlib. It provides a high-level interface for drawing innovative and informative statistical charts (source: <https://seaborn.pydata.org/>).

Matplotlib is a library for creating 2D plots of arrays in Python. Matplotlib is written in Python and makes use of NumPy arrays. It is well integrated with pandas to read columns and create plots. *Seaborn*, which is built on top of matplotlib, is a library for making elegant charts in Python and is well integrated with pandas DataFrame.

To create graphs and plots, we need to import *matplotlib.pyplot* and *seaborn* modules. To display the plots on the Jupyter Notebook, we need to provide a directive `%matplotlib inline`. Only if the directive is provided, the plots will be displayed on the notebook.

```
import matplotlib.pyplot as plt
import seaborn as sn
%matplotlib inline
```

2.3.2 | Bar Chart

Bar chart is a frequency chart for qualitative variable (or categorical variable). Bar chart can be used to assess the most-occurring and least-occurring categories within a dataset.

To draw a bar chart, call *barplot()* of *seaborn* library. The DataFrame should be passed in the parameter *data*. To display the average sold price by each age category, pass *SOLD PRICE* as *y* parameter and *AGE* as *x* parameter. Figure 2.3 shows a bar plot created to show the average SOLD PRICE for each age category.

```
sn.barplot(x = 'AGE', y = 'SOLD PRICE', data = soldprice_by_age);
```

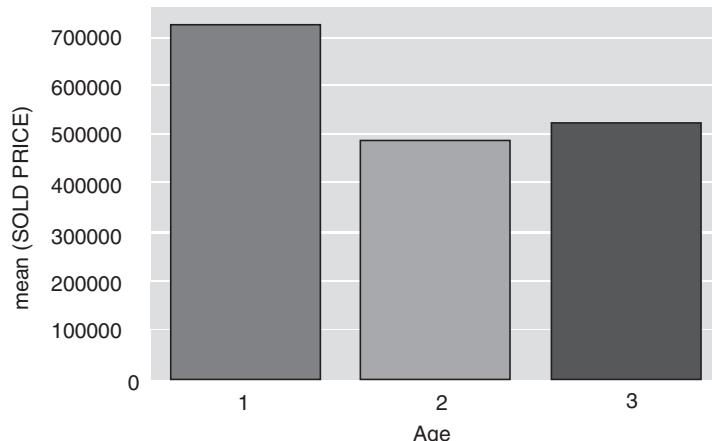


FIGURE 2.3 Bar plot for average sold price versus age.

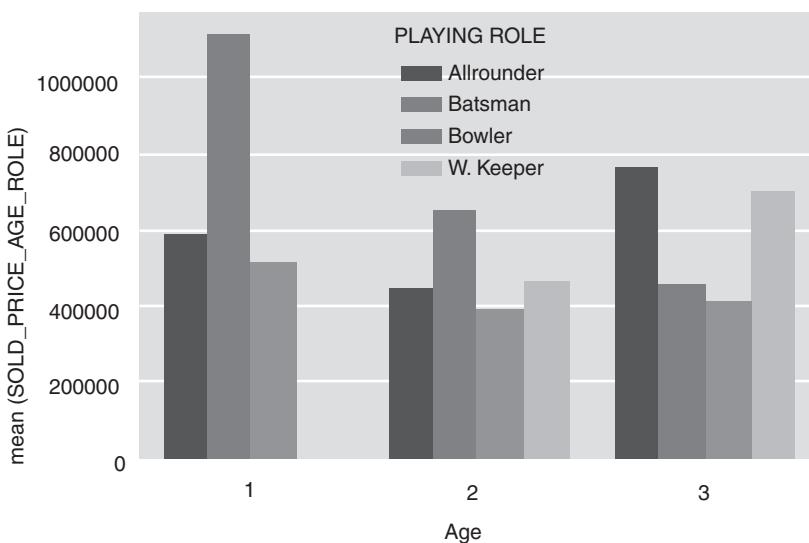


FIGURE 2.4 Bar plot for average sold price versus age by player roles. Here the first bar represents Allrounder, the second Batsman, the third Bowler, the fourth W. Keeper.

In Figure 2.3, it can be noted that the average sold price is higher for age category 1. We can also create bar charts, which are grouped by a third variable.

In the following example (Figure 2.4), average sold price is shown for each age category but grouped by a third variable, that is, playing roles. The parameter *hue* takes the third variable as parameter. In this case, we pass *PLAYING ROLE* as *hue* parameter.

```
sn.barplot(x = 'AGE', y = 'SOLD_PRICE_AGE_ROLE', hue = 'PLAYING ROLE', data = soldprice_comparison);
```

In Figure 2.4, it can be noted that in the age categories 1 and 2, batsmen are paid maximum, whereas allrounders are paid maximum in the age category 3. This could be because allrounders establish their credentials as good allrounders over a period.

2.3.3 | Histogram

A **histogram** is a plot that shows the frequency distribution of a set of continuous variable. Histogram gives an insight into the underlying distribution (e.g., normal distribution) of the variable, outliers, skewness, etc. To draw a histogram, invoke *hist()* method of matplotlib library. The following is an example of how to draw a histogram for *SOLD PRICE* and understand its distribution (Figure 2.5).

```
plt.hist( ipl_auction_df['SOLD PRICE'] );
```

The histogram shows that *SOLD PRICE* is right skewed. Most players are auctioned at low price range of 250000 and 500000, whereas few players are paid very highly, more than 1 million dollars.

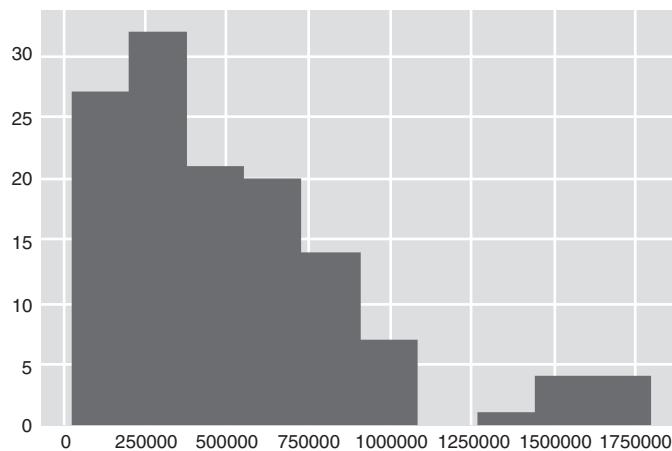


FIGURE 2.5 Histogram for SOLD PRICE.

By default, it creates 10 bins in the histogram. To create more bins, the `bins` parameter can be set in the `hist()` method as follows:

```
plt.hist( ipl_auction_df['SOLD PRICE'], bins = 20 );
```

Histogram for SOLD PRICE with 20 bins is shown in Figure 2.6.

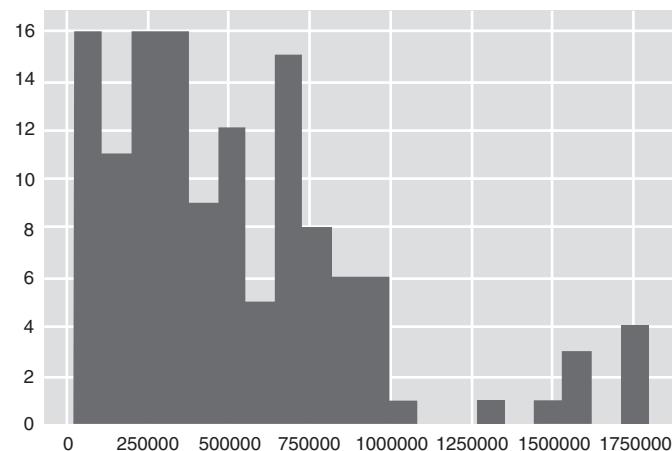


FIGURE 2.6 Histogram for SOLD PRICE with 20 bins.

2.3.4 | Distribution or Density Plot

A **distribution or density plot** depicts the distribution of data over a continuous interval. Density plot is like smoothed histogram and visualizes distribution of data over a continuous interval. So, a density plot also gives insight into what might be the distribution of the population.

To draw the distribution plot, we can use `distplot()` of seaborn library. Density plot for the outcome variable “SOLD PRICE” is shown in Figure 2.7.

```
sn.distplot(ipl_auction_df['SOLD PRICE']);
```

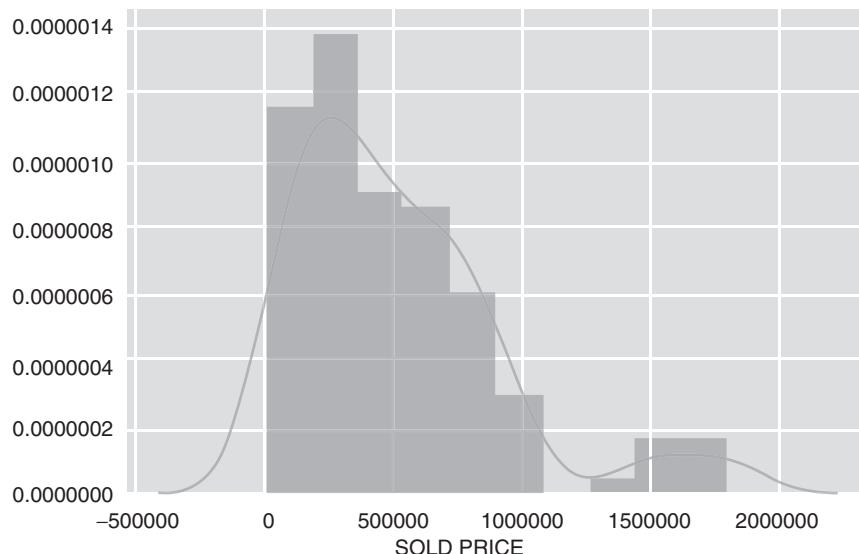


FIGURE 2.7 Distribution plot for SOLD PRICE.

2.3.5 | Box Plot

Box plot (aka Box and Whisker plot) is a graphical representation of numerical data that can be used to understand the variability of the data and the existence of outliers. Box plot is designed by identifying the following descriptive statistics:

1. Lower quartile (1st quartile), median and upper quartile (3rd quartile).
2. Lowest and highest values.
3. Inter-quartile range (IQR).

Box plot is constructed using IQR, minimum and maximum values. IQR is the distance (difference) between the 3rd quartile and 1st quartile. The length of the box is equivalent to IQR. It is possible that the data may contain values beyond $Q1 - 1.5 \text{IQR}$ and $Q3 + 1.5 \text{IQR}$. The whisker of the box plot extends till $Q1 - 1.5 \text{IQR}$ and $Q3 + 1.5 \text{IQR}$; observations beyond these two limits are potential outliers.

To draw the box plot, call `boxplot()` of seaborn library. The box plot for SOLD PRICE is shown in Figure 2.8 and indicates that there are few outliers in the data.

```
box = sn.boxplot(ipl_auction_df['SOLD PRICE']);
```

To obtain min, max, 25 percentile (1st quartile) and 75 percentile (3rd quartile) values in the boxplot, `boxplot()` method of matplotlib can be used. Let us assign the return value of `boxplot()` to a variable `box`.

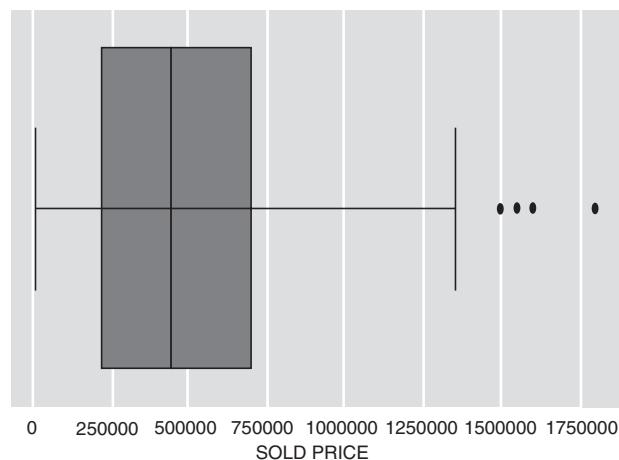


FIGURE 2.8 Box plot for SOLD PRICE.

```
box = plt.boxplot(ipl_auction_df['SOLD PRICE']);
```

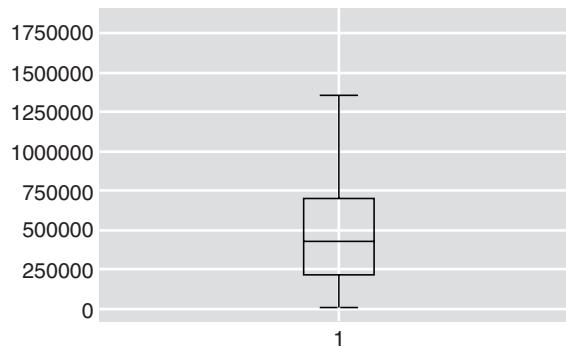


FIGURE 2.9 Box plot for SOLD PRICE with median line.

The above line of code creates a box plot as shown in Figure 2.9 and returns details of box plot in variable box. The *caps* key in box variable returns the min and max values of the distribution. Here the minimum auction price offered is 20,000 and maximum price is 13,50,000.

```
[item.get_ydata()[0] for item in box['caps']]  
[20000.0, 1350000.0]
```

The *whiskers* key in box variable returns the values of the distribution at 25 and 75 quantiles. Here the min value is 225,000 and max is 700,000.

```
[item.get_ydata()[0] for item in box['whiskers']]  
[225000.0, 700000.0]
```

So, inter-quartile range (IQR) is $700,000 - 225,000 = 475,000$.

The *medians* key in box variable returns the median value of the distribution. Here the median value is 437,500.

```
[item.get_ydata()[0] for item in box['medians']]  
[437500.0]
```

The box plot in Figure 2.8 shows that some of the players are auctioned at *SOLD PRICE*, which seem to be outliers. Let us find out the player names.

```
ipl_auction_df[ipl_auction_df['SOLD PRICE'] > 1350000.0][['PLAYER NAME', 'PLAYING ROLE', 'SOLD PRICE']]
```

	Player Name	Playing Role	Sold Price
15	Dhoni, MS	W. Keeper	1500000
23	Flintoff, A	Allrounder	1550000
50	Kohli, V	Batsman	1800000
83	Pietersen, KP	Batsman	1550000
93	Sehwag, V	Batsman	1800000
111	Tendulkar, SR	Batsman	1800000
113	Tiwarey, SS	Batsman	1600000
127	Yuvraj Singh	Batsman	1800000

The SOLD PRICE amount for the above players seems to be outlier considering what is paid to other players.

2.3.6 | Comparing Distributions

The distribution for different categories can be compared by overlapping the distributions. For example, the **SOLD PRICE** of players with and without CAPTAINCY EXP can be compared as below. Distributions can be compared by drawing distribution for each category and overlapping them in one plot.

```
sn.distplot(ipl_auction_df[ipl_auction_df['CAPTAINCY EXP'] == 1]
            ['SOLD PRICE'],
            color = 'y',
            label = 'Captaincy Experience')
sn.distplot(ipl_auction_df[ipl_auction_df['CAPTAINCY EXP'] == 0]
            ['SOLD PRICE'],
            color = 'r',
            label = 'No Captaincy Experience');
plt.legend();
```

Comparison plot for SOLD PRICE with and without captaincy experience is shown in Figure 2.10. It can be observed that players with captaincy experience seem to be paid higher.

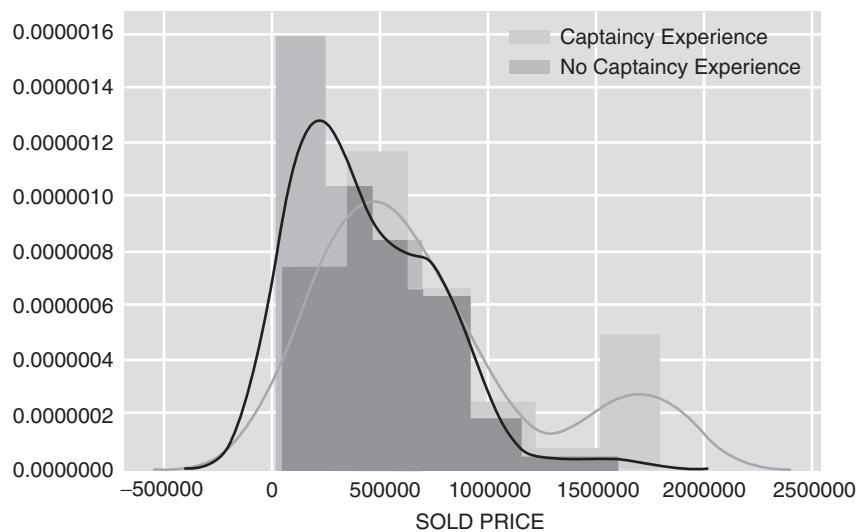


FIGURE 2.10 Distribution plots comparing SOLD PRICE with and without captaincy experience.

Similarly, the distributions can be compared using box plots. For example, if we want to visualize SOLD PRICE for each PLAYING ROLE, then pass PLAYING ROLE as x parameter and SOLD PRICE as y parameter to *boxplot()* method of seaborn.

An example of comparing the SOLD PRICE for players with different PLAYING ROLE is shown in Figure 2.11.

```
sn.boxplot(x = 'PLAYING ROLE', y = 'SOLD PRICE',
            data = ipl_auction_df);
```

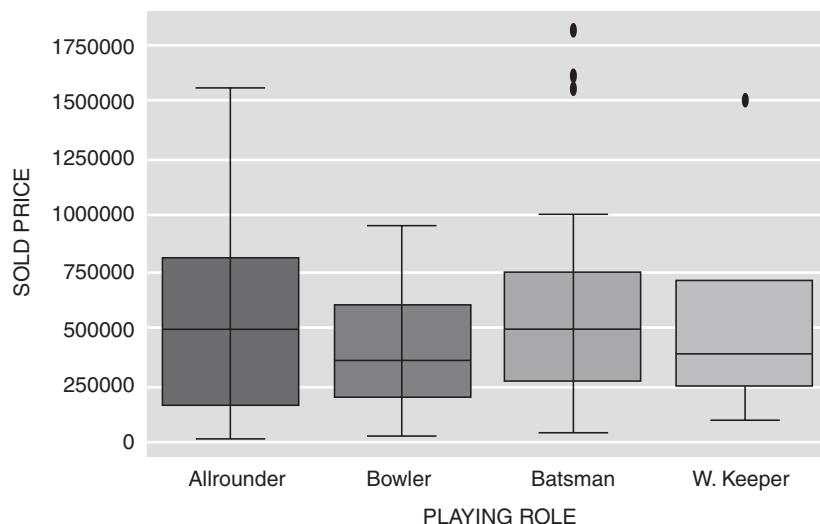


FIGURE 2.11 Box plot of SOLD PRICE for different playing roles.

Few observations from the plot in Figure 2.11 are as follows:

1. The median SOLD PRICE for allrounders and batsmen are higher than bowlers and wicket keepers.
2. Allrounders who are paid more than 1,35,0000 USD are not considered outliers. Allrounders have relatively high variance.
3. There are outliers in batsman and wicket keeper category. In Section 2.3.5, we have already found that MS DHONI is an outlier in the wicket keeper category.

2.3.7 | Scatter Plot

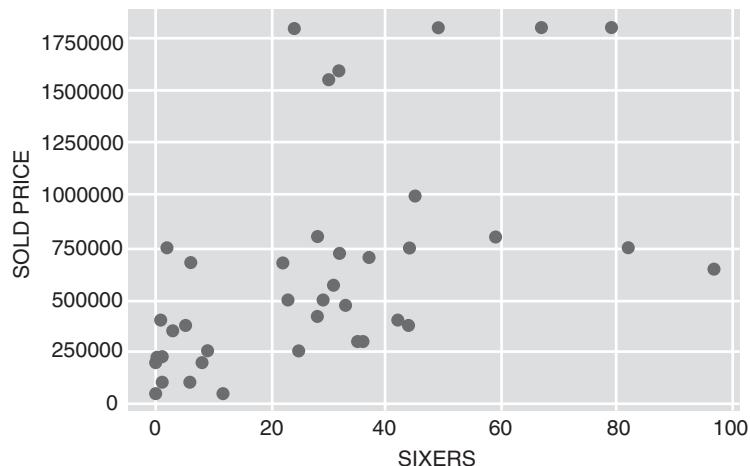
In a scatter plot, the values of two variables are plotted along two axes and resulting pattern can reveal correlation present between the variables, if any. The relationship could be linear or non-linear. A scatter plot is also useful for assessing the strength of the relationship and to find if there are any outliers in the data.

Scatter plots are used during regression model building to decide on the initial model, that is whether to include a variable in a regression model or not.

Since IPL is T20 cricket, it is believed that the number of sixers a player has hit in past would have influenced his SOLD PRICE. A scatter plot between SOLD PRICE of batsman and number of sixes the player has hit can establish this correlation. The *scatter()* method of matplotlib can be used to draw the scatter plot which takes both the variables. We will draw the scatter plot only for batsman playing role. Figure 2.12 shows the scatter plot between SIXERS and SOLD PRICE.

```
ipl_batsman_df = ipl_auction_df[ipl_auction_df['PLAYING ROLE'] ==
                                'Batsman']

plt.scatter(x = ipl_batsman_df.SIXERS,
            y = ipl_batsman_df['SOLD PRICE']);
```



```
sn.regplot( x = 'SIXERS',
            y = 'SOLD PRICE',
            data = ipl_batsman_df );
```

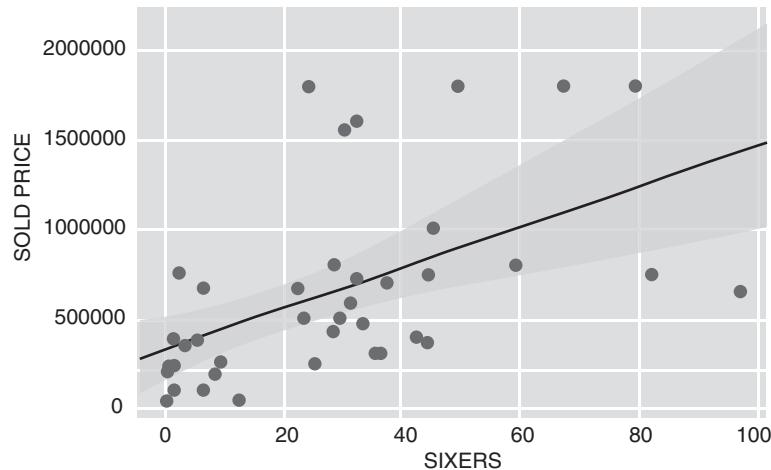


FIGURE 2.13 Scatter plot for SOLD PRICE versus SIXERS with a regression line.

The line in Figure 2.13 shows there is a positive correlation between number of sixes hit by a batsman and the SOLD PRICE.

2.3.8 | Pair Plot

If there are many variables, it is not convenient to draw scatter plots for each pair of variables to understand the relationships. So, a pair plot can be used to depict the relationships in a single diagram which can be plotted using *pairplot()* method.

In this example, we will explore the relationship of four variables, *SR-B*, *AVE*, *SIXERS*, *SOLD PRICE*, which we think may be the influential features in determining the *SOLD PRICE* of batsmen (Figure 2.14).

```
influential_features = ['SR-B', 'AVE', 'SIXERS', 'SOLD PRICE']
```

```
sn.pairplot(ipl_auction_df[influential_features], size=2)
```

The plot is drawn like a matrix and each row and column is represented by a variable. Each cell depicts the relationship between two variables, represented by that row and column variable. For example, the cell on second row and first column shows the relationship between *AVE* and *SR-B*. The diagonal of the matrix shows the distribution of the variable.

For all the correlations, *AVE* and *SIXERS* seem to be highly correlated with *SOLD PRICE* compared to *SR-B*.

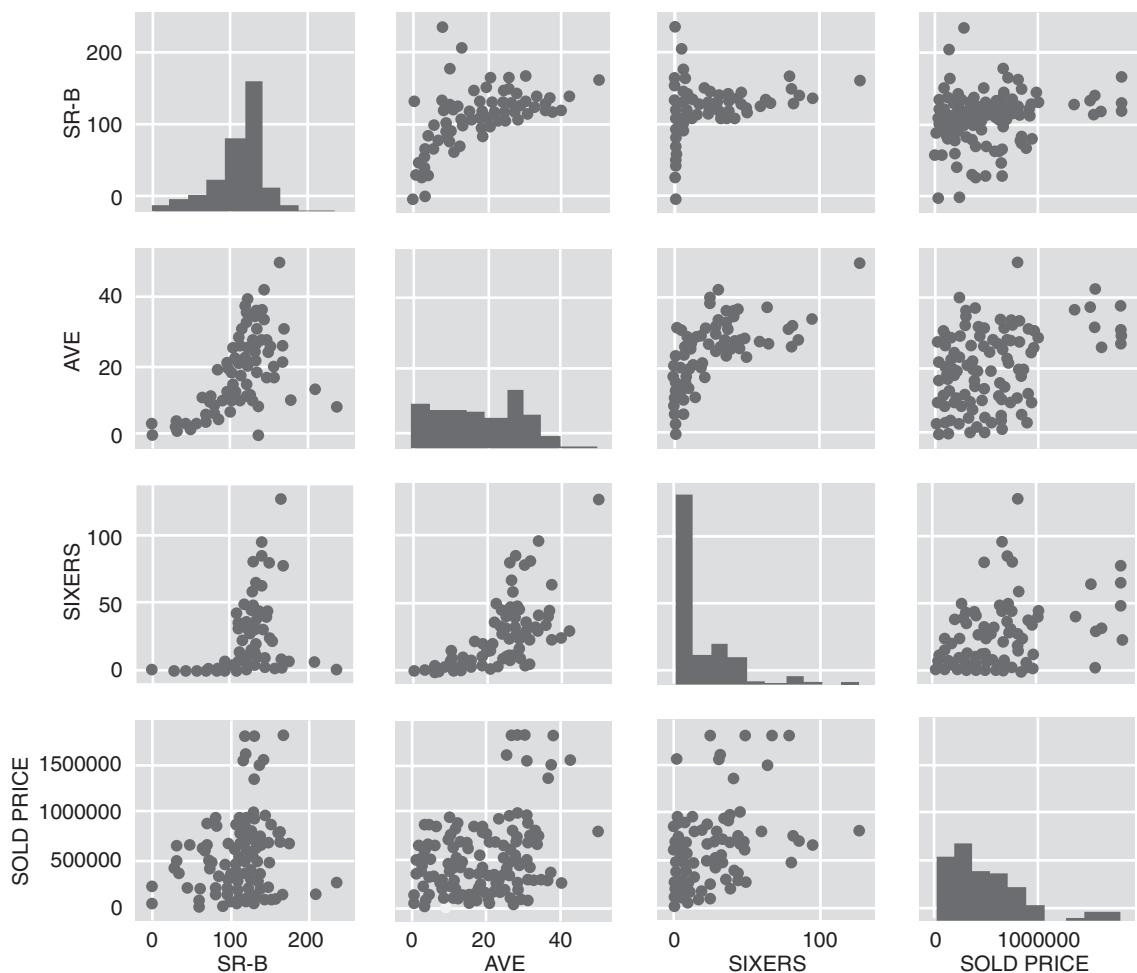


FIGURE 2.14 Pair plot between variables.

2.3.9 | Correlation and Heatmap

Correlation is used for measuring the strength and direction of the linear relationship between two continuous random variables X and Y . It is a statistical measure that indicates the extent to which two variables change together. A positive correlation means the variables increase or decrease together; a negative correlation means if one variable increases, the other decreases.

1. The correlation value lies between -1.0 and 1.0 . The sign indicates whether it is positive or negative correlation.
2. -1.0 indicates a perfect negative correlation, whereas $+1.0$ indicates perfect positive correlation.

Correlation values can be computed using `corr()` method of the DataFrame and rendered using a heatmap (Figure 2.15).

```
ipl_auction_df[influential_features].corr()
```

	SR-B	AVE	Sixers	Sold Price
SR-B	1.000000	0.583579	0.425394	0.184278
AVE	0.583579	1.000000	0.705365	0.396519
Sixers	0.425394	0.705365	1.000000	0.450609
Sold Price	0.184278	0.396519	0.450609	1.000000

```
sn.heatmap(ipl_auction_df[influential_features].corr(), annot=True);
```



FIGURE 2.15 Heatmap of correlation values.

The color map scale is shown along the heatmap. Setting *annot* attribute to *true* prints the correlation values in each box of the heatmap and improves readability of the heatmap. Here the heatmap shows that *AVE* and *SIXER* show positive correlation, while *SOLD PRICE* and *SR-B* are not so strongly correlated.

CONCLUSION

1. The objective of descriptive analytics is simple comprehension of data using data summarization, basic statistical measures and visualization.
2. DataFrames, which can be visualized as in-memory structured query language (SQL) tables, are widely used data structures for loading and manipulating structured data.
3. *Pandas* library provides excellent support for DataFrame and its operations. Operations like filtering, grouping, aggregations, sorting, joining and many more are readily available in this library.
4. *Matplotlib* and *seaborn* are two most widely used libraries for creating visualization.
5. Plots like histogram, distribution plots, box plots, scatter plots, pair plots, heatmap can be created to find insights during exploratory analysis.

EXERCISES**Use the Bollywood Dataset to Answer Questions 1 to 12.**

The data file *bollywood.csv* (link to the datasets is provided in the Preface) contains box office collection and social media promotion information about movies released in 2013–2015 period. Following are the columns and their descriptions.

- *SlNo* – Release Date
- *MovieName* – Name of the movie
- *ReleaseTime* – Mentions special time of release. LW (Long weekend), FS (Festive Season), HS (Holiday Season), N (Normal)
- *Genre* – Genre of the film such as Romance, Thriller, Action, Comedy, etc.
- *Budget* – Movie creation budget
- *BoxOfficeCollection* – Box office collection
- *YoutubeViews* – Number of views of the YouTube trailers
- *YoutubeLikes* – Number of likes of the YouTube trailers
- *YoutubeDislikes* – Number of dislikes of the YouTube trailers

Use Python code to answer the following questions:

1. How many records are present in the dataset? Print the metadata information of the dataset.
2. How many movies got released in each genre? Which genre had highest number of releases? Sort number of releases in each genre in descending order.
3. How many movies in each genre got released in different release times like long weekend, festive season, etc. (Note: Do a cross tabulation between *Genre* and *ReleaseTime*.)
4. Which month of the year, maximum number movie releases are seen? (Note: Extract a new column called month from *ReleaseDate* column.)
5. Which month of the year typically sees most releases of high budgeted movies, that is, movies with budget of 25 crore or more?
6. Which are the top 10 movies with maximum return on investment (ROI)? Calculate return on investment (ROI) as $(\text{BoxOfficeCollection} - \text{Budget}) / \text{Budget}$.
7. Do the movies have higher ROI if they get released on festive seasons or long weekend? Calculate the average ROI for different release times.
8. Draw a histogram and a distribution plot to find out the distribution of movie budgets. Interpret the plot to conclude if the most movies are high or low budgeted movies.
9. Compare the distribution of ROIs between movies with comedy genre and drama. Which genre typically sees higher ROIs?
10. Is there a correlation between box office collection and YouTube likes? Is the correlation positive or negative?
11. Which genre of movies typically sees more YouTube likes? Draw boxplots for each genre of movies to compare.
12. Which of the variables among *Budget*, *BoxOfficeCollection*, *YoutubeView*, *YoutubeLikes*, *YoutubeDislikes* are highly correlated? Note: Draw pair plot or heatmap.

Use the SAheart Dataset to Answer Questions 13 to 20.

The dataset SAheart.data is taken from the link below:

<http://www-stat.stanford.edu/~tibs/ElemStatLearn/datasets/SAheart.data>

The dataset contains retrospective sample of males in a heart-disease high-risk region of the Western Cape, South Africa. There are roughly two controls per case of Coronary Heart Disease (CHD). Many of the CHD-positive men have undergone blood pressure reduction treatment and other programs to reduce their risk factors after their CHD event. In some cases, the measurements were made after these treatments. These data are taken from a larger dataset, described in Rousseauw et al. (1983), *South African Medical Journal*. It is a tab separated file (csv) and contains the following columns (source: <http://www-stat.stanford.edu>)

- *sbp* – Systolic blood pressure
 - *tobacco* – Cumulative tobacco (kg)
 - *ldl* – Low density lipoprotein cholesterol
 - *adiposity*
 - *famhist* – Family history of heart disease (Present, Absent)
 - *typea* – Type-A behavior
 - *obesity*
 - *alcohol* – Current alcohol consumption
 - *age* – Age at onset
 - *chd* – Response, coronary heart disease
13. How many records are present in the dataset? Print the metadata information of the dataset.
 14. Draw a bar plot to show the number of persons having CHD or not in comparison to they having family history of the disease or not.
 15. Does age have any correlation with sbp? Choose appropriate plot to show the relationship.
 16. Compare the distribution of tobacco consumption for persons having CHD and not having CHD. Can you interpret the effect of tobacco consumption on having coronary heart disease?
 17. How are the parameters sbp, obesity, age and ldl correlated? Choose the right plot to show the relationships.
 18. Derive a new column called *agegroup* from *age* column where persons falling in different age ranges are categorized as below.
 (0–15): young
 (15–35): adults
 (35–55): mid
 (55–): old
 19. Find out the number of CHD cases in different age categories. Do a barplot and sort them in the order of age groups.
 20. Draw a box plot to compare distributions of *ldl* for different age groups.

REFERENCES

1. U Dinesh Kumar (2017). *Business Analytics: The Science of Data-Driven Decision Making*, Wiley India Pvt. Ltd., India.
2. UC Irvine Machine Learning Repository: <https://archive.ics.uci.edu/ml/index.php>
3. The Elements of Statistical Learning (Stanford): <https://web.stanford.edu/~hastie/ElemStatLearn/>
4. Pandas Library: <https://pandas.pydata.org/>
5. Matplotlib Library: <https://matplotlib.org/>
6. Seaborn Library: <https://seaborn.pydata.org/>
7. Rousseauw J, du Plessis J, Benade A, Jordaan P, Kotze J, and Ferreira J (1983). Coronary Risk Factor Screening in Three Rural Communities, *South African Medical Journal*, 64, pp. 430–436.



CHAPTER

3

Probability Distributions and Hypothesis Tests

LEARNING OBJECTIVES

After completing this chapter, you will be able to

- Learn the concept of a random variable and its role in analytics.
- Understand different probability distributions and their applications in analytics.
- Learn how to derive insights from statistical measures such as mean, variance, probability distribution functions, confidence interval, etc.
- Learn how to formulate and carry out hypothesis tests such as one-sample Z -test, t -test, two-sample t -test, paired t -test, chi-square tests, and analysis of variance (ANOVA).

3.1 | OVERVIEW

Analytics applications involve tasks such as prediction of the probability of occurrence of an event, testing a hypothesis, building models to explain variations in a KPI (key performance indicator) that is of importance to the business, such as profitability, market share, demand, etc. Many important tasks in analytics deal with uncertain events and it is essential to understand probability theory that can be used to measure and predict uncertain events.

In this chapter, we will introduce the concepts of random variables, probability distributions, distribution parameters, and how they can be applied to solve real-world problems. We will also explore the concepts of various discrete and continuous distributions, probability density functions, probability mass function, and cumulative distribution function. Finally, we will discuss hypothesis testing and its importance in analytics, understand the concept of significance (α) value, the probability value (p -value), and various hypotheses tests such as Z -test, various t -tests, and chi-square goodness of fit tests.

3.2 | PROBABILITY THEORY – TERMINOLOGY

In this section, we will be discussing various terminologies that are used in probability theory.

3.2.1 | Random Experiment

In machine learning, we mostly deal with uncertain events. Random experiment is an experiment in which the outcome is not known with certainty. That is, the output of a random experiment cannot be predicted with certainty.

3.2.2 | Sample Space

Sample space is the universal set that consists of all possible outcomes of an experiment. Sample space is usually represented using the letter "S" and individual outcomes are called the elementary events. The sample space can be finite or infinite. Few random experiments and their sample spaces are discussed below:

Experiment: Outcome of a college application

Sample Space = $S = \{\text{admitted, not admitted}\}$

Experiment: Predicting customer churn at an individual customer level

Sample Space = $S = \{\text{Churn, No Churn}\}$

Experiment: Television Rating Point (TRP) for a television program

Sample Space = $S = \{X \mid X \in R, 0 \leq X \leq 100\}$, that is X is a real number that can take any value between 0 and 100%.

3.2.3 | Event

Event (E) is a subset of a sample space and probability is usually calculated with respect to an event. Examples of events include:

1. Number of cancellation of orders placed at an E-commerce portal site exceeding 10%.
2. The number of fraudulent credit card transactions exceeding 1%.
3. The life of a capital equipment being less than one year.
4. Number of warranty claims less than 10 for a vehicle manufacturer with a fleet of 2000 vehicles under warranty.

3.3 | RANDOM VARIABLES

Random variables play an important role in describing, measuring, and analyzing uncertain events such as customer churn, employee attrition, demand for a product, and so on. A random variable is a function that maps every outcome in the sample space to a real number. A random variable can be classified as discrete or continuous depending on the values it can take.

If the random variable X can assume only a finite or countably infinite set of values, then it is called a *discrete random variable*. Examples of discrete random variables are as follows:

1. Credit rating (usually classified into different categories such as low, medium, and high or using labels such as AAA, AA, A, BBB, etc.).
2. Number of orders received at an e-commerce retailer which can be countably infinite.
3. Customer churn [the random variables take binary values: (a) Churn and (b) Do not churn].
4. Fraud [the random variables take binary values: (a) Fraudulent transaction and (b) Genuine transaction].

A random variable X which can take a value from an infinite set of values is called a *continuous random variable*. Examples of continuous random variables are as follows:

1. Market share of a company (which take any value from an infinite set of values between 0 and 100%).
2. Percentage of attrition of employees of an organization.
3. Time-to-failure of an engineering system.
4. Time taken to complete an order placed at an e-commerce portal.

Discrete random variables are described using *probability mass function* (PMF) and *cumulative distribution function* (CDF). PMF is the probability that a random variable X takes a specific value k ; for example, the number of fraudulent transactions at an e-commerce platform is 10, written as $P(X = 10)$. On the other hand, CDF is the probability that a random variable X takes a value less than or equal to 10, which is written as $P(X \leq 10)$.

Continuous random variables are described using *probability density function* (PDF) and cumulative distribution function (CDF). PDF is the probability that a continuous random variable takes value in a small neighborhood of “ x ” and is given by

$$f(x) = \lim_{\delta x \rightarrow 0} P[x \leq X \leq x + \delta x] \quad (3.1)$$

The CDF of a continuous random variable is the probability that the random variable takes value less than or equal to a value “ a ”. Mathematically,

$$F(a) = \int_{-\infty}^a f(x)dx \quad (3.2)$$

In the following sections, we will discuss the concepts of various discrete and continuous distributions and how to apply them to real-world scenarios.

3.4 | BINOMIAL DISTRIBUTION

Binomial distribution is a discrete probability distribution and has several applications in many business contexts. A random variable X is said to follow a binomial distribution when:

1. The random variable can have only two outcomes – success and failure (also known as Bernoulli trials).
2. The objective is to find the probability of getting x successes out of n trials.
3. The probability of success is p and thus the probability of failure is $(1 - p)$.
4. The probability p is constant and does not change between trials.

Success and failure are generic terminologies used in binomial distribution, based on the context we will interpret success and failure. Few examples of business problems with two possible outcomes are as follows:

1. Customer churn where the outcomes are: (a) Customer churn and (b) No customer churn.
2. Fraudulent insurance claims where the outcomes are: (a) Fraudulent claim and (b) Genuine claim.
3. Loan repayment default by a customer where the outcomes are: (a) Default and (b) No default.

The PMF of the binomial distribution (probability that the number of success will be exactly x out of n trials) is given by

$$PMF(x) = P(X = x) = \binom{n}{x} \times p^x \times (1-p)^{n-x} \quad (3.3)$$

where

$$\binom{n}{x} = \frac{n!}{x!(n-x)!}$$

The CDF of a binomial distribution (probability that the number of success will be x or less than x out of n trials) is given by

$$CDF(x) = P(X \leq x) = \sum_{k=0}^x \binom{n}{k} \times p^k \times (1-p)^{n-k} \quad (3.4)$$

In Python, the `scipy.stats.binom` class provides methods to work with binomial distribution.

3.4.1 | Example of Binomial Distribution

Fashion Trends Online (FTO) is an e-commerce company that sells women apparel. It is observed that 10% of their customers return the items purchased by them for many reasons (such as size, color, and material mismatch). On a specific day, 20 customers purchased items from FTO. Calculate:

1. Probability that exactly 5 customers will return the items.
2. Probability that a maximum of 5 customers will return the items.
3. Probability that more than 5 customers will return the items purchased by them.
4. Average number of customers who are likely to return the items and the variance and the standard deviation of the number of returns.

We solve each of these as follows:

1. Probability that exactly 5 customers will return the items.

The function `stats.binom.pmf()` calculates PMF for binomial distribution and takes three parameters:

- (a) Expected number of successful trials (5)
- (b) Total number of trials (20)
- (c) The probability of success (0.1)

Note: The values in the bracket indicate the value of the parameters.

```
from scipy import stats
stats.binom.pmf(5, 20, 0.1)
```

The corresponding probability is 0.03192, that is, the probability that exactly 5 customers will return the items is approximately 3%.

To visualize how the PMF varies with increasing number of successful trials, we will create a list of all possible number of successes (0 to 20) and corresponding PMF values and draw a bar plot as shown in Figure 3.1.

```
# range(0,21) returns all values from 0 to 20 (excluding 21)
pmf_df = pd.DataFrame({'success': range(0,21),
                       'pmf': list(stats.binom.pmf(range(0,21),
                                                    20, 0.1))})
# Creating a bar plot with number of success as x and pmf as y
sns.barplot(x = pmf_df.success, y = pmf_df.pmf)
plt.ylabel('pmf')
plt.xlabel('Number of items returned');
```

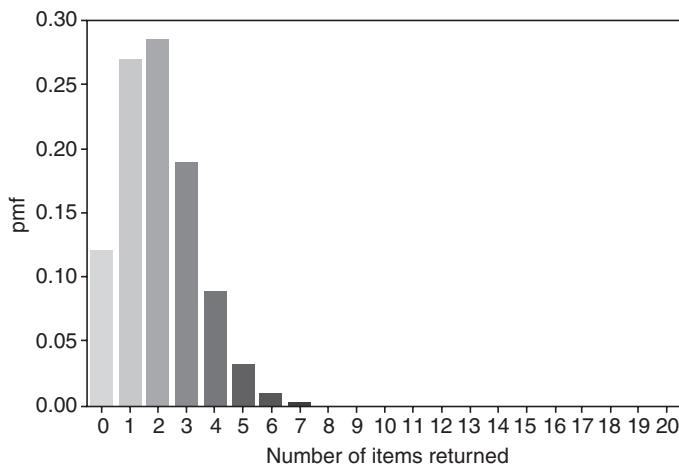


FIGURE 3.1 Binomial distribution.

2. Probability that a maximum of 5 customers will return the items.

The class `stats.binom.cdf()` computes the CDF for binomial distribution. In this case the cumulative distribution function returns the probability that a maximum of 5 customers will return items.

```
stats.binom.cdf(5, 20, 0.1)
```

The corresponding probability value is 0.9887.

3. Probability that more than 5 customers will return the items purchased by them.

Total probability of any number of customers returning items (including 0) is always equal to 1.0. So, the probability that more than 5 customers will return the items can be computed by subtracting the probability of a maximum of 5 customers will return items from 1.0. In other words, the probability that more than 5 customers will return the items can be obtained by computing CDF of 5 and then subtracting it from 1.0.

```
1 - stats.binom.cdf(5, 20, 0.1)
```

The corresponding probability value is 0.0112.

4. Average number of customers who are likely to return the items and the variance and the standard deviation of the number of returns.

- (a) Average of a binomial distribution is given by $n * p$
- (b) Variance of the binomial distribution is given by $n * p * (1 - p)$

```
mean, var = stats.binom.stats(20, 0.1)
print("Average: ", mean, " Variance:", var)
```

Average: 2.0 Variance: 1.8

3.5 | POISSON DISTRIBUTION

In many situations, we may be interested in calculating the number of events that may occur over a period of time or space. For example, number of cancellation of orders by customers at an e-commerce portal, number of customer complaints, number of cash withdrawals at an ATM, number of typographical errors in a book, number of potholes on Bangalore roads, etc. To find the probability of number of events, we use Poisson distribution. The PMF of a Poisson distribution is given by

$$P(X = k) = \frac{e^{-\lambda} \times \lambda^k}{k!} \quad (3.5)$$

where λ is the rate of occurrence of the events per unit of measurement (in many situations the unit of measurement is likely to be time).

3.5.1 | Example of Poisson Distribution

The number of calls arriving at a call center follows a Poisson distribution at 10 calls per hour.

1. Calculate the probability that the number of calls will be maximum 5.
2. Calculate the probability that the number of calls over a 3-hour period will exceed 30.

We solve each of these as follows:

1. Calculate the probability that a maximum of 5 calls will arrive at the call center.

As the number of calls arriving at the center follows Poisson distribution, we can use `stats.poisson.cdf` to calculate the probability value. It takes the following two parameters:

- (a) First parameter: Number of events (in this case, 5 calls) for which the probability needs to be calculated.
- (b) Second parameter: The average numbers of events (i.e., 10 calls per hour).

```
stats.poisson.cdf(5, 10)
```

0.06708

The corresponding probability is 0.067.

2. Calculate the probability that the number of calls over a 3-hour period will exceed 30.

Since the average calls per hour is 10 ($\lambda = 10$), and we are interested in finding the calls over 3 hours, the mean number of calls over 3 hours is $\lambda t = 30$. Probability that the number of calls will be more than 30 is given by

```
1 - stats.poisson.cdf(30, 30)
```

0.45164

The corresponding probability is 0.451.

To visualize the Poisson distribution for the average calls per hour as 10, we can plot PMF for all possible number of calls the call center can receive ranging from 0 to 30. We will create a DataFrame which will contain the number of calls ranging from 0 to 30 in one column named *success* and the corresponding PMFs in another column named *pmf*. The plotting is done using *barplot* in *seaborn* library.

```
# Range(0,30) returns all values from 0 to 30 (excluding 30)
pmf_df = pd.DataFrame({'success': range(0,30), 'pmf': list(stats.poisson.pmf(range(0,30), 10))})
# Creating a barplot with number of calls as x and pmf as y
sns.barplot(x = pmf_df.success, y = pmf_df.pmf);
plt.xlabel('Number of Calls Received');
```

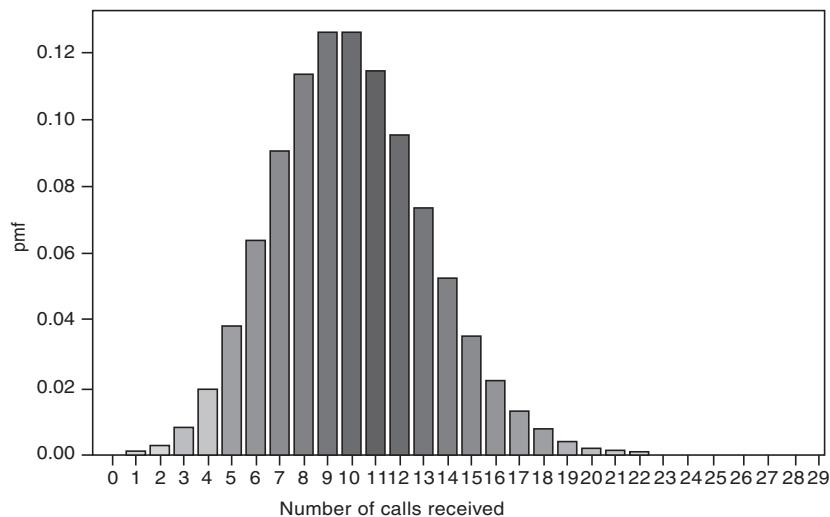


FIGURE 3.2 Poisson distribution.

The corresponding Poisson distribution plot is shown in Figure 3.2.

3.6 | EXPONENTIAL DISTRIBUTION

Exponential distribution is a single parameter continuous distribution that is traditionally used for modeling time-to-failure of electronic components. The exponential distribution represents a process in which events occur continuously and independently at a constant average rate.

The probability density function is given by

$$f(x) = \lambda e^{-\lambda x}, x \geq 0$$

where

1. The parameter λ is the scale parameter and represents the rate of occurrence of the event.
2. Mean of exponential distribution is given by $1/\lambda$.

3.6.1 | Example of Exponential Distribution

The time-to-failure of an avionic system follows an exponential distribution with a mean time between failures (MTBF) of 1000 hours. Calculate

1. The probability that the system will fail before 1000 hours.
2. The probability that it will not fail up to 2000 hours.
3. The time by which 10% of the system will fail (i.e., calculate P10 life).

We solve each of these as follows: Since time-to-failure is 1000 hours, so λ is 1/1000.

1. Calculate the probability that the system will fail before 1000 hours.

Cumulative distribution up to value 1000 for the exponential distribution will give the probability that the system will fail before 1000 hours. `stats.expon.cdf()` takes the number of hours and mean and scale of the exponential distribution as parameters to calculate CDF.

```
stats.expon.cdf(1000,
                 loc = 1/1000,
                 scale = 1000)
```

0.6321

The corresponding probability value is 0.6321.

2. Calculate the probability that it will not fail up to 2000 hours.

Probability that the system will not fail up to 2000 hours is same as the probability that the system will fail only beyond 2000 hours. This can be obtained by subtracting the probability that the system will fail up to 2000 hours from total probability (i.e. 1.0).

```
1 - stats.expon.cdf(2000,
                     loc = 1/1000,
                     scale = 1000)
```

0.1353

The corresponding probability value 0.1353.



3. Calculate the time by which 10% of the system will fail (i.e., calculate P10 life).

This can be calculated by ppf (percent point function) and is an inverse of CDF. *stats.expon.ppf* takes the percent point value and the mean and scale of the exponential distribution.

```
stats.expon.ppf(.1,  
                 loc = 1/1000,  
                 scale = 1000)
```

105.3615

That is, by 105.36 hours, 10% of systems will fail.

We can visualize the exponential distribution by plotting the PDF function against different time-to-failure hours. We will create a list of time-to-failure ranging from 100 to 5000 and then calculate and plot the PDF against those.

```
pdf_df = pd.DataFrame({'success': range(0,5000, 100),  
                      'pdf':  
                         list(stats.expon.pdf(range(0, 5000, 100),  
                                              loc = 1/1000,  
                                              scale = 1000))})  
  
plt.figure(figsize=(10,4))  
sns.barplot(x = pdf_df.success, y = pdf_df.pdf)  
plt.xticks(rotation=90);  
plt.xlabel('Time to failure');
```

The corresponding exponential distribution is shown in Figure 3.3.

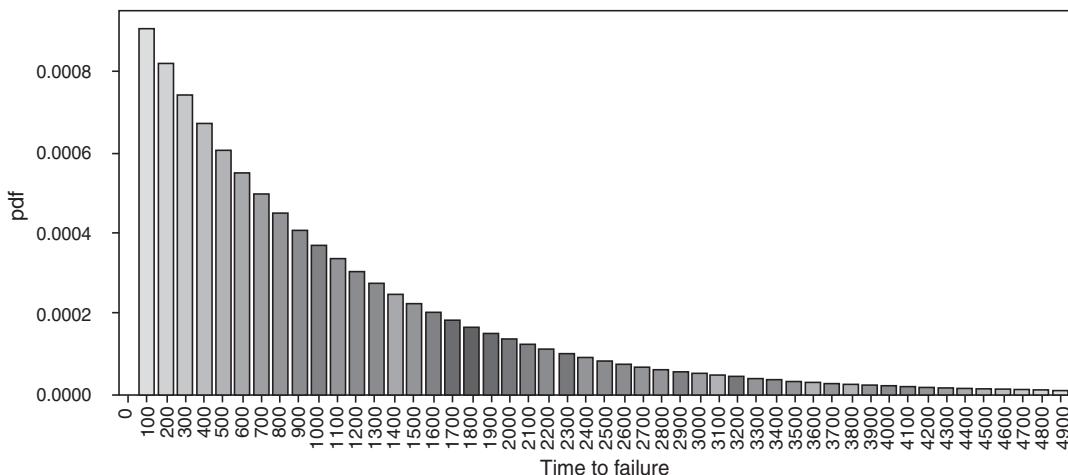


FIGURE 3.3 Exponential distribution.

3.7 | NORMAL DISTRIBUTION

Normal distribution, also known as *Gaussian distribution*, is one of the most popular continuous distribution in the field of analytics especially due to its use in multiple contexts. Normal distribution is observed across many naturally occurring measures such as age, salary, sales volume, birth weight, height, etc. It is also popularly known as bell curve (as it is shaped like a bell).

3.7.1 | Example of Normal Distribution

To understand normal distribution and its application, we will use daily returns of stocks traded in BSE (Bombay Stock Exchange). Imagine a scenario where an investor wants to understand the risks and returns associated with various stocks before investing in them. For this analysis, we will evaluate two stocks: *BEML* and *GLAXO*. The daily trading data (open and close price) for each stock is taken for the period starting from 2010 to 2016 from BSE site (www.bseindia.com).

First, we will load and prepare the data before getting back to the application of normal distribution.

```
import pandas as pd
import numpy as np
import warnings

beml_df = pd.read_csv('BEML.csv')
beml_df[0:5]
```

	Date	Open	High	Low	Last	Close	Total Trade Quantity	Turnover (Lacs)
0	2010-01-04	1121.0	1151.00	1121.00	1134.0	1135.60	101651.0	1157.18
1	2010-01-05	1146.8	1149.00	1128.75	1135.0	1134.60	59504.0	676.47
2	2010-01-06	1140.0	1164.25	1130.05	1137.0	1139.60	128908.0	1482.84
3	2010-01-07	1142.0	1159.40	1119.20	1141.0	1144.15	117871.0	1352.98
4	2010-01-08	1156.0	1172.00	1140.00	1141.2	1144.05	170063.0	1971.42

```
glaxo_df = pd.read_csv('GLAXO.csv')
glaxo_df[0:5]
```

	Date	Open	High	Low	Last	Close	Total Trade Quantity	Turnover (Lacs)
0	2010-01-04	1613.00	1629.10	1602.00	1629.0	1625.65	9365.0	151.74
1	2010-01-05	1639.95	1639.95	1611.05	1620.0	1616.80	38148.0	622.58
2	2010-01-06	1618.00	1644.00	1617.00	1639.0	1638.50	36519.0	595.09
3	2010-01-07	1645.00	1654.00	1636.00	1648.0	1648.70	12809.0	211.00
4	2010-01-08	1650.00	1650.00	1626.55	1640.0	1639.80	28035.0	459.11

The dataset contains daily Open and Close price along with daily High and Low prices, Total Trade Quantity, and Turnover (Lacs). Our discussion will involve only close price. The daily returns of a stock are calculated as the change in close prices with respect to the close price of yesterday.



Since our analysis will involve only daily close prices, so we will select *Date* and *Close* columns from the DataFrames.

```
beml_df = beml_df[['Date', 'Close']]  
glaxo_df = glaxo_df[['Date', 'Close']]
```

Visualizing the daily close prices will show how stock prices have moved over time. To show the trend of close price, the rows should be ordered by time. The DataFrames have a date column, so we can create a *DatetimeIndex* index from this column *Date*. It will ensure that the rows are sorted by time in ascending order.

```
glaxo_df = glaxo_df.set_index(pd.DatetimeIndex(glaxo_df['Date']))  
beml_df = beml_df.set_index(pd.DatetimeIndex(beml_df['Date']))
```

Let us display the first 5 records after the DataFrame is sorted by time to ensure that it is done correctly.

```
glaxo_df.head(5)
```

	Date	Close
Date		
2010-01-04	2010-01-04	1625.65
2010-01-05	2010-01-05	1616.80
2010-01-06	2010-01-06	1638.50
2010-01-07	2010-01-07	1648.70
2010-01-08	2010-01-08	1639.80

Now plot the trend of close prices of GLAXO stock using *plot()* method of matplotlib, which takes *glaxo_df.Close* as a parameter. The trend is shown in Figure 3.4.

```
import matplotlib.pyplot as plt  
import seaborn as sn  
%matplotlib inline  
  
plt.plot(glaxo_df.Close);  
plt.xlabel('Time');  
plt.ylabel('Close Price');
```

Now plot BEML stock close price trend. The trend is shown in Figure 3.5.

```
plt.plot(beml_df.Close);  
plt.xlabel('Time');  
plt.ylabel('Close');
```



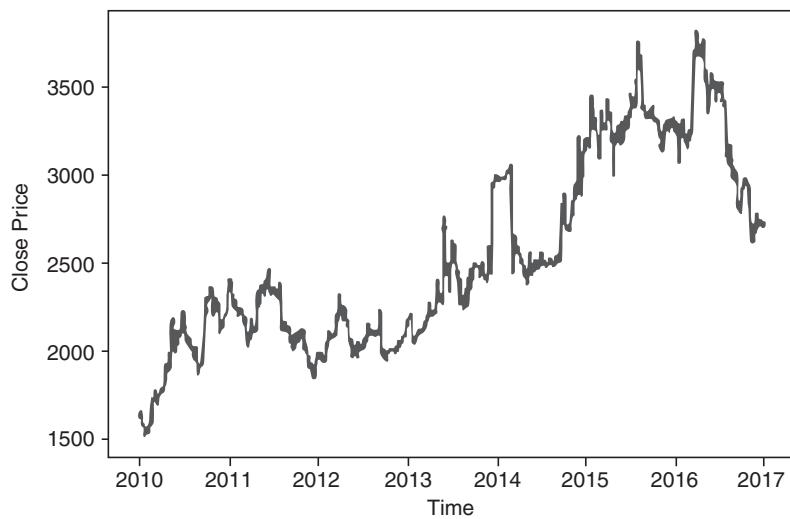


FIGURE 3.4 Close price trends of GLAXO stock.

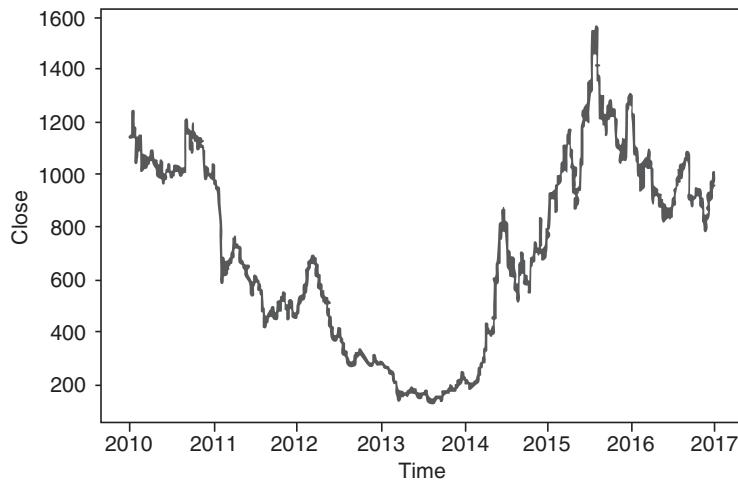


FIGURE 3.5 Close price trends of BEML stock.

It can be observed that there is an upward trend in the close price of Glaxo during 2010–2017 period. However, BEML had a downward trend during 2010–2013, followed by an upward trend since 2014 and then again a price correction from mid of 2015 (Figure 3.5).

What if a short-term (intraday) investor is interested in understanding the following characteristics about these stocks:

1. What is the expected daily rate of return of these stocks?
2. Which stocks have higher risk or volatility as far as daily returns are concerned?
3. What is the expected range of return for 95% confidence interval?

4. Which stock has higher probability of making a daily return of 2% or more?
5. Which stock has higher probability of making a loss (risk) of 2% or more?

To answer the above questions, we must find out the behavior of daily returns (we will refer to this as *gain* hence forward) on these stocks. The *gain* can be calculated as a percentage change in *close price*, from the previous day's *close price*.

$$gain = \frac{ClosePrice_t - ClosePrice_{t-1}}{ClosePrice_{t-1}}$$

The method *pct_change()* in Pandas will give the percentage change in a column value shifted by a period, which is passed as a parameter to *periods*. *periods = 1* indicates the value change since last row, that is, the previous day.

```
glaxo_df['gain'] = glaxo_df.Close.pct_change(periods = 1)
beml_df['gain'] = beml_df.Close.pct_change(periods = 1)
glaxo_df.head(5)
```

Date	Close	Gain
Date		
2010-01-04	1625.65	NaN
2010-01-05	1616.80	-0.005444
2010-01-06	1638.50	0.013422
2010-01-07	1648.70	0.006225
2010-01-08	1639.80	-0.005398

The first day gain is shown as *NAN*, as there is no previous day for it to calculate *gain*. We can drop this record using the *dropna()* method.

```
glaxo_df = glaxo_df.dropna()
beml_df = beml_df.dropna()
```

Now, plot *gain* against time (Figure 3.6).

```
plt.figure(figsize = (8, 6));
plt.plot(glaxo_df.index, glaxo_df.gain);
plt.xlabel('Time');
plt.ylabel('gain');
```

The plot in Figure 3.6 shows that the daily gain is highly random and fluctuates around 0.00. The gain remains mostly between 0.05 and -0.05. However, very high gain close to 0.20 has been observed once and similarly, high loss of around 0.08 has been observed once. We can draw distribution plot of gain of both BEML and Glaxo stocks to gain better insight (see Figure 3.7).

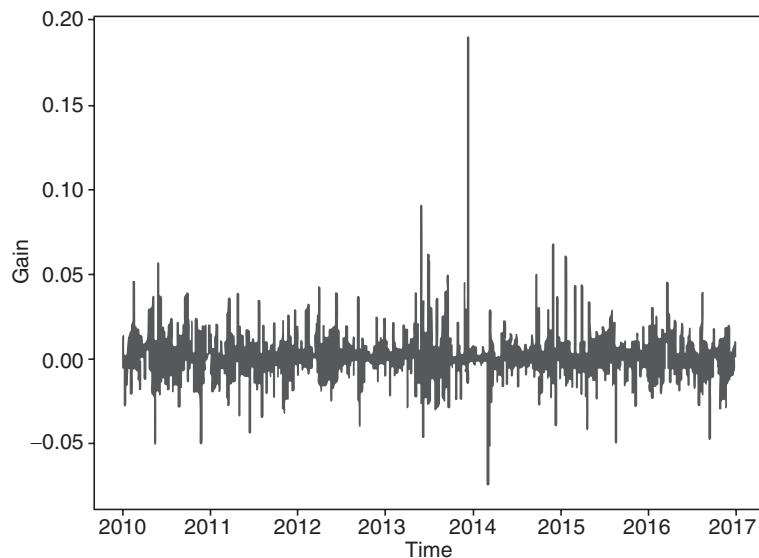


FIGURE 3.6 Daily gain of BEML stock.

```
sn.distplot(glaxo_df.gain, label = 'Glaxo');
sn.distplot(beml_df.gain, label = 'BEML');
plt.xlabel('gain');
plt.ylabel('Density');
plt.legend();
```

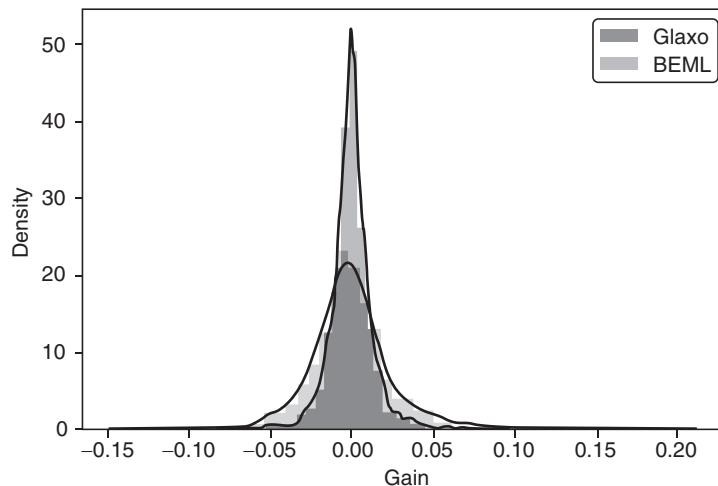


FIGURE 3.7 Distribution plot of daily gain of BEML and Glaxo stocks.

From the plot shown in Figure 3.7, *gain* seems to be normally distributed for both the stocks with a mean around 0.00. BEML seems to have a higher variance than Glaxo.

Note: This distribution has a long tail, but we will assume normal distribution for simplicity and discuss the example.

3.7.2 | Mean and Variance

The normal distribution is parameterized by two parameters: the mean of the distribution μ and the variance σ^2 . The sample mean of a normal distribution is given by

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$$

Variance is given by

$$\sigma^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2$$

The *standard deviation* is square root of variance and is denoted by σ .

Methods *mean()* and *std()* on DataFrame columns return mean and standard deviation, respectively. Mean and standard deviation for daily returns for Glaxo are:

```
print("Daily gain of Glaxo")
print("-----")
print("Mean: ", round(glaxo_df.gain.mean(), 4))
print("Standard Deviation: ", round(glaxo_df.gain.std(), 4))
```

```
Daily gain of Glaxo
-----
Mean:           0.0004
Standard Deviation: 0.0134
```

Mean and standard deviation for daily returns for BEML are:

```
print("Daily gain of BEML")
print("-----")
print("Mean: ", round(beml_df.gain.mean(), 4))
print("Standard Deviation: ", round(beml_df.gain.std(), 4))
```

```
Daily gain of BEML
-----
Mean:           0.0003
Standard Deviation: 0.0264
```

The *describe()* method of DataFrame returns the detailed statistical summary of a variables.

```
beml_df.gain.describe()

count      1738.000000
mean       0.000271
std        0.026431
min       -0.133940
25%       -0.013736
50%       -0.001541
75%        0.011985
max        0.198329
Name: gain, dtype: float64
```

The expected daily rate of return (*gain*) is around 0% for both stocks. Here variance or standard deviation of *gain* indicates risk. So, BEML stock has a higher risk as standard deviation of BEML is 2.64% whereas the standard deviation for Glaxo is 1.33%.

3.7.3 | Confidence Interval

To find out what is the expected range of return for 95% confidence interval, we need to calculate the values of *gain* for two standard deviations away from mean on both sides of the distribution, that is, $\mu \pm 2\sigma$.

For advanced statistical analysis, we can use Python library *scipy.stats*. The library contains sub-packages for analyzing different distributions. For example, methods to analyze normal distribution are given by *stats.norm*. The *interval()* method of *stats.norm* gives the confidence interval for a normal distribution.

stats.norm.interval() takes three parameters:

1. **alpha**: It is the interval, for example, 0.9 for 90% confidence interval.
2. **loc**: It is the location parameter of the distribution. It is *mean* for normal distribution.
3. **scale**: It is the scale parameter of the distribution. It is *standard deviation* for normal distribution.

```
from scipy import stats

glaxo_df_ci = stats.norm.interval(0.95,
                                 loc = glaxo_df.gain.mean(),
                                 scale = glaxo_df.gain.std())

print("Gain at 95% confidence interval is: ", np.round(glaxo_df_ci, 4))
```

Gain at 95% confidence interval is: [-0.0258 0.0266]

The result returned by the method is a tuple. The first value of the tuple is the leftmost value of the interval and second value is the rightmost value of the interval. For 95% confidence interval, *gain* of Glaxo remains between -2.58% and 2.66%.

Returns for BEML for 95% confidence interval is given by

```
beml_df_ci = stats.norm.interval(0.95,
                                 loc=beml_df.gain.mean(),
                                 scale=beml_df.gain.std())
```

```
print("Gain at 95% confidence interval is:", np.round(beml_df_ci, 4))
```

Gain at 95% confidence interval is: [-0.0515 0.0521]

Hence *gain* of BEML remains between -5.15% and 5.21% for 95% confidence interval.

3.7.4 | Cumulative Probability Distribution

To calculate the probability of gain higher than 2% or more, we need to find out what is the sum of all probabilities that *gain* can take values more than 0.02 (i.e., 2%).

The probability density function, $f(x_i)$, is defined as the probability that the value of random variable X lies between an infinitesimally small interval defined by x_i and $x_i + \delta x$. Cumulative distribution function $F(a)$ is the area under the probability density function up to $X = a$. The cumulative distribution function of a continuous random variable is as shown in Figure 3.8.

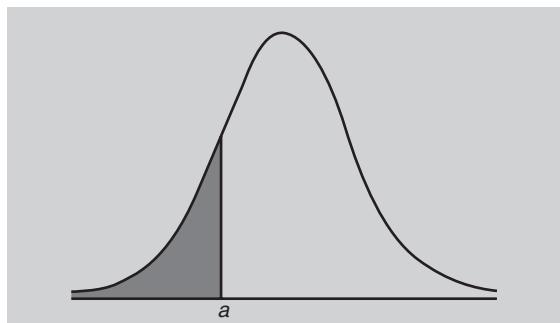


FIGURE 3.8 Cumulative distribution function $F(a)$.

To calculate the probability that the stocks' gain will be less than -0.02 (i.e., loss), cumulative distribution function can be used to calculate the area of the distribution from leftmost point up to -0.02. The *stats.norm.cdf()* class returns cumulative distribution for a normal distribution.

```
print("Probability of making 2% loss or higher in Glaxo: ")
stats.norm.cdf(-0.02,
                loc=glaxo_df.gain.mean(),
                scale=glaxo_df.gain.std())
```

Probability of making 2% loss or higher in Glaxo:
0.0635

```
print("Probability of making 2% loss or higher in BEML: ")
stats.norm.cdf(-0.02,
                loc=beml_df.gain.mean(),
                scale=beml_df.gain.std())
```

Probability of making 2% loss or higher in BEML:
0.2215

Cumulative distribution function value indicates that BEML has 22.1% probability, whereas Glaxo has only 6.35% probability of making a loss of 2% or higher. Similarly, the probability of making a daily gain of 2% or higher will be given by the area to the right of 0.02 of the distribution. As `stats.norm.cdf()` gives the cumulative area from left, the probability can be computed by subtracting the cumulative distribution function value from 1.

```
print("Probability of making 2% gain or higher in Glaxo: ",
      1 - stats.norm.cdf(0.02,
                          loc=glaxo_df.gain.mean(),
                          scale=glaxo_df.gain.std()))

print("Probability of making 2% gain or higher in BEML: ",
      1 - stats.norm.cdf(0.02,
                          loc=beml_df.gain.mean(),
                          scale=beml_df.gain.std()))
```

Probability of making 2% gain or higher in Glaxo: 0.0710
 Probability of making 2% gain or higher in BEML: 0.2276

The probability of making a *gain* of 2% or more for *Glaxo* is 7.1%, whereas it is 22.76% for *BEML*.

3.7.5 | Other Important Distributions

Probability distributions are fundamental to statistics, just like data structures are to computer science. Apart from the distributions discussed till now, following are few other important distributions:

- Beta Distribution:** It is a continuous distribution which can take values between 0 and 1.
- Gamma Distribution:** It is used to model waiting times in different contexts. We can model time until the next n events occur using Gamma distribution.
- Weibull Distribution:** It is a very popular distribution in reliability theory. Exponential distribution is a special case of Weibull distribution. In exponential distribution, the rate of occurrence of an event is constant, but using Weibull distribution we can model increasing (or decreasing) rates of occurrence of an event over time.
- Geometric Distribution:** If the binomial distribution is about “How many successes?” then the geometric distribution is about “How many failures until a success?”.

3.8 | CENTRAL LIMIT THEOREM

Central limit theorem (CLT) is one of the most important theorems in statistics due to its applications in the testing of hypothesis. Let S_1, S_2, \dots, S_k be samples of size n drawn from an independent and identically distributed population with mean μ and standard deviation σ . Let X_1, X_2, \dots, X_k be the sample means (of the samples S_1, S_2, \dots, S_k). According to the CLT, the distribution of X_1, X_2, \dots, X_k follows a normal distribution with mean μ and standard deviation of σ/\sqrt{n} . That is, the sampling distribution of mean will follow a normal distribution with mean μ (same as the mean of the population) and standard deviation σ/\sqrt{n} . We will understand in the following sections that central limit theorem is key to hypothesis testing, which primarily deals with sampling distribution.

3.9 | HYPOTHESIS TEST

Hypothesis is a claim and the objective of hypothesis testing is to either reject or retain a *null hypothesis* (current belief) with the help of data. Hypothesis testing consists of two complementary statements called *null hypothesis* and *alternative hypothesis*. *Null hypothesis* is an existing belief and *alternate hypothesis* is what we intend to establish with new evidences (samples).

Hypothesis tests are broadly classified into parametric tests and non-parametric tests. Parametric tests are about population parameters of a distribution such as mean, proportion, standard deviation, etc., whereas non-parametric tests are not about parameters, but about other characteristics such as independence of events or data following certain distributions such as normal distribution.

Few examples of the *null hypothesis* are as follows:

1. Children who drink the health drink Complan (a health drink produced by the company Heinz in India) are likely to grow taller.
2. Women use camera phone more than men (Freier, 2016).
3. Vegetarians miss few flights (Siegel, 2016).
4. Smokers are better sales people.

The steps for hypothesis tests are as follows:

1. Define null and alternative hypotheses. Hypothesis is described using a population parameter, that is, mean, standard deviation, etc. Normally, H_0 is used to denote null hypothesis and H_A for alternate hypothesis.
2. Identify the test statistic to be used for testing the validity of the null hypothesis, for example, Z-test or t-test.
3. Decide the criteria for rejection and retention of null hypothesis. This is called significance value (α). Typical value used for α is 0.05.
4. Calculate the *p*-value (probability value), which is the conditional probability of observing the test statistic value when the null hypothesis is true. We will use the functions provided in *scipy.stats* module for calculating the *p*-value.
5. Take the decision to reject or retain the null hypothesis based on the *p*-value and the significance value α .

For a detailed understanding of the hypothesis test, refer to Chapter 6 of the book *Business Analytics: The Science of Data-Driven Decision Making* by U Dinesh Kumar (2017), Wiley India.

For all examples, we will use the following notations:

1. μ – population mean
2. σ – population standard deviation
3. X – sample mean
4. S – sample standard deviation
5. n – sample size

Let us take a few examples and conduct the hypothesis test.

3.9.1 | Z-test

Z-test is used when

1. We need to test the value of population mean, given that population variance is known.
2. The population is a normal distribution and the population variance is known.
3. The sample size is large and the population variance is known. That is, the assumption of normal distribution can be relaxed for large samples ($n > 30$).

Z-statistic is calculated as

$$Z = \frac{X - \mu}{\sigma / \sqrt{n}}$$

3.1 EXAMPLE

A passport office claims that the passport applications are processed within 30 days of submitting the application form and all necessary documents. The file *passport.csv* contains processing time of 40 passport applicants. The population standard deviation of the processing time is 12.5 days. Conduct a hypothesis test at significance level $\alpha = 0.05$ to verify the claim made by the passport office.

In this case, the population mean (claim made by passport office) and standard deviation are known. Population mean is 30 and population standard deviation is 12.5. The dataset in *passport.csv* contains observations of actual processing time of 40 passports. We can calculate the mean of these observations and calculate Z-statistics. If the Z-statistics value is more than -1.64 (Z-critical value for left-tailed test), then it will be concluded that the processing time is not less than 30, but higher. And if the Z-statistic is less than -1.64, then it can be concluded that the processing time is less than 30 as claimed by passport office.

We now read the data and display first 5 records from *passport.csv* using the following code:

```
passport_df = pd.read_csv('passport.csv')
passport_df.head(5)
```

processing_time	
0	16.0
1	16.0
2	30.0
3	37.0
4	25.0

The following is used to print all the records from the dataset.

3.1 EXAMPLE (Continued)

```
print(list(passport_df.processing_time))  
  
[16.0, 16.0, 30.0, 37.0, 25.0, 22.0, 19.0, 35.0, 27.0, 32.0,  
34.0, 28.0, 24.0, 35.0, 24.0, 21.0, 32.0, 29.0, 24.0, 35.0,  
28.0, 29.0, 18.0, 31.0, 28.0, 33.0, 32.0, 24.0, 25.0, 22.0,  
21.0, 27.0, 41.0, 23.0, 23.0, 16.0, 24.0, 38.0, 26.0, 28.0]
```

Let us first define the hypothesis. If μ is population mean, that is, mean processing time of passports then

$$H_0: \mu \geq 30$$

$$H_A: \mu < 30$$

We will conduct a Z-test for this hypothesis test. We will define a method named `z_test`, which takes the population mean, population variance and the sample as parameter to calculate and return Z-score and the *p*-value.

```
import math  
  
def z_test(pop_mean, pop_std, sample):  
    z_score = (sample.mean() - pop_mean) / (pop_std/math.  
    sqrt(len(sample)))  
    return z_score, stats.norm.cdf(z_score)  
  
z_test(30, 12.5, passport_df.processing_time)  
(-1.4925, 0.0677)
```

The first value of the result is the Z-statistic value or Z-score and second value is the corresponding *p*-value. As the *p*-value is more than 0.05, the null hypothesis is retained. Also, Z-statistic value is higher than -1.64.

The *p*-value provides us significance of sample evidence. In this example, we see that there is 6.77% probability of observing a random sample at least as extreme as the observed sample. Since 6.77% is greater than the significance value 5%, there is not enough evidence to reject null hypothesis. The null hypothesis is retained and it can be concluded that average processing time of passports is greater than equal to 30.

3.9.2 | One-Sample t-Test

The *t*-test is used when the population standard deviation *S* is unknown (and hence estimated from the sample) and is estimated from the sample. Mathematically,

$$t\text{-Statistics} = \frac{(\bar{X} - \mu)}{S / \sqrt{n}}$$

The expected value (mean) of a sample of independent observations is equal to the given population mean.

3.2 EXAMPLE

Aravind Productions (AP) is a newly formed movie production house based out of Mumbai, India. AP was interested in understanding the production cost required for producing a Bollywood movie. The industry believes that the production house will require INR 500 million (50 crore) on average. It is assumed that the Bollywood movie production cost follows a normal distribution. The production costs of 40 Bollywood movies in millions of rupees are given in *bollywoodmovies.csv* file. Conduct an appropriate hypothesis test at $\alpha = 0.05$ to check whether the belief about average production cost is correct.

In this case, the population mean is 500 and the sample set for actual production cost is available in the file *bollywoodmovies.csv*. The population standard deviation is also not known. To compare if the average cost of the samples is equal to the 500 or not, we can conduct a one-sample *t*-test using *scipy.stats.ttest_1samp()*.

Using the following codes, read the data from the file *bollywoodmovies.csv* and display the first 5 records.

```
bollywood_movies_df = pd.read_csv('bollywoodmovies.csv')
bollywood_movies_df.head(5)
```

production_cost	
0	601
1	627
2	330
3	364
4	562

Now we print all the records from the dataset.

```
print(list(bollywood_movies_df.production_cost))

[601, 627, 330, 364, 562, 353, 583, 254, 528, 470, 125, 60,
101, 110, 60, 252, 281, 227, 484, 402, 408, 601, 593, 729,
402, 530, 708, 599, 439, 762, 292, 636, 444, 286, 636, 667,
252, 335, 457, 632]
```

3.2 EXAMPLE (Continued)

Null hypothesis is the *sample mean equals 500* and alternate hypothesis is the *sample mean does not equal 500*.

$$H_0: \mu = 500$$

$$H_A: \mu \neq 500$$

`scipy.stats.ttest_1samp()` method can be used for this test. It takes two parameters:

1. **a: array_like** – sample observation.
2. **popmean: float** – expected value in null hypothesis.

```
stats.ttest_1samp(bollywood_movies_df.production_cost, 500)
```

```
Ttest_1sampResult(statistic=-2.2845, pvalue=0.02786)
```

This returns -2.2845 value for t -statistics and 0.0278 for p -value. This implies the sample mean is less than (to the left of) population mean and has only 2.7% probability of being part of the distribution with a population mean of 500 . As p -value is less than 0.05 , we can conclude that the sample mean rejects that the production cost is equal to 500 .

3.9.3 | Two-Sample t-Test

A two-sample t -test is required to test difference between two population means where standard deviations are unknown. The parameters are estimated from the samples.

3.3 EXAMPLE

A company claims that children (in the age group between 7 and 12) who drink their (the company's) health drink will grow taller than the children who do not drink that health drink. Data in the file `healthdrink.xlsx` shows average increase in height over one-year period from two groups: one drinking the health drink and the other not drinking the health drink. At $\alpha = 0.05$, test whether the increase in height for the children who drink the health drink is different than those who do not drink health drink.

Solution: In this case, the population mean and standard deviation are not known. We have only two samples. One sample has observations of increase in height for the group drinking health drink and the other sample having observations of increase in height for the group not drinking health drink. We can read both the samples and conduct two-sample t -test using `scipy.stats.ttest_ind()`. If the method returns p -value less than 0.05 (α is given to be 0.05 in the question), then the increase in height can be concluded to be different.

The file `healthdrink.xlsx` contains two tabs `healthdrink_yes` and `healthdrink_no`, which include the respective samples.

3.3 EXAMPLE (Continued)

We first read the data using `pd.read_excel()`, which takes the filename `healthdrink.xlsx` and the tab `healthdrink_yes` as parameters. Then display first 5 records.

```
healthdrink_yes_df = pd.read_excel('healthdrink.xlsx',
'healthdrink_yes')
```

```
healthdrink_yes_df.head(5)
```

height_increase	
0	8.6
1	5.8
2	10.2
3	8.5
4	6.8

```
healthdrink_yes_df.columns
```

```
Index(['height_increase'], dtype='object')
```

Now read the data from the tab `healthdrink_no` in `healthdrink.xlsx` and display first 5 records.

```
healthdrink_no_df = pd.read_excel('healthdrink.xlsx',
'healthdrink_no') healthdrink_no_df.head(5)
```

height_increase	
0	5.3
1	9.0
2	5.7
3	5.5
4	5.4

We use the following code to display the distribution plots of increase in height separately for drinking health drink and not drinking health drink groups.

3.3 EXAMPLE (Continued)

```
sn.distplot(healthdrink_yes_df['height_increase'], label = 'healthdrink_yes') sn.distplot(healthdrink_no_df['height_increase'], label ='healthdrink_no') plt.legend();
```

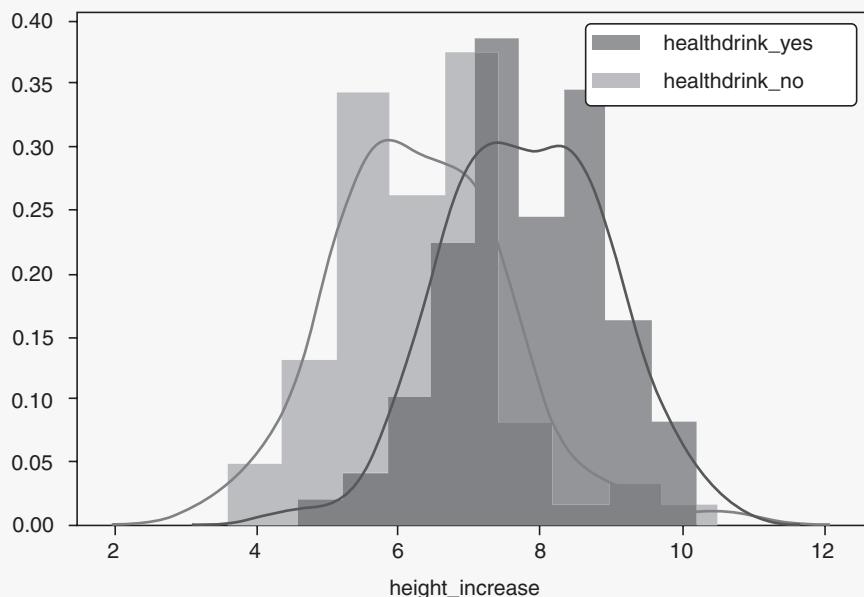


FIGURE 3.9 Comparing distributions for health drink and no health drink.

We can observe from Figure 3.9 that the distribution of increase in height for those who have the health drink has shifted to the right of those who did not have the health drink. But is the difference, as claimed, statistically significant?

The `scipy.stats.ttest_ind()` method takes two independent samples and returns the test statistics.

```
stats.ttest_ind(healthdrink_yes_df['height_increase'],  
                healthdrink_no_df['height_increase'])
```

```
Ttest_indResult(statistic=8.1316, pvalue=0.00)
```

The probability of the samples belonging to the same distribution is almost 0. This means that the increase in height for those who had health drink is significantly different than those who did not.

3.9.4 | Paired Sample t-Test

Sometimes we need to analyze whether an intervention (or treatment) such as an event, a training program, marketing promotions, treatment for specific illness, and lifestyle changes may have significantly changed the population parameter values such as mean before and after the intervention. The objective in this case is to check whether the difference in the parameter values is statistically significant before and after the intervention or between two different types of interventions. This is called a paired sample *t*-test and is used for comparing two different interventions applied on the same sample.

Consider the following example for a paired *t*-test.

3.4 EXAMPLE

The file *breakups.csv* contains alcohol consumption before and after breakup. Conduct a paired *t*-test to check whether the alcohol consumption is more after the breakup at 95% confidence ($\alpha = 0.05$).

Solution: Read the data onto the DataFrame and display first 5 records.

```
breakups_df = pd.read_csv('breakups.csv')
breakups_df.head(5)
```

	Before_Breakup	After_Breakup
0	470	408
1	354	439
2	496	321
3	351	437
4	349	335

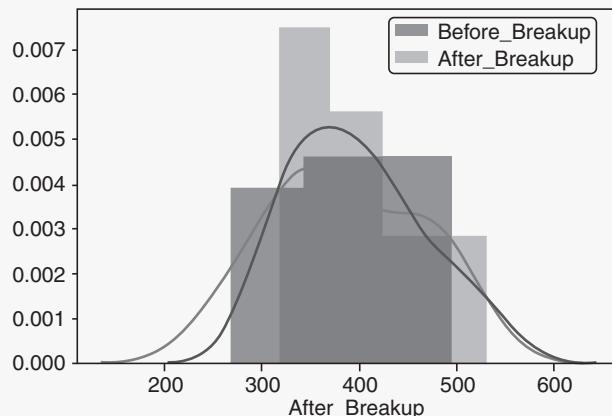
Use the following to display the distribution plots of alcohol consumption separately for before and after breakups.

```
sn.distplot(breakups_df['Before_Breakup'], label =
'Before_Breakup')
sn.distplot(breakups_df['After_Breakup'], label =
'After_Breakup')
plt.legend();
```

Figure 3.10 shows that the distribution of drinking habits before and after breakups are very similar. We can use paired *t*-test to confirm this. *scipy.stats.ttest_rel* is used to conduct the paired sample *t*-test. It takes both related samples as parameters.

```
stats.ttest_rel(breakups_df['Before_Breakup'], breakups_df
['After_Breakup'])

Ttest_relResult(statistic=-0.5375, pvalue=0.5971)
```

3.4 EXAMPLE (Continued)**FIGURE 3.10** Comparing distributions for drinking habits before and after breakups.

As the p -value is 0.597, which is more than 0.05 value, we conclude that they are part of same distribution. There is no change in alcohol consumption pattern before and after breakup.

3.9.5 | Chi-Square Goodness of Fit Test

Chi-square goodness of fit test is a non-parametric test used for comparing the observed distribution of data with the expected distribution of the data to decide whether there is any statistically significant difference between the observed distribution and a theoretical distribution. Chi-square statistics is given by

$$\chi^2 = \sum_{i=1}^n \frac{(O_i - E_i)^2}{E_i}$$

where O_i is the observed frequency and E_i is the expected frequency of the i^{th} category (interval of data or bin).

3.5 EXAMPLE

Hanuman Airlines (HA) operated daily flights to several Indian cities. One of the problems HA faces is the food preferences by the passengers. Captain Cook, the operations manager of HA, believes that 35% of their passengers prefer vegetarian food, 40% prefer non-vegetarian food, 20% low calorie food, and 5% request for diabetic food. A sample of 500 passengers was chosen to analyze the food preferences and the observed frequencies are as follows:

1. Vegetarian: 190
2. Non-vegetarian: 185

3.5 EXAMPLE (Continued)

3. Low calorie: 90
4. Diabetic: 35

Conduct a chi-square test to check whether Captain Cook's belief is true at $\alpha = 0.05$.

Solution: The `scipy.stats.chisquare` method is used for chi-square test. It takes the following parameters:

1. `f_obs : array_like` – Observed frequencies in each category.
2. `f_exp : array_like` – Expected frequencies in each category.

From the data we can create the following arrays:

```
## Observed frequencies
f_obs = [190, 185, 90, 35]
## Expected frequencies from the percentages expected
f_exp = [500*0.35, 500*0.4, 500*.2, 500*0.05]
print(f_exp)
```

```
[175.0, 200.0, 100.0, 25.0]
```

```
stats.chisquare(f_obs, f_exp)
```

```
Power_divergenceResult(statistic=7.4107, pvalue=0.0598)
```

As the p -value is more than 0.05, we retain the null hypothesis (original claim), that is, Captain Cook's belief is true.

3.10 | ANALYSIS OF VARIANCE (ANOVA)

Sometimes it may be necessary to conduct a hypothesis test to compare mean values simultaneously for more than two groups (samples) created using a factor (or factors). For example, a marketer may like to understand the impact of three different discount values (such as 0%, 10%, and 20% discount) on the average sales.

One-way ANOVA can be used to study the impact of a single treatment (also known as factor) at different levels (thus forming different groups) on a continuous response variable (or outcome variable).

Then the null and alternative hypotheses for one-way ANOVA for comparing 3 groups are given by

$$H_0: \mu_1 = \mu_2 = \mu_3$$

$$H_A: \text{Not all } \mu \text{ values are equal}$$

where μ_1, μ_2, μ_3 are mean values of each group.

Note that the alternative hypothesis, *not all μ values are equal*, implies that some of groups could be equal.

3.10.1 | Example of One-Way ANOVA

Ms Rachael Khanna the brand manager of ENZO detergent powder at the “one-stop” retail was interested in understanding whether the price discounts have any impact on the sales quantity of ENZO. To test whether the price discounts had any impact, price discounts of 0% (no discount), 10%, and 20% were given on randomly selected days. The quantity (in kilograms) of ENZO sold in a day under different discount levels is shown in Table 3.1. Conduct a one-way ANOVA to check whether discount had any significant impact on the average sales quantity at $\alpha = 0.05$.

TABLE 3.1 Sales for different discount levels

No Discount (0% discount)									
39	32	25	25	37	28	26	26	40	29
37	34	28	36	38	38	34	31	39	36
34	25	33	26	33	26	26	27	32	40
10% Discount									
34	41	45	39	38	33	35	41	47	34
47	44	46	38	42	33	37	45	38	44
38	35	34	34	37	39	34	34	36	41
20% Discount									
42	43	44	46	41	52	43	42	50	41
41	47	55	55	47	48	41	42	45	48
40	50	52	43	47	55	49	46	55	42

In this case, an ANOVA test can be conducted to test if there is any effect of different discounts on the average sales. The observations for No discount, 10% discount and 15% discount is given as three columns in the file *onestop.csv*.

Read the records from the file and print the first few records.

```
onestop_df = pd.read_csv('onestop.csv')
onestop_df.head(5)
```

	discount_0	discount_10	discount_20
0	39	34	42
1	32	41	43
2	25	45	44
3	25	39	46
4	37	38	41

Let us visualize the distribution of group using distribution plot (Figure 3.11).

```
sn.distplot(onestop_df['discount_0'], label = 'No Discount')
sn.distplot(onestop_df['discount_10'], label = '10% Discount')
sn.distplot(onestop_df['discount_20'], label = '20% Discount')
plt.legend();
```

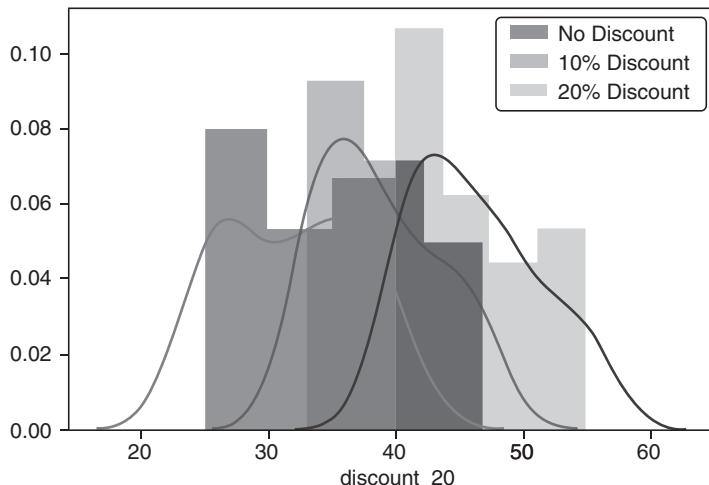


FIGURE 3.11 Comparing distributions of sales for different discount levels.

From the plot, it seems the distributions are not same. It may be by chance. Only a one-way ANOVA test will ensure if they are same or not. The `scipy.stats.f_oneway()` method conducts one-way ANOVA and returns F -statistics and p -value. If p -value is less than 0.05, the null hypothesis can be rejected and one can conclude that the mean of each group is not same.

```
from scipy.stats import f_oneway
f_oneway(onestop_df['discount_0'],
         onestop_df['discount_10'],
         onestop_df['discount_20'])
```

F_onewayResult(statistic=65.8698, pvalue=0.00)

As p -value is less than 0.05, we reject the null hypothesis and conclude that the mean sales quantity values under different discounts are different.

CONCLUSION

1. The chapter introduced the basic concepts of distributions, their parameters and how they can be applied in the real world to solve problems.
2. Some of the distributions we explored are normal distribution, exponential distribution, binomial, and Poisson distribution and their applications.

3. The objective of hypothesis testing is to either reject or retain an existing claim or belief with the help of newly collected evidences or samples. The existing belief is called a *null hypothesis* and a new claim is called an *alternative hypothesis* in a hypothesis test.
4. Z-test can be used to test the value of population mean when sample size is large ($n > 30$) and population variance is known.
5. The *t*-test is used when the population standard deviation is unknown and is estimated from the sample.
6. Chi-square tests are hypothesis tests that are used for comparing the observed distribution of data with expected distribution of the data to decide whether there is any statistically significant difference between the observed distribution and a theoretical distribution.

EXERCISES

Answer Questions 1 to 8 by Writing Code in Python.

1. The number of customer returns in a retail chain per day follows a Poisson distribution at a rate of 25 returns per day. Write Python code to answer the following questions:
 - (a) Calculate the probability that the number of returns exceeds 30 in a day.
 - (b) If the chance of fraudulent return is 0.05, calculate the probability that there will be at least 2 fraudulent returns in any given day.
2. A student is applying for Masters course in 8 US Universities and believes that she has in each of the eight universities a constant and independent 0.42 probability of getting selected. Write code to answer the following questions:
 - (a) What is the probability that she will get call from at least 3 universities?
 - (b) What is the probability that she will get calls from exactly 4 universities?
3. The time-of-failure of a machine follows exponential distribution with mean time between failures (MTBF) estimated to be 85 hrs. Write code to answer the following questions:
 - (a) Calculate the probability that the system will fail before 85 hrs.
 - (b) Calculate the probability that it will not fail up to 150 hrs.
4. As per a survey on use of pesticides among 1000 farmers in grape farming for around 10 acres of grape farmland, it was found that the grape farmers spray 38 liters of pesticides in a week on an average with the corresponding standard deviation of 5 liters. Assume that the pesticide spray per week follows a normal distribution. Write code to answer the following questions:
 - (a) What proportion of the farmers is spraying more than 50 liters of pesticide in a week?
 - (b) What proportion of farmers is spraying less than 10 liters?
 - (c) What proportion of farmers is spraying between 30 liters and 60 liters?
5. A bottle filling machine fills water into 5 liters (5000 cm^3) bottles. The company wants to test the null hypothesis that the average amount of water filled by the machine into the bottle is at least 5000 cm^3 . A random sample of 60 bottles coming out of the machine was selected and the exact contents of the selected bottles are recorded. The sample mean was $4,998.1 \text{ cm}^3$. The population standard deviation is known from the experience to be 1.30 cm^3 . Assume that the population is normally distributed with the standard deviation of 1.30 cm^3 . Write code to test the hypothesis at α of 5%. Explain the results.
6. A fabric manufacturer would like to understand the proportion of defective fabrics they produce. His shop floor staff have been stating that the percentage of defective is not more than 18%. He would like to test whether the claim made by his shop floor staff is correct. He picked up a random sample of 100 fabrics and found 22 defectives. Use $\alpha = 0.05$ and write code to test the hypothesis that the percentage of defective components is less than 18%.

7. Suppose that the makers of ABC batteries want to demonstrate that their battery lasts an average of at least 60 min longer than their competitor brand. Two independent random samples of 100 batteries of each kind are selected from both the brands, and the batteries are used continuously. The sample average life of ABC is found to be 450 min. The average life of competitor batteries is 368 min with the sample standard deviation of 82 min and 78 min, respectively. Frame a hypothesis and write the code to test ABC's claim at 95% significance.
8. A training institute would like to check the effectiveness of their training programs and if the scores of the people trained improve post the training. A sample of 30 students was taken and their scores were calculated before and after the training. File *trainingscores.csv* contains scores of the students before and after the training. Frame and write the code to test the hypothesis of training effectiveness for the training institute.

REFERENCES

1. Scikit-learn documentations at <http://scikit-learn.org/>
2. Statsmodel documentation at <https://www.statsmodels.org/stable/index.html>
3. Scipy documentation at <https://docs.scipy.org/doc/scipy/reference/>
4. Freier A (2016). "Women spend more Time using their Smartphones than Men", *Business of Apps*, June 6, 2016, available at <http://www.businessofapps.com/women-spend-more-time-using-their-smartphones-than-men/> (accessed on March 30, 2017).
5. Siegel E (2016). *Predictive Analytics: The Power to Predict who will Click, Buy, Lie or Die*, Wiley, New York.
6. Historical stock prices from <http://www.bseindia.com/>

CHAPTER

4

Linear Regression

LEARNING OBJECTIVES

After completing this chapter, you will be able to

- Understand the concept of simple and multiple linear regression and its applications in predictive analytics.
- Learn to build simple linear regression and multiple linear regression models using the Python package *statsmodel*.
- Learn to perform activities such as
 - (a) splitting the dataset into training and validation data.
 - (b) building regression models using Python package on training dataset and test on the validation dataset.
- Learn how to incorporate categorical (qualitative) variables in a regression model.
- Learn regression model diagnostics and perform tasks such as residual analysis, identification of outliers, and influential observations.

4.1 | SIMPLE LINEAR REGRESSION

Simple linear regression is a statistical technique used for finding the existence of an association relationship between a dependent variable (aka response variable or outcome variable) and an independent variable (aka explanatory variable, predictor variable or feature). We can only establish that change in the value of the outcome variable (Y) is associated with change in the value of feature X , that is, regression technique cannot be used for establishing causal relationship between two variables.

Regression is one of the most popular supervised learning algorithms in predictive analytics. A regression model requires the knowledge of both the outcome and the feature variables in the training dataset.

The following are a few examples of simple and multiple linear regression problems:

1. A hospital may be interested in finding how the total cost of a patient for a treatment varies with the body weight of the patient.
2. Insurance companies would like to understand the association between healthcare costs and ageing.
3. An organization may be interested in finding the relationship between revenue generated from a product and features such as the price, money spent on promotion, competitors' price, and promotion expenses.

4. Restaurants would like to know the relationship between the customer waiting time after placing the order and the revenue.
5. E-commerce companies such as Amazon, BigBasket, and Flipkart would like to understand the relationship between revenue and features such as
 - (a) Number of customer visits to their portal.
 - (b) Number of clicks on products.
 - (c) Number of items on sale.
 - (d) Average discount percentage.
6. Banks and other financial institutions would like to understand the impact of variables such as unemployment rate, marital status, balance in the bank account, rain fall, etc. on the percentage of non-performing assets (NPA).

4.2 | STEPS IN BUILDING A REGRESSION MODEL

In this section, we will explain the steps used in building a regression model. Building a regression model is an iterative process and several iterations may be required before finalizing the appropriate model.

STEP 1: Collect/Extract Data

The first step in building a regression model is to collect or extract data on the dependent (outcome) variable and independent (feature) variables from different data sources. Data collection in many cases can be time-consuming and expensive, even when the organization has well-designed enterprise resource planning (ERP) system.

STEP 2: Pre-Process the Data

Before the model is built, it is essential to ensure the quality of the data for issues such as reliability, completeness, usefulness, accuracy, missing data, and outliers.

1. Data imputation techniques may be used to deal with missing data. Use of descriptive statistics and visualization (such as box plot and scatter plot) may be used to identify the existence of outliers and variability in the dataset.
2. Many new variables (such as the ratio of variables or product of variables) can be derived (aka feature engineering) and also used in model building.
3. Categorical data must be pre-processed using dummy variables (part of feature engineering) before it is used in the regression model.

STEP 3: Dividing Data into Training and Validation Datasets

In this stage the data is divided into two subsets (sometimes more than two subsets): training dataset and validation or test dataset. The proportion of training dataset is usually between 70% and 80% of the data and the remaining data is treated as the validation data. The subsets may be created using random/stratified sampling procedure. This is an important step to measure the performance of the model using dataset not used in model building. It is also essential to check for any overfitting of the model. In many cases, multiple training and multiple test data are used (called cross-validation).

STEP 4: Perform Descriptive Analytics or Data Exploration

It is always a good practice to perform descriptive analytics before moving to building a predictive analytics model. Descriptive statistics will help us to understand the variability in the model and visualization of the data through, say, a box plot which will show if there are any outliers in the data. Another visualization technique, the scatter plot, may also reveal if there is any obvious relationship between the two variables under consideration. Scatter plot is useful to describe the functional relationship between the dependent or outcome variable and features.

STEP 5: Build the Model

The model is built using the training dataset to estimate the regression parameters. The method of Ordinary Least Squares (OLS) is used to estimate the regression parameters.

STEP 6: Perform Model Diagnostics

Regression is often misused since many times the modeler fails to perform necessary diagnostics tests before applying the model. Before it can be applied, it is necessary that the model created is validated for all model assumptions including the definition of the function form. If the model assumptions are violated, then the modeler must use remedial measure.

STEP 7: Validate the Model and Measure Model Accuracy

A major concern in analytics is over-fitting, that is, the model may perform very well on the training dataset, but may perform badly in validation dataset. It is important to ensure that the model performance is consistent on the validation dataset as is in the training dataset. In fact, the model may be cross-validated using multiple training and test datasets.

STEP 8: Decide on Model Deployment

The final step in the regression model is to develop a deployment strategy in the form of actionable items and business rules that can be used by the organization.

4.3 | BUILDING SIMPLE LINEAR REGRESSION MODEL

Simple Linear Regression (SLR) is a statistical model in which there is only one independent variable (or feature) and the functional relationship between the outcome variable and the regression coefficient is linear. Linear regression implies that the mathematical function is linear with respect to regression parameters.

One of the functional forms of SLR is as follows:

$$Y = \beta_0 + \beta_1 X + \varepsilon \quad (4.1)$$

For a dataset with n observations (X_i, Y_i) , where $i = 1, 2, \dots, n$, the above functional form can be written as follows:

$$Y_i = \beta_0 + \beta_1 X_i + \varepsilon_i \quad (4.2)$$

where Y_i is the value of i th observation of the dependent variable (outcome variable) in the sample, X_i is the value of i th observation of the independent variable or feature in the sample, ε_i is the random error (also known as residuals) in predicting the value of Y_i , β_0 and β_1 are the regression parameters (or regression coefficients or feature weights).

The regression relationship stated in Eq. (4.2) is a statistical relationship, and so is not exact, unlike a mathematical relationship, and thus the error terms ε_i . Equation (4.2) can be written as

$$\varepsilon_i = Y_i - \beta_0 - \beta_1 X_i \quad (4.3)$$

The regression parameters β_0 and β_1 are estimated by minimizing the sum of squared errors (SSE).

$$\text{SSE} = \sum_{i=1}^n \varepsilon_i^2 = \sum_{i=1}^n (Y_i - \beta_0 - \beta_1 X_i)^2 \quad (4.4)$$

The estimated values of regression parameters are given by taking partial derivative of SSE with respect to β_0 and β_1 and solving the resulting equations for the regression parameters. The estimated parameter values are given by

$$\hat{\beta}_1 = \sum_{i=1}^n \frac{(X_i - \bar{X})(Y_i - \bar{Y})}{(X_i - \bar{X})^2} \quad (4.5)$$

$$\hat{\beta}_0 = \bar{Y} - \hat{\beta}_1 \bar{X} \quad (4.6)$$

where $\hat{\beta}_0$ and $\hat{\beta}_1$ are the estimated values of the regression parameters β_0 and β_1 . The above procedure is known as method of ordinary least square (OLS). The estimate using OLS gives the best linear unbiased estimates (BLUE) of regression parameters.

Assumptions of the Linear Regression Model

1. The errors or residuals ε_i are assumed to follow a normal distribution with expected value of error $E(\varepsilon_i) = 0$.
2. The variance of error, $\text{VAR}(\varepsilon_i)$, is constant for various values of independent variable X . This is known as homoscedasticity. When the variance is not constant, it is called heteroscedasticity.
3. The error and independent variable are uncorrelated.
4. The functional relationship between the outcome variable and feature is correctly defined.

Properties of Simple Linear Regression

1. The mean value of Y_i for given X_i , $E(Y_i | X) = \hat{\beta}_0 + \hat{\beta}_1 X$.
2. Y_i follows a normal distribution with mean $\hat{\beta}_0 + \hat{\beta}_1 X$ and variance $\text{VAR}(\varepsilon_i)$.

Let us consider an example of predicting MBA Salary (outcome variable) from marks in Grade 10 (feature).

4.1

EXAMPLE Predicting MBA Salary from Grade in 10th Marks

Table 4.1 contains the salary of 50 graduating MBA students of a Business School in 2016 and their corresponding percentage marks in grade 10 (File name: *MBA_Salary.csv*). Develop an SLR model to understand and predict salary based on the percentage of marks in Grade 10.

TABLE 4.1 Salary of MBA students versus their grade 10 marks

S. No.	Percentage in Grade 10	Salary	S. No.	Percentage in Grade 10	Salary
1	62.00	270000	26	64.60	250000
2	76.33	200000	27	50.00	180000
3	72.00	240000	28	74.00	218000
4	60.00	250000	29	58.00	360000
5	61.00	180000	30	67.00	150000
6	55.00	300000	31	75.00	250000
7	70.00	260000	32	60.00	200000
8	68.00	235000	33	55.00	300000
9	82.80	425000	34	78.00	330000
10	59.00	240000	35	50.08	265000
11	58.00	250000	36	56.00	340000
12	60.00	180000	37	68.00	177600
13	66.00	428000	38	52.00	236000
14	83.00	450000	39	54.00	265000
15	68.00	300000	40	52.00	200000
16	37.33	240000	41	76.00	393000
17	79.00	252000	42	64.80	360000
18	68.40	280000	43	74.40	300000
19	70.00	231000	44	74.50	250000
20	59.00	224000	45	73.50	360000
21	63.00	120000	46	57.58	180000
22	50.00	260000	47	68.00	180000

(Continued)

4.1 EXAMPLE (Continued)**TABLE 4.1** (Continued)

S. No.	Percentage in Grade 10	Salary	S. No.	Percentage in Grade 10	Salary
23	69.00	300000	48	69.00	270000
24	52.00	120000	49	66.00	240000
25	49.00	120000	50	60.80	300000

Steps for building a regression model using Python:

1. Import *pandas* and *numpy* libraries
2. Use *read_csv* to load the dataset into DataFrame.
3. Identify the feature(s) (*X*) and outcome (*Y*) variable in the DataFrame for building the model.
4. Split the dataset into training and validation sets using *train_test_split()*.
5. Import *statsmodel* library and fit the model using *OLS()* method.
6. Print model summary and conduct model diagnostics.

For loading dataset into a DataFrame, we need to import *pandas* and *numpy* libraries.

```
import pandas as pd
import numpy as np

## Setting pandas print option to print decimal values upto
## 4 decimal places
np.set_printoptions(precision=4, linewidth=100)

mba_salary_df = pd.read_csv( 'MBA Salary.csv' )
mba_salary_df.head( 10 )
```

S. No.	Percentage in Grade 10	Salary
0	62.00	270000
1	76.33	200000
2	72.00	240000
3	60.00	250000
4	61.00	180000

(Continued)

4.1 EXAMPLE (Continued)

S. No.		Percentage in Grade 10	Salary
5	6	55.00	300000
6	7	70.00	260000
7	8	68.00	235000
8	9	82.80	425000
9	10	59.00	240000

We can print more information about the dataset using *Info()* method.

```
mba_salary_df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50 entries, 0 to 49
Data columns (total 3 columns):
S. No.          50 non-null int64
Percentage in Grade 10      50 non-null float64
Salary          50 non-null int64
dtypes: float64(1), int64(2)
memory usage: 1.2 KB
```

4.3.1 | Creating Feature Set (X) and Outcome Variable (Y)

The *statsmodel* library is used in Python for building statistical models. *OLS* API available in *statsmodel.api* is used for estimation of parameters for simple linear regression model. The *OLS()* model takes two parameters Y and X . In this example, *Percentage in Grade 10* will be X and *Salary* will be Y . *OLS* API available in *statsmodel.api* estimates only the coefficient of X parameter [refer to Eq. (4.1)]. To estimate regression coefficient β_0 , a constant term of 1 needs to be added as a separate column. As the value of the columns remains same across all samples, the parameter estimated for this feature or column will be the intercept term.

```
import statsmodels.api as sm
X = sm.add_constant(mba_salary_df['Percentage in Grade 10'])
X.head(5)
```

	Const	Percentage in Grade 10
0	1.0	62.00
1	1.0	76.33
2	1.0	72.00
3	1.0	60.00
4	1.0	61.00

```
Y = mba_salary_df['Salary']
```

4.3.2 | Splitting the Dataset into Training and Validation Sets

`train_test_split()` function from `sklearn.model_selection` module provides the ability to split the dataset randomly into training and validation datasets. The parameter `train_size` takes a fraction between 0 and 1 for specifying training set size. The remaining samples in the original set will be test or validation set. The records that are selected for training and test set are randomly sampled. The method takes a seed value in parameter named `random_state`, to fix which samples go to training and which ones go to test set.

`train_test_split()` The method returns four variables as below.

1. `train_X` contains X features of the training set.
2. `train_y` contains the values of response variable for the training set.
3. `test_X` contains X features of the test set.
4. `test_y` contains the values of response variable for the test set.

```
from sklearn.model_selection import train_test_split

train_X, test_X, train_y, test_y = train_test_split( X,
                                                    Y,
                                                    train_size = 0.8,
                                                    random_state = 100 )
```

`train_size = 0.8` implies 80% of the data is used for training the model and the remaining 20% is used for validating the model.

4.3.3 | Fitting the Model

We will fit the model using `OLS` method and pass `train_y` and `train_X` as parameters.

```
mba_salary_lm = sm.OLS( train_y, train_X ).fit()
```

The `fit()` method on `OLS()` estimates the parameters and returns model information to the variable `mba_salary_lm`, which contains the model parameters, accuracy measures, and residual values among other details.

4.3.3.1 Printing Estimated Parameters and Interpreting Them

```
print( mba_salary_lm.params )

Const                  30587.285652
Percentage in Grade 10    3560.587383
dtype: float64
```

The estimated (predicted) model can be written as

MBA Salary = 30587.285 + 3560.587 * (Percentage in Grade 10)

The equation can be interpreted as follows: For every 1% increase in Grade 10, the salary of the MBA students will increase by 3560.587.

4.3.3.2 Complete Code for Building Regression Model

The complete code for building the regression model is as follows:

```
# Importing all required libraries for building the regression model
import pandas as pd import numpy as np
import statsmodels.api as sm
from sklearn.model_selection import train_test_split

# Load the dataset into dataframe
mba_salary_df = pd.read_csv( 'MBA Salary.csv' )

# Add constant term of 1 to the dataset
X = sm.add_constant( mba_salary_df['Percentage in Grade 10'] )

# Split dataset into train and test set into 80:20 respectively
train_X, test_X, train_y, test_y = train_test_split( X,
                                                    Y,
                                                    train_size = 0.8,
                                                    random_state = 100 )

# Fit the regression model
mba_salary_lm = sm.OLS( train_y, train_X ).fit()

# Print the model parameters
print( mba_salary_lm.params )
```

4.4 | MODEL DIAGNOSTICS

It is important to validate the regression model to ensure its validity and goodness of fit before it can be used for practical applications. The following measures are used to validate the simple linear regression models:

1. Co-efficient of determination (R -squared).
2. Hypothesis test for the regression coefficient.
3. Analysis of variance for overall model validity (important for multiple linear regression).
4. Residual analysis to validate the regression model assumptions.
5. Outlier analysis, since the presence of outliers can significantly impact the regression parameters.

4.4.1 | Co-efficient of Determination (R -Squared or R^2)

The primary objective of regression is to explain the variation in Y using the knowledge of X . The co-efficient of determination (R -squared or R^2) measures the percentage of variation in Y explained by the model ($\beta_0 + \beta_1 X$). The simple linear regression model can be broken into

1. Variation in outcome variable explained by the model.
2. Unexplained variation as shown in Eq. (4.7):

$$\underbrace{Y_i}_{\text{Variation in } Y} = \underbrace{\beta_0 + \beta_1 X_i}_{\text{Variation in } Y \text{ explained by the model}} + \underbrace{\varepsilon_i}_{\text{Variation in } Y \text{ not explained by the model}} \quad (4.7)$$

It can be proven mathematically that

$$\underbrace{\sum_{i=1}^n (Y_i - \bar{Y})^2}_{SST} = \underbrace{\sum_{i=1}^n (\hat{Y}_i - \bar{Y})^2}_{SSR} + \underbrace{\sum_{i=1}^n (Y_i - \hat{Y}_i)^2}_{SSE} \quad (4.8)$$

where $\hat{Y}_i = \hat{\beta}_0 + \hat{\beta}_1 X_i$ is the predicted value of Y_i . The hat ($\hat{\cdot}$) symbol is used for denoting the estimated value. SST is the sum of squares of total variation, SSR is the sum of squares of explained variation due to the regression model, and SSE is the sum of squares of unexplained variation (error). The co-efficient of determination (R -squared) is given by

$$R\text{-Squared} = \frac{SSR}{SST} = 1 - \frac{SSE}{SST} \quad (4.9)$$

The co-efficient of determination (R -squared) has the following properties:

1. The value of R -squared lies between 0 and 1.
2. Mathematically, R -squared (R^2) is square of correlation coefficient ($R^2 = r^2$), where r is the Pearson correlation co-efficient.
3. Higher R -squared indicates better fit; however, one should be careful about the spurious relationship.

4.4.2 | Hypothesis Test for the Regression Co-efficient

The regression co-efficient (β_1) captures the existence of a linear relationship between the outcome variable and the feature. If $\beta_1 = 0$, we can conclude that there is no statistically significant linear relationship between the two variables. It can be proved that the sampling distribution of β_1 is a t -distribution (Kutner et al., 2013; U Dinesh Kumar, 2017). The null and alternative hypotheses are

$$H_0: \beta_1 = 0$$

$$H_A: \beta_1 \neq 0$$

The corresponding test statistic is given by

$$t_{\alpha/2, n-2} = \frac{\hat{\beta}_1}{S_e(\hat{\beta}_1)} \quad (4.10)$$

The standard error of estimate of the regression co-efficient is given by

$$S_e(\hat{\beta}_1) = \sqrt{\frac{S_e}{\sum(X_i - \bar{X})^2}} \quad (4.11)$$

where S_e is the standard error of the estimated value of Y_i (and the residuals) and is given by

$$S_e = \sqrt{\frac{\sum(Y_i - \hat{Y}_i)^2}{n-2}} \quad (4.12)$$

The hypothesis test is a two-tailed test. The test statistic in Eq. (4.10) is a t -distribution with $n - 2$ degrees of freedom (two degrees of freedom are lost due to the estimation of two regression parameters β_0 and β_1). $S_e(\widehat{\beta}_1)$ is the standard error of regression co-efficient $\widehat{\beta}_1$.

4.4.3 | Analysis of Variance (ANOVA) in Regression Analysis

We can check the overall validity of the regression model using ANOVA in the case of multiple linear regression model with k features. The null and alternative hypotheses are given by

$$H_0: \beta_1 = \beta_2 = \dots = \beta_k = 0$$

$$H_A: \text{Not all regression coefficients are zero}$$

The corresponding F -statistic is given by

$$F = \frac{\text{MSR}}{\text{MSE}} = \frac{\text{SSR} / k}{\text{SSE} / (n - k - 1)} \quad (4.13)$$

where MSR (= SSR/k) and MSE [= SSE/(n - k - 1)] are mean squared regression and mean squared error, respectively. F -test is used for checking whether the overall regression model is statistically significant or not.

4.4.4 | Regression Model Summary Using Python

The function `summary2()` prints the model summary which contains the information required for diagnosing a regression model (Table 4.2).

```
mba_salary_lm.summary2()
```

TABLE 4.2 Model summary: Simple linear regression model

Model:	OLS	Adj. R-squared:	0.190
Dependent Variable:	Salary	AIC:	1008.8680
Date:	2018-04-08 07:27	BIC:	1012.2458
No. Observations:	40	Log-Likelihood:	-502.43
Df Model:	1	F-statistic:	10.16
Df Residuals:	38	Prob (F-statistic):	0.00287
R-squared:	0.211	Scale:	5.0121e+09

	Coef.	Std.Err.	t	P > t	[0.025	0.975]
const	30587.2857	71869.4497	0.4256	0.6728	-114904.8089	176079.3802
Percentage in Grade 10	3560.5874	1116.9258	3.1878	0.0029	1299.4892	5821.6855

Omnibus:	2.048	Durbin-Watson:	2.611
Prob(Omnibus):	0.359	Jarque-Bera (JB):	1.724
Skew:	0.369	Prob(JB):	0.422
Kurtosis:	2.300	Condition No.:	413

From the summary output shown in Table 4.2, we can infer the following:

1. The model R -squared value is 0.211, that is, the model explains 21.1% of the variation in salary.
2. The p -value for the t -test is 0.0029 which indicates that there is a statistically significant relationship (at significance value $\alpha = 0.05$) between the feature, percentage in grade 10, and salary. Also, the probability value of F -statistic of the model is 0.0029 which indicates that the overall model is statistically significant. Note that, in a simple linear regression, the p -value for t -test and F -test will be the same since the null hypothesis is the same. (Also $F = t^2$ in the case of SLR.)

4.4.5 | Residual Analysis

Residuals or errors are the difference between the actual value of the outcome variable and the predicted value ($Y_i - \hat{Y}_i$). Residual (error) analysis is important to check whether the assumptions of regression models have been satisfied. It is performed to check the following:

1. The residuals are normally distributed.
2. Variance of residual is constant (homoscedasticity).
3. The functional form of regression is correctly specified.
4. There are no outliers.

4.4.5.1 Check for Normal Distribution of Residual

The normality of residuals can be checked using the **probability–probability plot (P-P plot)**. P-P plot compares the cumulative distribution function of two probability distributions against each other. In the current context, we use the P-P plot to check whether the distribution of the residuals matches with that of a normal distribution. In Python, *ProbPlot()* method on *statsmodel* draws the P-P plot as shown in Figure 4.1.

```
import matplotlib.pyplot as plt
import seaborn as sn
%matplotlib inline

mba_salary_resid = mba_salary_lm.resid
probplot = sm.ProbPlot(mba_salary_resid)
plt.figure( figsize = (8, 6))
probplot.ppplot( line='45' )
plt.title( "Fig 4.1 - Normal P-P Plot of Regression Standardized
Residuals" )
plt.show()
```

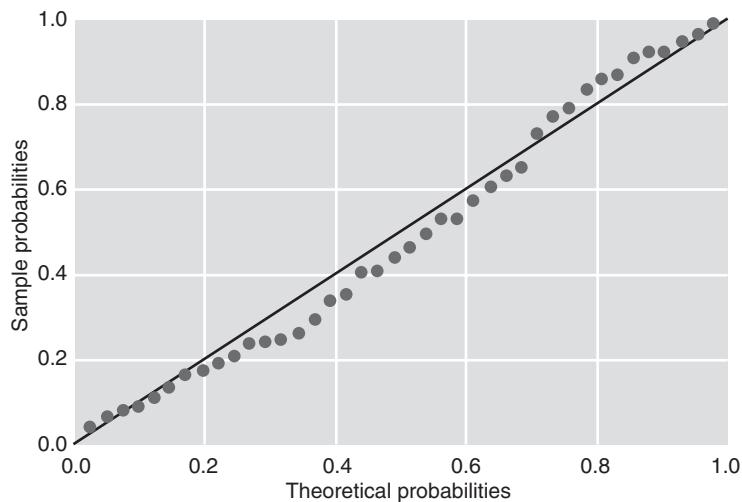


FIGURE 4.1 Normal P-P plot of regression standardized residuals.

In Figure 4.1, the diagonal line is the cumulative distribution of a normal distribution, whereas the dots represent the cumulative distribution of the residuals. Since the dots are close to the diagonal line, we can conclude that the residuals follow an approximate normal distribution (we need only an approximate normal distribution).

4.4.5.2 Test of Homoscedasticity

An important assumption of the regression model is that the residuals have constant variance (homoscedasticity) across different values of the predicted value (Y). The homoscedasticity can be observed by drawing a residual plot, which is a plot between standardized residual value and standardized predicted value. If there is heteroscedasticity (non-constant variance of residuals), then a funnel type shape in the residual plot can be expected. A non-constant variance of the residuals is known as heteroscedasticity.

The following custom method `get_standardized_values()` creates the standardized values of a series of values (variable). It subtracts values from mean and divides by standard deviation of the variable.

```
def get_standardized_values( vals ):
    return (vals - vals.mean()) / vals.std()
```

```
plt.scatter( get_standardized_values( mba_salary_lm.fittedvalues ),
            get_standardized_values( mba_salary_resid ) )
plt.title( "Fig 4.2 - Residual Plot: MBA Salary Prediction" );
plt.xlabel( "Standardized predicted values" )
plt.ylabel( "Standardized Residuals" );
```

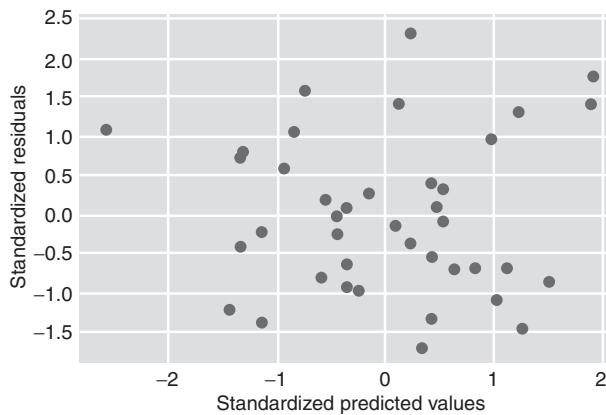


FIGURE 4.2 Residual plot: MBA salary prediction.

It can be observed in Figure 4.2 that the residuals are random and have no funnel shape, which means the residuals have constant variance (homoscedasticity).

4.4.6 | Outlier Analysis

Outliers are observations whose values show a large deviation from the mean value. Presence of an outlier can have a significant influence on the values of regression coefficients. Thus, it is important to identify the existence of outliers in the data.

The following distance measures are useful in identifying influential observations:

1. Z-Score
2. Mahalanobis Distance
3. Cook's Distance
4. Leverage Values

4.4.6.1 Z-Score

Z-score is the standardized distance of an observation from its mean value. For the predicted value of the dependent variable Y , the Z-score is given by

$$Z = \frac{Y_i - \bar{Y}}{\sigma_Y} \quad (4.14)$$

where Y_i is the predicted value of Y for i th observation, \bar{Y} is the mean or expected value of Y , σ_Y is the variance of Y .

Any observation with a Z-score of more than 3 may be flagged as an outlier. The Z-score in the data can be obtained using the following code:

```
from scipy.stats import zscore
```

```
mba_salary_df['z_score_salary'] = zscore(mba_salary_df['Salary'])
```

```
mba_salary_df[ (mba_salary_df.z_score_salary > 3.0) | (mba_salary_df.z_score_salary < -3.0) ]
```

S. No.	Percentage in Grade 10	Salary	z_score_salary
--------	------------------------	--------	----------------

So, there are no observations that are outliers as per the Z-score.

4.4.6.2 Cook's Distance

Cook's distance measures how much the predicted value of the dependent variable changes for all the observations in the sample when a particular observation is excluded from the sample for the estimation of regression parameters.

A Cook's distance value of more than 1 indicates highly influential observation. Python code for calculating Cook's distance is provided below. In this `get_influence()` returns the influence of observations in the model and `cook_distance` variable provides Cook's distance measures. Then the distances can be plotted against the observation index to find out which observations are influential.

```
import numpy as np

mba_influence = mba_salary_lm.get_influence()
(c, p) = mba_influence.cooks_distance

plt.stem(np.arange( len( train_X ) ),
         np.round( c, 3 ),
         markerfmt="," );
plt.title( "Figure 4.3 - Cooks distance for all observations in MBA Salary data set" );
plt.xlabel("Row index")
plt.ylabel("Cooks Distance");
```

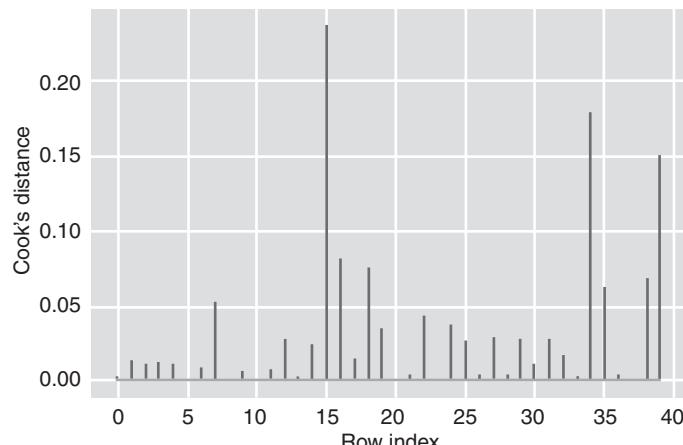


FIGURE 4.3 Cook's distance for all observations in MBA salary dataset.

From Figure 4.3, it can be observed that none of the observations' Cook's distance exceed 1 and hence none of them are outliers.

4.4.6.3 Leverage Values

Leverage value of an observation measures the influence of that observation on the overall fit of the regression function and is related to the Mahalanobis distance. Leverage value of more than $3(k + 1)/n$ is treated as highly influential observation, where k is the number of features in the model and n is the sample size.

statsmodels.graphics.regressionplots module provides *influence_plot()* which draws a plot between standardized residuals and leverage value. Mostly, the observations with high leverage value (as mentioned above) and high residuals [more than value $3(k + 1)/n$] can be removed from the training dataset.

```
from statsmodels.graphics.regressionplots import influence_plot
fig, ax = plt.subplots( figsize=(8, 6) )
influence_plot( mba_salary_lm, ax = ax )
plt.title("Figure 4.4 - Leverage Value Vs Residuals")
plt.show();
```

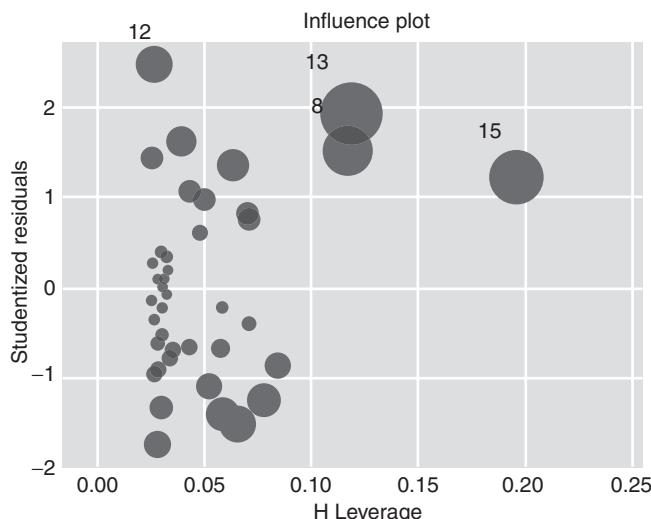


FIGURE 4.4 Leverage value versus residuals.

In Figure 4.4, the size of the circle is proportional to the product of residual and leverage value. The larger the circle, the larger is the residual and hence influence of the observation.

4.4.7 | Making Prediction and Measuring Accuracy

Ideally, the prediction should be made on the validation (or test) data and the accuracy of prediction should be evaluated.

4.4.7.1 Predicting using the Validation Set

The model variable has a method `predict()`, which takes the X parameters and returns the predicted values.

```
pred_y = mba_salary_lm.predict( test_X )
```

`pred_y` variable contains the predicted value. We can compare the predicted values with the actual values and calculate the accuracy in the next section.

4.4.7.2 Finding R-Squared and RMSE

Several measures can be used for measuring the accuracy of prediction. **Mean Square Error (MSE)**, **Root Mean Square Error (RMSE)** and **Mean Absolute Percentage Error (MAPE)** are some of the frequently used measures. `sklearn.metrics` has `r2_score` and `mean_squared_error` for measuring R -squared and MSE values. We need to take the square root of the MSE value to get RMSE value. Both the methods take predicted Y values and actual Y values to calculate the accuracy measures. `Numpy` module has `sqrt` method to calculate the square root of a value.

```
from sklearn.metrics import r2_score, mean_squared_error
```

```
np.abs(r2_score(test_y, pred_y))
```

```
0.15664584974230378
```

So, the model only explains 15.6% of the variance in the validation set.

```
import numpy
```

```
np.sqrt(mean_squared_error(test_y, pred_y))
```

```
73458.043483468937
```

RMSE means the average error the model makes in predicting the outcome. The smaller the value of RMSE, the better the model is.

4.4.7.3 Calculating Prediction Intervals

The regression equation gives us the point estimate of the outcome variable for a given value of the independent variable. In many applications, we would be interested in knowing the interval estimate of Y_i for a given value of explanatory variable. `wls_prediction_std()` returns the prediction interval while making a prediction. It takes significance value (α) to calculate the interval. An α -value of 0.1 returns the prediction at confidence interval of 90%. The code for calculating prediction interval is as follows:

```
from statsmodels.sandbox.regression.predstd import wls_prediction_std
# Predict the y values
pred_y = mba_salary_lm.predict( test_X )
```

```
# Predict the low and high interval values for y
_, pred_y_low, pred_y_high = wls_prediction_std(mba_salary_lm,
                                                test_X,
                                                alpha = 0.1)

# Store all the values in a dataframe
pred_y_df = pd.DataFrame( { 'grade_10_perc': test_X['Percentage in Grade 10'],
                            'pred_y': pred_y,
                            'pred_y_left': pred_y_low,
                            'pred_y_right': pred_y_high } )
```

```
pred_y_df[0:10]
```

	grade_10_perc	pred_y	pred_y_left	pred_y_right
6	70.0	279828.402452	158379.832044	401276.972860
36	68.0	272707.227686	151576.715020	393837.740352
37	52.0	215737.829560	92950.942395	338524.716726
28	58.0	237101.353858	115806.869618	358395.838097
43	74.5	295851.045675	173266.083342	418436.008008
49	60.8	247070.998530	126117.560983	368024.436076
5	55.0	226419.591709	104507.444388	348331.739030
33	78.0	308313.101515	184450.060488	432176.142542
20	63.0	254904.290772	134057.999258	375750.582286
42	74.4	295494.986937	172941.528691	418048.445182

4.5 | MULTIPLE LINEAR REGRESSION

Multiple linear regression (MLR) is a supervised learning algorithm for finding the existence of an association relationship between a dependent variable (aka response variable or outcome variable) and several independent variables (aka explanatory variables or predictor variable or features).

The functional form of MLR is given by

$$Y_i = \beta_0 + \beta_1 X_{1i} + \beta_2 X_{2i} + \dots + \beta_k X_{ki} + \varepsilon_i$$

The regression coefficients $\beta_1, \beta_2, \dots, \beta_k$ are called partial regression coefficients since the relationship between an explanatory variable and the response (outcome) variable is calculated after removing (or controlling) the effect all the other explanatory variables (features) in the model.

The assumptions that are made in multiple linear regression model are as follows:

1. The regression model is linear in regression parameters (β -values).
2. The residuals follow a normal distribution and the expected value (mean) of the residuals is zero.
3. In time series data, residuals are assumed to uncorrelated.

4. The variance of the residuals is constant for all values of X_i . When the variance of the residuals is constant for different values of X_i , it is called homoscedasticity. A non-constant variance of residuals is called heteroscedasticity.
5. There is no high correlation between independent variables in the model (called multi-collinearity). Multi-collinearity can destabilize the model and can result in an incorrect estimation of the regression parameters.

The partial regressions coefficients are estimated by minimizing the sum of squared errors (SSE). We will explain the multiple linear regression model by using the example of auction pricing of players in the Indian premier league (IPL).

4.5.1 | Predicting the SOLD PRICE (Auction Price) of Players

The Indian Premier League (IPL) is a professional league for Twenty20 (T20) cricket championships that was started in 2008 in India. IPL was initiated by the BCCI with eight franchises comprising players from across the world. The first IPL auction was held in 2008 for ownership of the teams for 10 years, with a base price of USD 50 million. The franchises acquire players through an English auction that is conducted every year. However, there are several rules imposed by the IPL. For example, only international players and popular Indian players are auctioned.

The performance of the players could be measured through several metrics. Although the IPL follows the Twenty20 format of the game, it is possible that the performance of the players in the other formats of the game such as Test and One-Day matches could influence player pricing. A few players had excellent records in Test matches, but their records in Twenty20 matches were not very impressive. The performances of 130 players who played in at least one season of the IPL (2008–2011) measured through various performance metrics are provided in Table 4.3.

TABLE 4.3 Metadata of IPL dataset

Data Code	Description
AGE	Age of the player at the time of auction classified into three categories. Category 1 (L25) means the player is less than 25 years old, category 2 means that the age is between 25 and 35 years (B25–35) and category 3 means that the age is more than 35 (A35).
RUNS-S	Number of runs scored by a player.
RUNS-C	Number of runs conceded by a player.
HS	Highest score by a batsman in IPL.
AVE-B	Average runs scored by a batsman in IPL.
AVE-BL	Bowling average (number of runs conceded/number of wickets taken) in IPL.
SR-B	Batting strike rate (ratio of the number of runs scored to the number of balls faced) in IPL.
SR-BL	Bowling strike rate (ratio of the number of balls bowled to the number of wickets taken) in IPL.
SIXERS	Number of six runs scored by a player in IPL.
WKTS	Number of wickets taken by a player in IPL.

(Continued)

TABLE 4.3 (Continued)

Data Code	Description
ECON	Economy rate of a bowler (number of runs conceded by the bowler per over) in IPL.
CAPTAINCY EXP	Captained either a T20 team or a national team.
ODI-SR-B	Batting strike rate in One-Day Internationals.
ODI-SR-BL	Bowling strike rate in One-Day Internationals.
ODI-RUNS-S	Runs scored in One-Day Internationals.
ODI-WKTS	Wickets taken in One-Day Internationals.
T-RUNS-S	Runs scored in Test matches.
T-WKTS	Wickets taken in Test matches.
PLAYER-SKILL	Player's primary skill (batsman, bowler, or allrounder).
COUNTRY	Country of origin of the player (AUS: Australia; IND: India; PAK: Pakistan; SA: South Africa; SL: Sri Lanka; NZ: New Zealand; WI: West Indies; OTH: Other countries).
YEAR-A	Year of Auction in IPL.
IPL TEAM	Team(s) for which the player had played in the IPL (CSK: Chennai Super Kings; DC: Deccan Chargers; DD: Delhi Dare-devils; KXJ: Kings XI Punjab; KKR: Kolkata Knight Riders; MI: Mumbai Indians; PWI: Pune Warriors India; RR: Rajasthan Royals; RCB: Royal Challengers Bangalore). A + sign is used to indicate that the player has played for more than one team. For example, CSK+ would mean that the player has played for CSK as well as for one or more other teams.

4.5.2 | Developing Multiple Linear Regression Model Using Python

In this section, we will be discussing various steps involved in developing a multiple linear regression model using Python.

4.5.2.1 Loading the Dataset

Loading data from *IPL IMB381IPL2013.csv* the file and print the meta data.

```
ipl_auction_df = pd.read_csv('IPL IMB381IPL2013.csv')
```

```
ipl_auction_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 130 entries, 0 to 129
Data columns (total 26 columns):
Sl.NO.          130    non-null    int64
PLAYER NAME     130    non-null    object
AGE             130    non-null    int64
COUNTRY         130    non-null    object
TEAM            130    non-null    object
```

```

PLAYING ROLE      130    non-null   object
T-RUNS            130    non-null   int64
T-WKTS            130    non-null   int64
ODI-RUNS-S        130    non-null   int64
ODI-SR-B          130    non-null   float64
ODI-WKTS          130    non-null   int64
ODI-SR-BL         130    non-null   float64
CAPTAINCY EXP     130    non-null   int64
RUNS-S             130    non-null   int64
HS                 130    non-null   int64
AVE                130    non-null   float64
SR-B               130    non-null   float64
SIXERS             130    non-null   int64
RUNS-C             130    non-null   int64
WKTS               130    non-null   int64
AVE-BL             130    non-null   float64
ECON               130    non-null   float64
SR-BL              130    non-null   float64
AUCTION YEAR       130    non-null   int64
BASE PRICE          130    non-null   int64
SOLD PRICE          130    non-null   int64
dtypes: float64(7), int64(15), object(4)
memory usage: 26.5+ KB

```

There are 130 observations (records) and 26 columns (features) in the data, and there are no missing values.

4.5.2.2 Displaying the First Five Records

As the number of columns is very large, we will display the initial 10 columns for the first 5 rows. The function `df.iloc()` is used for displaying a subset of the dataset.

```
ipl_auction_df.iloc[0:5, 0:10]
```

SI.NO.	PLAYER NAME	AGE	COUNTRY	TEAM	PLAYING ROLE	T-RUNS	T-WKTS	ODI-RUNS-S	ODI-SR-B
0	Abdulla, YA	2	SA	KXIP	Allrounder	0	0	0	0.00
1	Abdur Razzak	2	BAN	RCB	Bowler	214	18	657	71.41
2	Agarkar, AB	2	IND	KKR	Bowler	571	58	1269	80.62
3	Ashwin, R	1	IND	CSK	Bowler	284	31	241	84.56
4	Badrinath, S	2	IND	CSK	Batsman	63	0	79	45.93

We can build a model to understand what features of players are influencing their `SOLD PRICE` or predict the player's auction prices in future. However, all columns are not features. For example, `Sl. NO.`

is just a serial number and cannot be considered a feature of the player. We will build a model using only player's statistics. So, *BASE PRICE* can also be removed. We will create a variable *X_feature* which will contain the list of features that we will finally use for building the model and ignore rest of the columns of the DataFrame. The following function is used for including the features in the model building.

```
X_features = ipl_auction_df.columns
```

Most of the features in the dataset are numerical (ratio scale) whereas features such as *AGE*, *COUNTRY*, *PLAYING ROLE*, *CAPTAINCY EXP* are categorical and hence need to be encoded before building the model. Categorical variables cannot be directly included in the regression model, and they must be encoded using dummy variables before incorporating in the model building.

```
X_features = ['AGE', 'COUNTRY', 'PLAYING ROLE',
               'T-RUNS', 'T-WKTS', 'ODI-RUNS-S', 'ODI-SR-B',
               'ODI-WKTS', 'ODI-SR-BL', 'CAPTAINCY EXP', 'RUNS-S',
               'HS', 'AVE', 'SR-B', 'SIXERS', 'RUNS-C', 'WKTS',
               'AVE-BL', 'ECON', 'SR-BL']
```

4.5.3 | Encoding Categorical Features

Qualitative variables or categorical variables need to be encoded using dummy variables before incorporating them in the regression model. If a categorical variable has n categories (e.g., the player role in the data has four categories, namely, batsman, bowler, wicket-keeper and allrounder), then we will need $n - 1$ dummy variables. So, in the case of *PLAYING ROLE*, we will need three dummy variables since there are four categories.

Finding unique values of column *PLAYING ROLE* shows the values: *Allrounder*, *Bowler*, *Batsman*, *W. Keeper*. The following Python code is used to encode a categorical or qualitative variable using dummy variables:

```
ipl_auction_df['PLAYING ROLE'].unique()
```

```
array(['Allrounder', 'Bowler', 'Batsman', 'W. Keeper'], dtype=object)
```

The variable can be converted into four dummy variables. Set the variable value to 1 to indicate the role of the player. This can be done using *pd.get_dummies()* method. We will create dummy variables for only *PLAYING ROLE* to understand and then create dummy variables for the rest of the categorical variables.

```
pd.get_dummies(ipl_auction_df['PLAYING ROLE']) [0:5]
```

	Allrounder	Batsman	Bowler	W. Keeper
0	1.0	0.0	0.0	0.0
1	0.0	0.0	1.0	0.0

(Continued)

	Allrounder	Batsman	Bowler	W. Keeper
2	0.0	0.0	1.0	0.0
3	0.0	0.0	1.0	0.0
4	0.0	1.0	0.0	0.0

As shown in the table above, the `pd.get_dummies()` method has created four dummy variables and has already set the variables to 1 as variable value in each sample.

Whenever we have n levels (or categories) for a qualitative variable (categorical variable), we will use $(n - 1)$ dummy variables, where each dummy variable is a binary variable used for representing whether an observation belongs to a category or not. The reason why we create only $(n - 1)$ dummy variables is that inclusion of dummy variables for all categories and the constant in the regression equation will create perfect multi-collinearity (will be discussed later). To drop one category, the parameter `drop_first` should be set to `True`.

We must create dummy variables for all categorical (qualitative) variables present in the dataset.

```
categorical_features = ['AGE', 'COUNTRY', 'PLAYING ROLE',
                        'CAPTAINCY EXP']
```

```
ipl_auction_encoded_df = pd.get_dummies(ipl_auction_df[X_features],
                                         columns = categorical_features,
                                         drop_first = True)
```

```
ipl_auction_encoded_df.columns
```

```
Index(['T-RUNS', 'T-WKTS', 'ODI-RUNS-S', 'ODI-SR-B', 'ODI-WKTS',
       'ODI-SR-BL', 'RUNS-S', 'HS', 'AVE', 'SR-B', 'SIXERS',
       'RUNS-C', 'WKTS', 'AVE-BL', 'ECON', 'SR-BL', 'AGE_2',
       'AGE_3', 'COUNTRY_BAN', 'COUNTRY_ENG', 'COUNTRY_IND',
       'COUNTRY_NZ', 'COUNTRY_PAK', 'COUNTRY_SA', 'COUNTRY_SL',
       'COUNTRY_WI', 'COUNTRY_ZIM', 'PLAYING ROLE_Batsman',
       'PLAYING ROLE_Bowler', 'PLAYING ROLE_W. Keeper',
       'CAPTAINCY EXP_1'], dtype='object')
```

The dataset contains the new dummy variables that have been created. We can reassign the new features to the variable `X_features`, which we created earlier to keep track of all features that will be used to build the model finally.

```
X_features = ipl_auction_encoded_df.columns
```

4.5.4 | Splitting the Dataset into Train and Validation Sets

Before building the model, we will split the dataset into 80:20 ratio. The `split` function allows using a parameter `random_state`, which is a seed function for reproducibility of randomness. This parameter is not required to be passed. Setting this variable to a fixed number will make sure that the records that go

into training and test set remain unchanged and hence the results can be reproduced. We will use the value 42 (it is again selected randomly). You can use the same random seed of 42 for the reproducibility of results obtained here. Using another random seed may give different training and test data and hence different results.

```
X = sm.add_constant( ipl_auction_encoded_df )
Y = ipl_auction_df['SOLD PRICE']

train_X, test_X, train_y, test_y = train_test_split(X ,
                                                    Y,
                                                    train_size = 0.8,
                                                    random_state = 42 )
```

4.5.5 | Building the Model on the Training Dataset

We will build the MLR model using the training dataset and analyze the model summary. The summary provides details of the model accuracy, feature significance, and signs of any multi-collinearity effect, which is discussed in detail in the next section.

```
ipl_model_1 = sm.OLS(train_y, train_X).fit()
ipl_model_1.summary2()
```

TABLE 4.4 Model summary for *ipl_model_1*

Model:	OLS	Adj. R-squared:	0.362
Dependent Variable:	SOLD PRICE	AIC:	2965.2841
Date:	2018-04-08 07:27	BIC:	3049.9046
No. Observations:	104	Log-Likelihood:	-1450.6
Df Model:	31	F-statistic:	2.883
Df Residuals:	72	Prob (F-statistic):	0.000114
R-squared:	0.554	Scale:	1.1034e+11

	Coef.	Std.Err.	t	P> t	[0.025	0.975]
const	375827.1991	228849.9306	1.6422	0.1049	-80376.7996	832031.1978
T-RUNS	-53.7890	32.7172	-1.6441	0.1045	-119.0096	11.4316
T-WKTS	-132.5967	609.7525	-0.2175	0.8285	-1348.1162	1082.9228
ODI-RUNS-S	57.9600	31.5071	1.8396	0.0700	-4.8482	120.7681
ODI-SR-B	-524.1450	1576.6368	-0.3324	0.7405	-3667.1130	2618.8231
ODI-WKTS	815.3944	832.3883	0.9796	0.3306	-843.9413	2474.7301
ODI-SR-BL	-773.3092	1536.3334	-0.5033	0.6163	-3835.9338	2289.3154
RUNS-S	114.7205	173.3088	0.6619	0.5101	-230.7643	460.2054
HS	-5516.3354	2586.3277	-2.1329	0.0363	-10672.0855	-360.5853
AVE	21560.2760	7774.2419	2.7733	0.0071	6062.6080	37057.9439

(Continued)

	Coef.	Std.Err.	t	P > t	[0.025	0.975]
SR-B	-1324.7218	1373.1303	-0.9647	0.3379	-4062.0071	1412.5635
SIXERS	4264.1001	4089.6000	1.0427	0.3006	-3888.3685	12416.5687
RUNS-C	69.8250	297.6697	0.2346	0.8152	-523.5687	663.2187
WKTS	3075.2422	7262.4452	0.4234	0.6732	-11402.1778	17552.6622
AVE-BL	5182.9335	10230.1581	0.5066	0.6140	-15210.5140	25576.3810
ECON	-6820.7781	13109.3693	-0.5203	0.6045	-32953.8282	19312.2721
SR-BL	-7658.8094	14041.8735	-0.5454	0.5871	-35650.7726	20333.1539
AGE_2	-230767.6463	114117.2005	-2.0222	0.0469	-458256.1279	-3279.1648
AGE_3	-216827.0808	152246.6232	-1.4242	0.1587	-520325.1772	86671.0155
COUNTRY_BAN	-122103.5196	438719.2796	-0.2783	0.7816	-996674.4194	752467.3801
COUNTRY_ENG	672410.7654	238386.2220	2.8207	0.0062	197196.5172	1147625.0135
COUNTRY_IND	155306.4011	126316.3449	1.2295	0.2229	-96500.6302	407113.4325
COUNTRY_NZ	194218.9120	173491.9293	1.1195	0.2667	-151630.9280	540068.7521
COUNTRY_PAK	75921.7670	193463.5545	0.3924	0.6959	-309740.7804	461584.3143
COUNTRY_SA	64283.3894	144587.6773	0.4446	0.6579	-223946.8775	352513.6563
COUNTRY_SL	17360.1530	176333.7497	0.0985	0.9218	-334154.7526	368875.0586
COUNTRY_WI	10607.7792	230686.7892	0.0460	0.9635	-449257.9303	470473.4887
COUNTRY_ZIM	-145494.4793	401505.2815	-0.3624	0.7181	-945880.6296	654891.6710
PLAYING ROLE_Batsman	75724.7643	150250.0240	0.5040	0.6158	-223793.1844	375242.7130
PLAYING ROLE_Bowler	15395.8752	126308.1272	0.1219	0.9033	-236394.7744	267186.5249
PLAYING ROLE_W.Keeper	-71358.6280	213585.7444	-0.3341	0.7393	-497134.0278	354416.7718
CAPTAINCY EXP_1	164113.3972	123430.6353	1.3296	0.1878	-81941.0772	410167.8716

Omnibus:	0.891	Durbin-Watson:	2.244
Prob(Omnibus):	0.640	Jarque-Bera (JB):	0.638
Skew:	0.190	Prob(JB):	0.727
Kurtosis:	3.059	Condition No.:	84116

MLR model output from Python is provided in Table 4.4. As per the *p*-value (<0.05), only the features *HS*, *AGE_2*, *AVE* and *COUNTRY_ENG* have come out significant. The model says that none of the other features are influencing *SOLD PRICE* (at a significance value of 0.05). This is not very intuitive and could be a result of multi-collinearity effect of variables.

4.5.6 | Multi-Collinearity and Handling Multi-Collinearity

When the dataset has a large number of independent variables (features), it is possible that few of these independent variables (features) may be highly correlated. The existence of a high correlation between independent variables is called multi-collinearity. Presence of multi-collinearity can destabilize the

multiple linear regression model. Thus, it is necessary to identify the presence of multi-collinearity and take corrective actions.

Multi-collinearity can have the following impact on the model:

1. The standard error of estimate, $S_e(\bar{\beta})$, is inflated.
2. A statistically significant explanatory variable may be labelled as statistically insignificant due to the large p -value. This is because when the standard error of estimate is inflated, it results in an underestimation of t -statistic value.
3. The sign of the regression coefficient may be different, that is, instead of negative value for regression coefficient, we may have a positive regression coefficient and vice versa.
4. Adding/removing a variable or even an observation may result in large variation in regression coefficient estimates.

4.5.6.1 Variance Inflation Factor (VIF)

Variance Inflation Factor (VIF) is a measure used for identifying the existence of multi-collinearity. For example, consider two independent variables X_1 and X_2 and regression between them.

$$X_1 = \alpha_0 + \alpha_1 X_2 \quad (4.15)$$

Let R_{12} be the R -squared value of this model. Then the VIF, which is a measure of multi-collinearity, is given by

$$\text{VIF} = \frac{1}{1 - R_{12}^2} \quad (4.16)$$

$\sqrt{\text{VIF}}$ is the value by which the t -statistic value is deflated. VIF value of greater than 4 requires further investigation to assess the impact of multi-collinearity. One approach to eliminate multi-collinearity is to remove one of the variables from the model building.

`variance_inflation_factor()` method available in `statsmodels.stats.outliers_influence` package can be used to calculate VIF for the features. The following method is written to calculate VIF and assign the VIF to the columns and return a DataFrame:

```
from statsmodels.stats.outliers_influence import variance_inflation_factor

def get_vif_factors( X ):
    X_matrix = X.as_matrix()
    vif = [ variance_inflation_factor( X_matrix, i ) for i in range( X_matrix.shape[1] ) ]
    vif_factors = pd.DataFrame()
    vif_factors['column'] = X.columns
    vif_factors['VIF'] = vif

    return vif_factors
```

Now, calling the above method with the X features will return the VIF for the corresponding features.

```
vif_factors = get_vif_factors( X[X_features] )
vif_factors
```

	Column	VIF
1	T-WKTS	7.679284
2	ODI-RUNS-S	16.426209
3	ODI-SR-B	13.829376
4	ODI-WKTS	9.951800
5	ODI-SR-BL	4.426818
6	RUNS-S	16.135407
7	HS	22.781017
8	AVE	25.226566
9	SR-B	21.576204
10	SIXERS	9.547268
11	RUNS-C	38.229691
12	WKTS	33.366067
13	AVE-BL	100.198105
14	ECON	7.650140
15	SR-BL	103.723846
16	AGE_2	6.996226
17	AGE_3	3.855003
18	COUNTRY_BAN	1.469017
19	COUNTRY_ENG	1.391524
20	COUNTRY_IND	4.568898
21	COUNTRY_NZ	1.497856
22	COUNTRY_PAK	1.796355
23	COUNTRY_SA	1.886555
24	COUNTRY_SL	1.984902
25	COUNTRY_WI	1.531847
26	COUNTRY_ZIM	1.312168
27	PLAYING ROLE_Batsman	4.843136
28	PLAYING ROLE_Bowler	3.795864
29	PLAYING ROLE_W.Keeper	3.132044
30	CAPTAINCY EXP_1	4.245128

4.5.6.2 Checking Correlation of Columns with Large VIFs

We can generate a correlation heatmap to understand the correlation between the independent variables which can be used to decide which features to include in the model. We will first select the features that have VIF value of more than 4.

```
columns_with_large_vif = vif_factors[vif_factors.vif > 4].column
```

Then plot the heatmap for features with VIF more than 4 (see Figure 4.5).

```
plt.figure( figsize = (12,10) )
sn.heatmap( X[columns_with_large_vif].corr(), annot = True );
plt.title("Figure 4.5 - Heatmap depicting correlation between features");
```

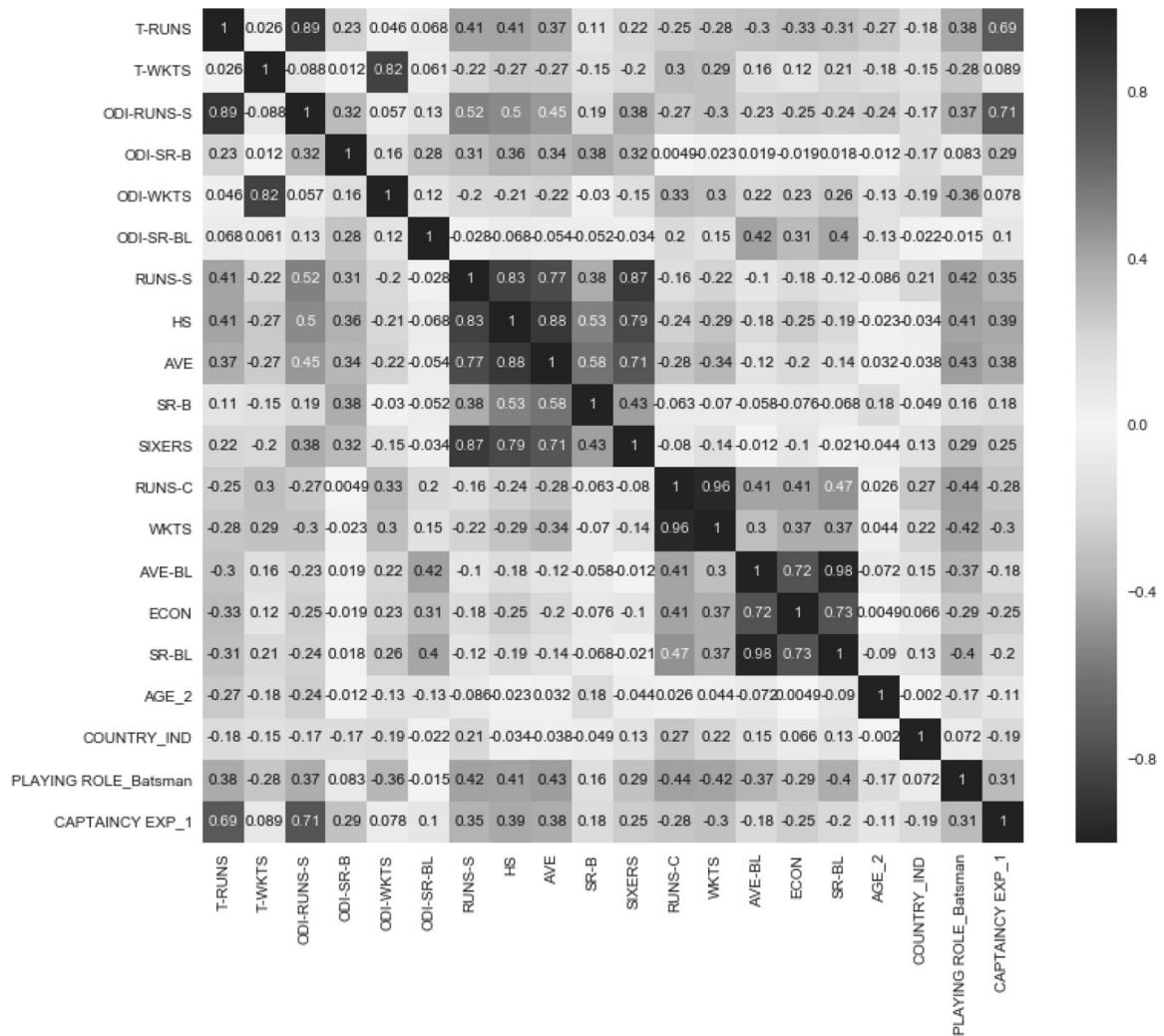


FIGURE 4.5 Heatmap depicting the correlation between features.

The following observations are made from the heatmap (Figure 4.5):

1. *T-RUNS* and *ODI-RUNS-S* are highly correlated, whereas *ODI-WKTS* and *T-WKTS* are highly correlated.

2. Batsman features like *RUNS-S*, *HS*, *AVE*, *SIXERS* are highly correlated, while bowler's features like *AVE-BL*, *ECON* and *SR-BL* are highly correlated.

To avoid multi-collinearity, we can keep only one column from each group of highly correlated variables and remove the others. Now which one to keep and which one to remove depends on the understanding of the data and the domain.

We have decided to remove the following features. Please note that it may take multiple iterations before deciding at a final set of variables, which do not have multi-collinearity. These iterations have been omitted here for simplicity.

```
columns_to_be_removed = [ 'T-RUNS', 'T-WKTS', 'RUNS-S', 'HS', 'AVE',
                           'RUNS-C', 'SR-B', 'AVE-BL', 'ECON',
                           'ODI-SR-B', 'ODI-RUNS-S', 'AGE_2', 'SR-BL' ]
```

```
X_new_features = list( set(X_features) - set(columns_to_be_removed) )
```

```
get_vif_factors( X[X_new_features] )
```

	Column	VIF
0	AGE_3	1.779861
1	ODI-SR-BL	2.822148
2	COUNTRY_IND	3.144668
3	COUNTRY_ENG	1.131869
4	COUNTRY_NZ	1.173418
5	COUNTRY_PAK	1.334773
6	COUNTRY_WI	1.194093
7	COUNTRY_SL	1.519752
8	COUNTRY_ZIM	1.205305
9	CAPTAINCY_EXP_1	2.458745
10	PLAYING ROLE_W. Keeper	1.900941
11	PLAYING ROLE_Bowler	3.060168
12	SIXERS	2.397409
13	COUNTRY_BAN	1.094293
14	COUNTRY_SA	1.416657
15	PLAYING ROLE_Batsman	2.680207
16	ODI-WKTS	2.742889
17	WKTS	2.883101

The VIFs on the final set of variables indicate that there is no multi-collinearity present any more (VIF values are less than 4). We can proceed to build the model with these set of variables now.

4.5.6.3 Building a New Model after Removing Multi-collinearity

```
train_X = train_X[X_new_features]
ipl_model_2 = sm.OLS(train_y, train_X).fit()
ipl_model_2.summary2()
```

TABLE 4.5 Model summary for *ipl_model_2*

Model:	OLS	Adj. R-squared:	0.728
Dependent Variable:	SOLD PRICE	AIC:	2965.1080
Date:	2018-04-08 07:27	BIC:	3012.7070
No. Observations:	104	Log-Likelihood:	-1464.6
Df Model:	18	F-statistic:	16.49
Df Residuals:	86	Prob (F-statistic):	1.13e-20
R-squared:	0.775	Scale:	1.2071e+11

	Coef.	Std.Err.	t	P > t	[0.025	0.975]
COUNTRY_IND	282829.8091	96188.0292	2.9404	0.0042	91614.3356	474045.2827
COUNTRY_BAN	-108758.6040	369274.1916	-0.2945	0.7691	-842851.4010	625334.1930
AGE_3	-8950.6659	98041.9325	-0.0913	0.9275	-203851.5772	185950.2453
COUNTRY_PAK	122810.2480	159600.8063	0.7695	0.4437	-194465.6541	440086.1502
COUNTRY_WI	-22234.9315	213050.5847	-0.1044	0.9171	-445765.4766	401295.6135
ODI-WKTS	772.4088	470.6354	1.6412	0.1044	-163.1834	1708.0009
COUNTRY_SA	108735.9086	115092.9596	0.9448	0.3474	-120061.3227	337533.1399
COUNTRY_ENG	682934.7166	216150.8279	3.1595	0.0022	253241.0920	1112628.3411
CAPTAINCY EXP_1	208376.6957	98128.0284	2.1235	0.0366	13304.6315	403448.7600
WKTS	2431.8988	2105.3524	1.1551	0.2512	-1753.4033	6617.2008
SIXERS	7862.1259	2086.6101	3.7679	0.0003	3714.0824	12010.1694
PLAYING ROLE_W.Keeper	-55121.9240	169922.5271	-0.3244	0.7464	-392916.7280	282672.8801
COUNTRY_ZIM	-67977.6781	390859.9289	-0.1739	0.8623	-844981.5006	709026.1444
PLAYING ROLE_Bowler	-18315.4968	106035.9664	-0.1727	0.8633	-229108.0215	192477.0279
COUNTRY_SL	55912.3398	142277.1829	0.3930	0.6953	-226925.3388	338750.0184
COUNTRY_NZ	142968.8843	151841.7382	0.9416	0.3491	-158882.5009	444820.2695
PLAYING ROLE_Batsman	121382.0570	106685.0356	1.1378	0.2584	-90700.7746	333464.8886
ODI-SR-BL	909.0021	1267.4969	0.7172	0.4752	-1610.6983	3428.7026

Omnibus:	8.635	Durbin-Watson:	2.252
Prob(Omnibus):	0.013	Jarque-Bera (JB):	8.345
Skew:	0.623	Prob(JB):	0.015
Kurtosis:	3.609	Condition No.:	1492

In Table 4.5, the p -values of the coefficients estimated show whether the variables are statistically significant in influencing response variables or not. If the p -value is less than the significance value (α) then the feature is statistically significant, otherwise it is not. The value of α is usually selected as 0.05; however, it may be chosen based on the context of the problem.

Based on the p -values, only the variables *COUNTRY_IND*, *COUNTRY_ENG*, *SIXERS*, *CAPTAINCY_EXP_1* have come out statistically significant. So, the features that decide the *SOLD PRICE* are

1. Whether the players belong to India or England (that is, origin country of the player).
2. How many sixes has the player hit in previous versions of the IPL? How many wickets have been taken by the player in ODIs?
3. Whether the player has any previous captaincy experience or not.

Let us create a new list called *significant_vars* to store the column names of significant variables and build a new model (Table 4.6).

```
significant_vars = ['COUNTRY_IND', 'COUNTRY_ENG', 'SIXERS',
'CAPTAINCY_EXP_1']

train_X = train_X[significant_vars]

ipl_model_3 = sm.OLS(train_y, train_X).fit()
ipl_model_3.summary2()
```

TABLE 4.6 Model summary for *ipl_model_3*

Model:	OLS	Adj. R-squared:	0.704
Dependent Variable:	SOLD PRICE	AIC:	2961.8089
Date:	2018-04-08 07:27	BIC:	2972.3864
No. Observations:	104	Log-Likelihood:	-1476.9
Df Model:	4	F-statistic:	62.77
Df Residuals:	100	Prob (F-statistic):	1.97e-26
R-squared:	0.715	Scale:	1.3164e+11

	Coef.	Std.Err.	t	P > t	[0.025	0.975]
COUNTRY_IND	387890.2538	63007.1511	6.1563	0.0000	262885.8606	512894.6471
COUNTRY_ENG	731833.6386	214164.4988	3.4172	0.0009	306937.3727	1156729.9045
SIXERS	8637.8344	1675.1313	5.1565	0.0000	5314.4216	11961.2472
CAPTAINCY_EXP_1	359725.2741	74930.3460	4.8008	0.0000	211065.6018	508384.9463

Omnibus:	1.130	Durbin-Watson:	2.238
Prob(Omnibus):	0.568	Jarque-Bera (JB):	0.874
Skew:	0.223	Prob(JB):	0.646
Kurtosis:	3.046	Condition No.:	165

The following inference can be derived from the latest model *ipl_model_3* (Table 4.6)

1. All the variables are statistically significant, as *p*-value is less than 0.05%.
2. The overall model is significant as the *p*-value for the *F*-statistics is also less than 0.05%.
3. The model can explain 71.5% of the variance in *SOLD PRICE* as the *R*-squared value is 0.715 and the adjusted *R*-squared value is 0.704%. Adjusted R-squared is a measure that is calculated after normalizing SSE and SST with the corresponding degrees of freedom.

4.5.7 | Residual Analysis in Multiple Linear Regression

4.5.7.1 Test for Normality of Residuals (P-P Plot)

One of the important assumptions of regression is that the residuals should be normally distributed. This can be verified using P-P plot. We will develop a method called *draw_pp_plot()* which takes the model output (residuals) and draws the P-P plot.

```
def draw_pp_plot( model, title ):
    probplot = sm.ProbPlot( model.resid );
    plt.figure( figsize = (8, 6) );
    probplot.ppplot( line='45' );
    plt.title( title );
    plt.show();
```

```
draw_pp_plot(ipl_model_3,
             "Figure 4.6 - Normal P-P Plot of Regression Standardized
             Residuals");
```

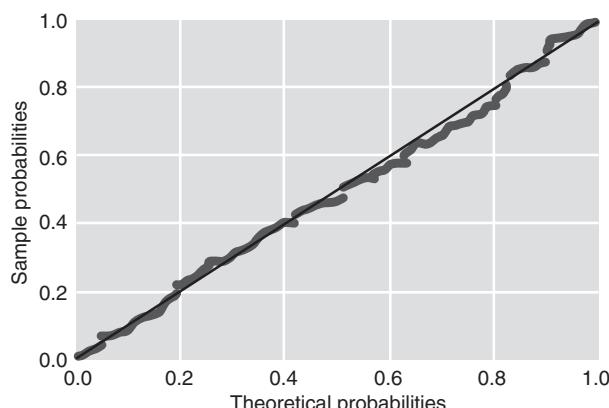


FIGURE 4.6 Normal P-P plot of regression standardized residuals for *ipl_model_3*.

The P-P plot in Figure 4.6 shows that the residuals follow an approximate normal distribution.

4.5.7.2 Residual Plot for Homoscedasticity and Model Specification

As explained in the previous example, the residual plot is a plot between standardized fitted values and residuals. The residuals should not have any patterns. Residual plot with shape such as a funnel may indicate existence of heteroscedasticity. Any pattern in the residual plot may also indicate the use of incorrect functional form in the regression model development.

```
def plot_resid_fitted(fitted, resid, title):
    plt.scatter( get_standardized_values( fitted ),
                get_standardized_values( resid ) )
    plt.title(title)
    plt.xlabel("Standardized predicted values")
    plt.ylabel("Standardized residual values")
    plt.show()
```

```
plot_resid_fitted(ipl_model_3.fittedvalues,
                  ipl_model_3.resid,
                  "Figure 4.7 - Residual Plot")
```

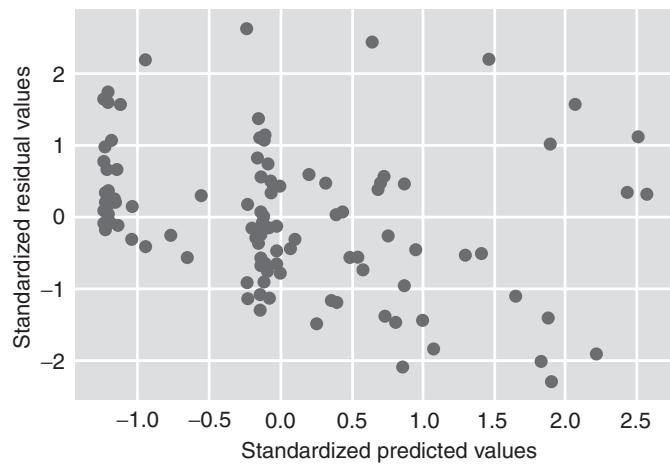


FIGURE 4.7 Residual plot.

The residuals in Figure 4.7 do not show any signs of heteroscedasticity (no funnel-like pattern).

4.5.8 | Detecting Influencers

In OLS estimate, we assume that each record in the data has equal influence on the model parameters (regression coefficients), which may not be true. We can use the function *influence_plot()* as explained in

the previous example to identify highly influential observations. Leverage values of more than $3(k + 1)/n$ are treated as highly influential observations.

```
k = train_X.shape[1]
n = train_X.shape[0]
```

```
print("Number of variables:", k, " and number of observations:", n)
```

Number of variables: 4 and number of observations: 104

```
leverage_cutoff = 3*((k + 1)/n)
print("Cutoff for leverage value:", round(leverage_cutoff, 3))
```

Cutoff for leverage value: 0.144

So, observations with leverage values more than 0.178 are highly influential.

```
from statsmodels.graphics.regressionplots import influence_plot
fig, ax = plt.subplots(figsize=(8, 6))
influence_plot(ipl_model_3, ax=ax)
plt.title("Figure 4.8 - Leverage Value Vs Residuals")
plt.show()
```

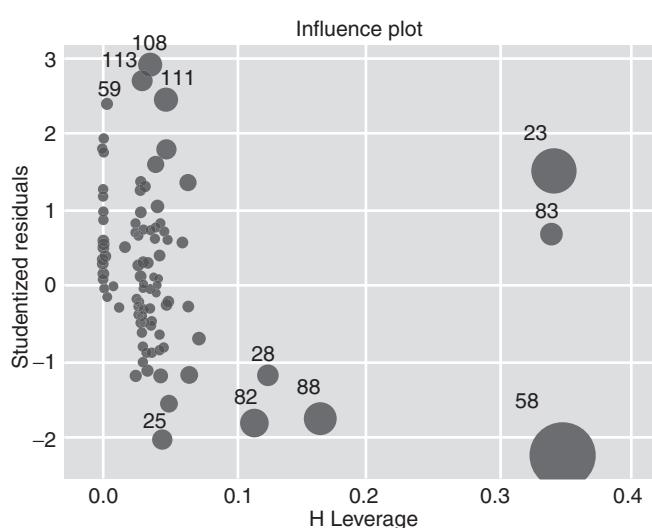


FIGURE 4.8 Leverage value versus residuals.

The above diagram shows there are three observations 23, 58, 83 that have comparatively high leverage with residuals. We can filter out the influential observations.

```
ipl_auction_df[ipl_auction_df.index.isin([23, 58, 83])]
```

SI.NO.	PLAYER NAME	AGE	COUNTRY	TEAM	PLAYING ROLE	T-RUNS	T-WKTS	ODI-RUNS-S	ODI-SR-B	SR-B	SIXERS	RUNS-C		
23	24	Flintoff, A	2	ENG	CSK	Allrounder	3845	226	3394	88.82	...	116.98	2	105
58	59	Mascarenhas, AD	2	ENG	RR+	Allrounder	0	0	245	95.33	...	101.37	1	331
83	84	Pietersen, KP	2	ENG	RCB+	Batsman	6654	5	4184	86.76	...	141.20	30	215

These observations do not have large residuals. So, it may not be necessary to remove these observations. But if the observations need to be removed, the following method *drop()* can be used:

```
train_X_new = train_X.drop([23, 58, 83], axis = 0)
train_y_new = train_y.drop([23, 58, 83], axis = 0)
```

We will leave the next steps of building a model after removing the important observations, to the reader to try themselves for practice.

4.5.9 | Transforming Response Variable

Transformation is a process of deriving new dependent and/or independent variables to identify the correct functional form of the regression model. For example, the dependent variable Y may be replaced in the model with $\ln(Y)$, $1/Y$, etc. and similarly, an independent variable X may be replaced with $\ln(X)$, $1/X$, etc.

Transformation in MLR is used to address the following issues:

1. Poor fit (low R -squared value).
2. Pattern in residual analysis indicating a potential non-linear relationship between the dependent and independent variables.
3. Residuals do not follow a normal distribution.
4. Residuals are not homoscedastic.

We will create a new response variable by taking the square root of the existing values. This can be done by using Numpy's *sqrt()* method.

```
train_y = np.sqrt(train_y)
```

```
ipl_model_4 = sm.OLS(train_y, train_X).fit()
ipl_model_4.summary2()
```

TABLE 4.7 Model summary for *ipl_model_4*

Model:	OLS	Adj. R-squared:	0.741
Dependent Variable:	SOLD PRICE	AIC:	1527.9999
Date:	2018-04-08 07:30	BIC:	1538.5775

(Continued)

TABLE 4.7 (Continued)

No. Observations:	104	Log-Likelihood:	-760.00
Df Model:	4	F-statistic:	75.29
Df Residuals:	100	Prob (F-statistic):	2.63e-29
R-squared:	0.751	Scale:	1.3550e+05

	Coef.	Std.Err.	t	P > t	[0.025	0.975]
COUNTRY_IND	490.7089	63.9238	7.6765	0.0000	363.8860	617.5318
COUNTRY_ENG	563.0261	217.2801	2.5912	0.0110	131.9486	994.1036
SIXERS	8.5338	1.6995	5.0213	0.0000	5.1620	11.9055
CAPTAINCY EXP_1	417.7575	76.0204	5.4953	0.0000	266.9352	568.5799

Omnibus:	0.017	Durbin-Watson:	1.879
Prob(Omnibus):	0.992	Jarque-Bera (JB):	0.145
Skew:	0.005	Prob(JB):	0.930
Kurtosis:	2.817	Condition No.:	165

In Table 4.7, the *R*-squared value of the model has increased to 0.751. The P-P plot in Figure 4.9 also shows that the residuals follow a normal distribution.

```
draw_pp_plot(ipl_model_4,
             "Figure 4.9 - Normal P-P Plot of Regression Standardized
             Residuals");
```

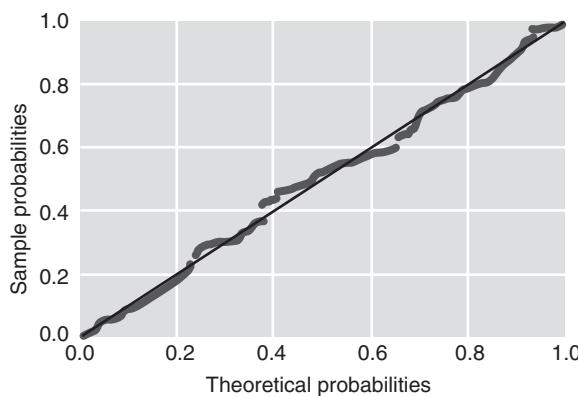


FIGURE 4.9 Normal P-P plot of regression standardized residuals.

4.5.10 | Making Predictions on the Validation Set

After the final model is built as per our requirements and the model has passed all diagnostic tests, we can apply the model on the validation test data to predict the *SOLD PRICE*. As the model we have built

predicts square root of the *SOLD PRICE*, we need to square the predicted values to get the actual *SOLD PRICE* of the players.

```
pred_y = np.power(ipl_model_4.predict(test_X[train_X.columns]), 2)
```

4.5.10.1 Measuring RMSE

Calculating RMSE of the validation set.

```
from sklearn import metrics
np.sqrt(metrics.mean_squared_error(pred_y, test_y))
```

496151.18122558104

4.5.10.2 Measuring R-squared Value

Similarly *R*-squared value is calculated using the code below.

```
np.round( metrics.r2_score(pred_y, test_y), 2 )
```

0.44

The accuracy (*R*-squared) value on validation set (0.44) is quite low compared to the accuracy reported by the model on training dataset (0.751). This could a sign of model over-fitting. We will discuss under-fitting and over-fitting of models and how to deal with it in a later chapter.

4.5.10.3 Auto-correlation Between Error Terms

One of the assumptions of the regression model is that there should be no correlation between error terms. If there is an auto-correlation, the standard error estimate of the beta coefficient may be underestimated and that will result in over-estimation of the *t*-statistic value, which, in turn, will result in a low *p*-value. Thus, a variable which has no statistically significant relationship with the response variable may be accepted in the model due to the presence of auto-correlation. The presence of auto-correlation can be established using the Durbin–Watson test. As a thumb rule, a Durbin–Watson statistic close to 2 would imply the absence of autocorrelation.

The model summary shows that the Durbin–Watson statistics value is 2.2, which very close to 2; hence it can be concluded that the error terms are not auto-correlated. Note that auto-correlation is more relevant in the case of time-series data.

CONCLUSION

In this chapter, we learnt how to build, diagnose, understand and validate simple and multiple linear regression model using Python APIs.

1. Multiple linear regression (MLR) model is used to find the existence of association relationship between a dependent variable and more than one independent variables.

2. In MLR, the regression coefficients are called partial regression coefficients, since they measure the change in the value of the dependent variable for every one unit change in the value of the independent variable, when all other independent variables in the model are kept constant.
3. While building the MLR model, categorical variables with n categories should be replaced with $(n - 1)$ dummy (binary) variables to avoid model misspecification.
4. Apart from checking for normality, heteroscedasticity, every MLR model should be checked for the presence of multi-collinearity, since multi-collinearity can destabilize the MLR model.
5. Transforming of variables can be done to improve accuracy or remove heteroscedasticity and improve the model fit.

EXERCISES

Answer Questions 1 to 4 using the following dataset. Develop Python code to answer the following questions.

The dataset *country.csv* contains Corruption Perception Index and Gini Index of 20 countries. Corruption Perception Index close to 100 indicates low corruption and close to 0 indicates high corruption. Gini Index is a measure of income distribution among citizens of a country (high Gini indicates high inequality). Corruption Index is taken from Transparency International, while Gini Index is sourced from Wikipedia.

1. Develop a simple linear regression model ($Y = \beta_0 + \beta_1 X$) between corruption perception index (Y) and Gini index (X). What is the change in the corruption perception index for every one unit increase in Gini index?
2. What proportion of the variation in corruption perception index is explained by Gini index?
3. Is there a statistically significant relationship between corruption perception index and Gini index at alpha value 0.1?
4. Calculate the 95% confidence interval for the regression coefficient β_1 .

Answer Questions 5 to 18 using the following dataset. Develop Python code to answer the following questions.

The dataset DAD Hospital contains data about cost of treatment for patients. It also contains information about patient's demographics and health parameters. The following are the description of the columns in the dataset.

SI_NO	Sequence Number
AGE	Patients' age
GENDER	M for Male and F for Female
MARITAL_STATUS	Patient is married or unmarried
KEY_COMPLAINTS_CODE	Code for patients' complaint. The key health condition for which the patient was treated.
BODY_HEIGHT	Body weight
HR_PULSE	Heart pulse rate

(Continued)

SI_NO	Sequence Number
BP_HIGH	Systolic blood pressure
BP_LOW	Diastolic blood pressure
RR	Respiratory Rate
PAST_MEDICAL_HISTORY_CODE	Code of patients' medical history e.g. diabetes, hypertension etc.
HB	Haemoglobin Count
UREA	Urea Count
CREATININE	Creatinine count in blood and urine
MODE_OF_ARRIVAL	Mode of arrival e.g. AMBULANCE, WALKED IN
STATE_AT_THE_TIME_OF_ARRIVAL	Patients' state at the time of arrival
TYPE_OF_ADMSN	Type of admission ELECTIVE, EMERGENCY
TOTAL_COST_TO_HOSPITAL	Actual cost to the hospital
TOTAL_AMOUNT_BILLED_TO_THE_PATIENT	Actual amount billed to the patient
CONCESSION	Concession given at the time of billing
ACTUAL_RECEIVABLE_AMOUNT	Actual amount received by the hospital
TOTAL_LENGTH_OF_STAY	Number of days patients was admitted to the hospital
LENGTH_OF_STAY_ICU	Number of days patients was admitted in the ICU
LENGTH_OF_STAY_WARD	Number of days patients was admitted in the hospital's ward
IMPLANT_USED_Y_N	Any implants used
COST_OF_IMPLANT	Cost of implant used

- DAD hospital wants to understand what are the key factors influencing the cost to hospital. The hospital wants to provide treatment packages (fixed price contract) to the patients at the time of the admission. Can the hospital build a model using the historical data to estimate the cost of treatment?
- Build a correlation matrix between all the numeric features in the dataset. Report the features which are correlated at a cutoff of 0.70. What actions will you take on the features which are highly correlated?
- Build a new feature named BMI using body height and body weight. Include this as a part of the DataFrame created in step 1.
- Past medical history code has 175 instances of missing value (NaN). Impute 'None' as a label wherever the value is NaN for this feature.
- Select the features that can be used to build a model to estimate the cost to the hospital.
- Identify which features are numerical and which are categorical. Create a new DataFrame with the selected numeric features and categorical features. Encode the categorical features and create dummy features.
- Which features have the symptoms of multi-collinearity and need to be removed from the model?
- Find the outliers in the dataset using Z-score and Cook's distance. If required, remove the observations from the dataset.
- Split the data into training set and test set. Use 80% of data for model training and 20% for model testing.
- Build a regression model with *statsmodel.api* to estimate the total cost to hospital. How do you interpret the model outcome?

15. Which features are statistically significant in predicting the total cost to the hospital?
16. Build a linear regression model with significant features and report model performance.
17. Conduct residual analysis using P-P plot to find out if the model is valid.
18. Predict the total cost using the test set and report RMSE of the model.

REFERENCES

1. Kutner M, Nachtsheim CJ, Neter J, and Li W (2013). *Applied Linear Statistics Models*, 5th edition, McGraw Hill Education.
2. U Dinesh Kumar (2017). *Business Analytics: The Science of Data-Driven Decision Making*, Wiley India, Delhi.
3. Statsmodel <https://www.statsmodels.org/stable/index.html>
4. Pandas Library: <https://pandas.pydata.org/>
5. Matplotlib Library: <https://matplotlib.org/>
6. Seabon Library: <https://seaborn.pydata.org/>

CHAPTER

5

Classification Problems

LEARNING OBJECTIVES

After completing this chapter, you will be able to

- Understand the concept of classification problems and their applications across different sectors.
- Learn to build logistic regression and decision tree models using the Python package *statsmodel* and *sklearn* APIs.
- Learn to perform activities such as
 - (a) splitting the dataset into training and validation datasets.
 - (b) building model using Python package on training dataset and test on the validation dataset.
- Learn to measure model accuracies using confusion matrix, sensitivity (recall), specificity, precision, receiver operating characteristic (ROC) curve, and area under ROC curve (AUC).
- Learn about statistical significance, model diagnostics and finding the optimal cut-off probabilities for classification.

5.1 | CLASSIFICATION OVERVIEW

Classification problems are an important category of problems in analytics in which the outcome variable or response variable (Y) takes discrete values. Primary objective of a classification model is to predict the probability of an observation belonging to a class, known as **class probability**. Few examples of classification problem are as follows:

1. A bank would like to classify the customers based on risk such as low-risk or high-risk customers.
2. E-commerce providers would like to predict whether a customer is likely to churn or not. It is a loss of revenue for the company if an existing and valuable customer churns.
3. Health service providers may classify a patient, based on the diagnostic results, as positive (presence of disease) or negative.
4. The HR department of a firm may want to predict if an applicant would accept an offer or not.
5. Predicting outcome of a sporting event, for example, whether India will win the next world cup cricket tournament.

6. Sentiments of customers on a product or service may be classified as positive, negative, neutral, and sarcastic.
7. Based on the image of a plant, one can predict if the plant is infected with a specific disease or not.

Classification problems may have binary or multiple outcomes or classes. Binary outcomes are called **binary classification** and multiple outcomes are called **multinomial classification**. There are several techniques used for solving classification problems such as logistic regression, classification trees, discriminant analysis, neural networks, and support vector machines. In this chapter, we will focus on *logistic regression* and *classification tree* (aka *decision tree learning*) for building and evaluating classification models. Some more models will be discussed in *Chapter 6*.

5.2 | BINARY LOGISTIC REGRESSION

Logistic regression is a statistical model in which the response variable takes a discrete value and the explanatory variables can either be continuous or discrete. If the outcome variable takes only two values, then the model is called binary logistic regression model. Assume that the outcomes are called positive (usually coded as $Y = 1$) and negative (usually coded as $Y = 0$). Then the probability that a record belongs to a positive class, $P(Y = 1)$, using the binary logistic regression model is given by

$$P(Y = 1) = \frac{e^Z}{1 + e^Z} \quad (5.1)$$

where

$$Z = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 + \cdots + \beta_m X_m$$

Here X_1, X_2, \dots, X_m are the independent variables or features. The logistic regression has an S-shaped curve, and gives the class probability of an observation belonging to class labelled as 1, that is, $P(Y = 1)$.

Equation (5.1) can be re-written as

$$\ln\left(\frac{P(Y = 1)}{1 - P(Y = 1)}\right) = Z = \beta_0 + \beta_1 X_1 + \cdots + \beta_m X_m \quad (5.2)$$

The left-hand side of Eq. (5.2) is a log natural of odds and is known as logit function; the right-hand side is a linear function. Such models are called **generalized linear models** (GLM). In GLM, the errors may not follow normal distribution and there exists a transformation function of the outcome variable that takes a linear functional form.

The logistic function in Eq. (5.1) has an S-shaped curve (thus also known as Sigmoid function). Consider an example of classifying a person with heart disease and without heart disease. That is, we are interested in finding the probability of a person suffering from heart disease as a function of age. The outcome variable is either someone suffers from heart disease or not and the independent variable (feature) is *age*. Then a logistic or sigmoid function can be fit to explain the probability of suffering from heart disease with respect to age as shown in Figure 5.1.

The corresponding logistic function is given as

$$P(\text{Heart Disease} = 1) = \frac{e^{\beta_0 + \beta_1 \text{Age}}}{1 + e^{\beta_0 + \beta_1 \text{Age}}}$$

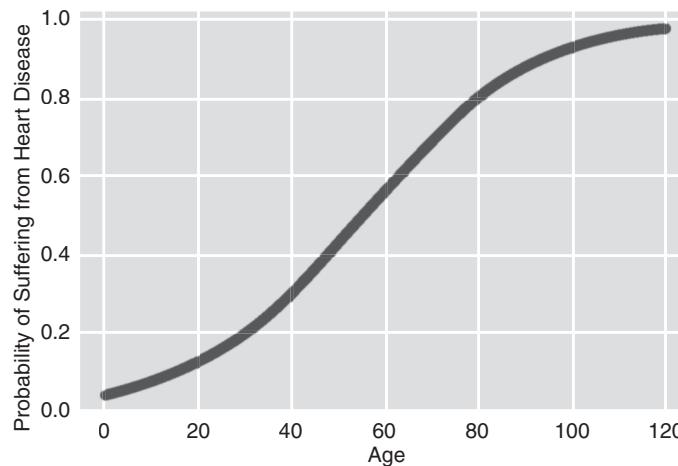


FIGURE 5.1 Logistic regression model for predicting probability of suffering from Heart Disease.

5.3 | CREDIT CLASSIFICATION

In this chapter, we will use the German credit rating dataset available at the University of California Irvine (UCI) machine learning laboratory to predict whether a credit is a good or bad credit. Credit classification is one of the popular classification problems encountered by financial institutions. For example, when a customer applies for a loan, financial institutions such as banks would like to predict the probability of default. In this section, we will be using the German credit rating dataset provided by Prof. Hofmann (available at UCI machine learning data repository¹). The data contains several attributes (Table 5.1) of the persons who availed the credit.

TABLE 5.1 Data description for German credit rating dataset

Variable	Variable Type	Description	Categories
checkin_acc	categorical	Status of existing checking account	<ul style="list-style-type: none"> • A11 : ... < 0 DM • A12 : 0 <= ... < 200 DM • A13 : ... >= 200 DM / salary assignments for at least 1 year • A14 : no checking account
duration	numerical	Duration of the credit given in months	<ul style="list-style-type: none"> • A30 : no credits taken/all credits paid back duly • A31 : all credits at this bank paid back duly • A32 : existing credits paid back duly till now • A33 : delay in paying off in the past • A34 : critical account/other credits existing (not at this bank)
credit_history	categorical	Credit History	

(Continued)

¹ Source: <https://archive.ics.uci.edu/ml/datasets/Statlog+%28German+Credit+Data%29>.

TABLE 5.1 (Continued)

Variable	Variable Type	Description	Categories
amount	numerical	Amount of credit/loan	
savings_acc	categorical	Balance in savings account	<ul style="list-style-type: none"> • A61 : ... <100 DM • A62 : 100 <= ... <500 DM • A63 : 500 <= ... < 1000 DM • A64 : .. >=1000 DM • A65 : unknown/no savings account
present_emp_since	categorical	Employment in years	<ul style="list-style-type: none"> • A71 : unemployed • A72 : ... < 1 year • A73 : 1 <=... < 4 years • A74 : 4 <=... < 7 years • A75 : .. >= 7 years
inst_rate	numerical	Installment rate	
personal_status	categorical	Marital status	<ul style="list-style-type: none"> • A91 : male : divorced/separated • A92 : female : divorced/separated/married • A93: male : single • A94 : male : married/widowed • A95 : female : single
residing_since	numerical	Residing since in years	
age	numerical	Age in years	
inst_plans	categorical	Other installment plans of the applicant	<ul style="list-style-type: none"> • A141 : bank • A142 : stores • A143 : none
checkin_acc	categorical	Balance in checking account	<ul style="list-style-type: none"> • A11 : ... < 0 DM • A12 : 0 <= ... < 200 DM • A13 : ... >= 200 DM/salary assignments for at least 1 year • A14 : no checking account
job	categorical	Job	<ul style="list-style-type: none"> • A171 : unemployed/unskilled - non-resident • A172 : unskilled - resident • A173 : skilled employee/official • A174 : management/self-employed/highly qualified employee/officer
status	categorical	Credit status	<ul style="list-style-type: none"> • 0: Good Credit • 1: Bad Credit

The dataset contains records with comma separated values (*csv*) and can be read using pandas' *read_csv()* method.

Reading and displaying few records from the dataset is shown below:

```
import pandas as pd
import numpy as np

credit_df = pd.read_csv("German Credit Data.csv")
credit_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 14 columns):
checkin_acc      1000    non-null   object
duration         1000    non-null   int64
credit_history   1000    non-null   object
amount           1000    non-null   int64
savings_acc      1000    non-null   object
present_emp_since 1000    non-null   object
inst_rate         1000    non-null   int64
personal_status  1000    non-null   object
residing_since   1000    non-null   int64
age               1000    non-null   int64
inst_plans        1000    non-null   object
num_credits       1000    non-null   int64
job               1000    non-null   object
status            1000    non-null   int64
dtypes: int64(7), object(7)
memory usage: 109.5+ KB
```

The dataset contains 1000 records and 14 columns. Printing all the columns together will exceed the page width, so we will print the first 7 columns, that is, [1:7] and then print the remaining ones. The first column is a serial number and hence not selected for printing.

```
credit_df.iloc[0:5,1:7]
```

	duration	credit_history	Amount	savings_acc	present_emp_since	inst_rate
0	6	A34	1169	A65	A75	4
1	48	A32	5951	A61	A73	2
2	12	A34	2096	A61	A74	2
3	42	A32	7882	A61	A74	2
4	24	A33	4870	A61	A73	3

Now, let us print the remaining 7 columns.

```
credit_df.iloc[0:5, 7:]
```

	personal_status	residing_since	age	inst_plans	num_credits	job	status
0	A93	4	67	A143	2	A173	0
1	A92	2	22	A143	1	A173	1
2	A93	3	49	A143	1	A172	0
3	A93	4	45	A143	1	A173	0
4	A93	4	53	A143	2	A173	1

There are few columns which are categorical and have been inferred as objects. For example, *checkin_acc* has following categories:

1. A11 : ... < 0 DM
2. A12 : 0 <= ... < 200 DM
3. A13 : ... >= 200 DM/salary assignments for at least 1 year
4. A14 : no checking account

To find the number of observations in the dataset for good credit and for bad credit, a simple *value_count()* on the *status* column will provide the information.

```
credit_df.status.value_counts()
```

```
0    700
1    300
Name: status, dtype: int64
```

The output displays that there are 700 observations of good credit and 300 observations of bad credit.

For building a classification model, the *status* column will be used as the dependent variable, while the remaining columns will be independent variables or features. We will create a list named *X_features* and store the names of all independent variables for future use.

```
X_features = list( credit_df.columns )
X_features.remove( 'status' )
X_features

['checkin_acc',
'duration',
'credit_history',
'amount',
'savings_acc',
'present_emp_since',
'inst_rate',
'personal_status',
'residing_since',
```

```
'age',
'inst_plans',
'num_credits',
'job']
```

5.3.1 | Encoding Categorical Features

There are several categorical features in the data which need to be binary encoded (using dummy variables), also known as one hot encoding (OHE). Details of encoding a categorical feature is already explained in Section 4.5.3, Chapter 4. Pandas method `pd.get_dummies()` is used to encode the categorical features and create dummy variables. If the feature has n categories, then either n or $(n - 1)$ dummy variables can be created [the model will have only $(n - 1)$ dummy variables]. If $(n - 1)$ categories are created, then the omitted category serves as base category. `pd.get_dummies()` takes a parameter `drop_first`, which drops the first category if set to `True`.

```
encoded_credit_df = pd.get_dummies(credit_df[X_features],
                                   drop_first = True)
```

We can list the new dummy features that have been created along with other continuous features.

```
list(encoded_credit_df.columns)
```

```
['duration',
'amount',
'inst_rate',
'residing_since',
'age',
'num_credits',
'checkin_acc_A12',
'checkin_acc_A13',
'checkin_acc_A14',
'credit_history_A31',
'credit_history_A32',
'credit_history_A33',
'credit_history_A34',
'savings_acc_A62',
'savings_acc_A63',
'savings_acc_A64',
'savings_acc_A65',
'present_emp_since_A72',
'present_emp_since_A73',
'present_emp_since_A74',
'present_emp_since_A75',
'personal_status_A92',
'personal_status_A93',
'personal_status_A94',
```

```
'inst_plans_A142',
'inst_plans_A143',
'job_A172',
'job_A173',
'job_A174']
```

For example, *checkin_acc* variable is encoded into the following three dummy variables and the base category.

checkin_acc_A11 is dropped. If all the following three variables' values are set to 0, it indicates the account type is *checkin_acc_A11*.

1. *checkin_acc_A12*
2. *checkin_acc_A13*
3. *checkin_acc_A14*

Displaying the dummy features to verify how they are encoded.

```
encoded_credit_df[['checkin_acc_A12',
                   'checkin_acc_A13',
                   'checkin_acc_A14']].head(5)
```

	checkin_acc_A12	checkin_acc_A13	checkin_acc_A14
0	0	0	0
1	1	0	0
2	0	0	1
3	0	0	0
4	0	0	0

The first record has all the dummy variable values set to 0, indicating it is checkin account type *checkin_acc_A11*, which is the *base category*. The account type in second observation is *checkin_acc_A12*, so on.

As the variables have been prepared, we can set *X (features)* and *Y (outcome)* variables before proceeding to build the model. Similar to *OLS()* method to build a linear regression model (as explained in Chapter 4), *statsmodel* api provides *Logit()* method for building logistic regression model. We need to add a new column and set its value to 1, for the model to estimate the intercept. In logistic regression, the regression parameters (feature weights) are estimated using maximum likelihood estimation.

```
import statsmodels.api as sm
Y = credit_df.status
X = sm.add_constant(encoded_credit_df)
```

5.3.2 | Splitting Dataset into Training and Test Sets

Before building the model, split the dataset into 70:30 (or 80:20) ratio for creating training and validation datasets. The model will be built using the training set and tested using test set.

```
from sklearn.model_selection import
X_train,X_test,y_train,y_test = train_test_split(X,
                                                Y,
                                                test_size = 0.3,
                                                random_state = 42)
```

X_train and *y_train* contain the independent variables and response variable values for the training dataset respectively. Similarly, *X_test* and *y_test* contain the independent variables and response variable values for the test dataset, respectively.

5.3.3 | Building Logistic Regression Model

We will fit the model using *Logit* and pass training sets *y_train* and *X_train* as parameters. Invoking *fit()* method available in *statsmodels.api* module helps to estimate the parameters for the logistic regression model.

```
import statsmodels.api as sm
logit = sm.Logit(y_train, X_train)
logit_model = logit.fit()
```

```
Optimization terminated successfully.
    Current function value: 0.488938
    Iterations 6
```

The method *fit()* on *Logit()* estimates the parameters and returns model details in the variable *logit_model*, which contains the model parameters, accuracy measures, residual values among other details.

5.3.4 | Printing Model Summary

The model object *logit_model* has *summary2()* method, which provides details of the model, its parameters and results of all necessary statistical tests. The model summary contains important information to validate the model.

```
logit_model.summary2()
```

Model:	Logit	Pseudo R-squared:	0.198
Dependent Variable:	status	AIC:	744.5132
Date:	2018-05-06 18:46	BIC:	881.0456
No. Observations:	700	Log-Likelihood:	-342.26

(Continued)

Df Model:	29	LL-Null:	-426.75
Df Residuals:	670	LLR p-value:	1.0630e-21
Converged:	1.0000	Scale:	1.0000
No. Iterations:	6.0000		

	Coef.	Std.Err.	z	P > z	[0.025	0.975]
const	-0.1511	1.1349	-0.1331	0.8941	-2.3754	2.0733
duration	0.0206	0.0104	1.9927	0.0463	0.0003	0.0409
amount	0.0001	0.0000	2.3765	0.0175	0.0000	0.0002
inst_rate	0.3064	0.0986	3.1083	0.0019	0.1132	0.4996
residing_since	0.0967	0.0920	1.0511	0.2932	-0.0836	0.2771
age	-0.0227	0.0103	-2.2131	0.0269	-0.0428	-0.0026
num_credits	0.2854	0.2139	1.3342	0.1821	-0.1338	0.7045
checkin_acc_A12	-0.4126	0.2391	-1.7260	0.0843	-0.8812	0.0559
checkin_acc_A13	-0.9053	0.4338	-2.0868	0.0369	-1.7556	-0.0550
checkin_acc_A14	-1.6052	0.2586	-6.2073	0.0000	-2.1120	-1.0983
credit_history_A31	0.1532	0.5795	0.2643	0.7916	-0.9827	1.2890
credit_history_A32	-0.4960	0.4411	-1.1245	0.2608	-1.3604	0.3685
credit_history_A33	-0.8881	0.5022	-1.7683	0.0770	-1.8724	0.0962
credit_history_A34	-1.4124	0.4528	-3.1190	0.0018	-2.2999	-0.5249
savings_acc_A62	-0.0496	0.3208	-0.1545	0.8772	-0.6782	0.5791
savings_acc_A63	-0.6640	0.4818	-1.3779	0.1682	-1.6084	0.2804
savings_acc_A64	-1.1099	0.6019	-1.8439	0.0652	-2.2896	0.0699
savings_acc_A65	-0.6061	0.2745	-2.2080	0.0272	-1.1441	-0.0681
present_emp_since_A72	0.0855	0.4722	0.1810	0.8564	-0.8401	1.0110
present_emp_since_A73	-0.0339	0.4492	-0.0754	0.9399	-0.9142	0.8465
present_emp_since_A74	-0.3789	0.4790	-0.7910	0.4289	-1.3178	0.5600
present_emp_since_A75	-0.2605	0.4554	-0.5721	0.5673	-1.1532	0.6321
personal_status_A92	-0.0069	0.4841	-0.0142	0.9887	-0.9557	0.9419
personal_status_A93	-0.4426	0.4764	-0.9291	0.3528	-1.3762	0.4911
personal_status_A94	-0.3080	0.5554	-0.5546	0.5792	-1.3967	0.7806
inst_plans_A142	-0.2976	0.5157	-0.5772	0.5638	-1.3084	0.7131
inst_plans_A143	-0.4458	0.2771	-1.6086	0.1077	-0.9889	0.0974
job_A172	-0.0955	0.7681	-0.1243	0.9011	-1.6009	1.4100
job_A173	-0.0198	0.7378	-0.0269	0.9786	-1.4658	1.4262
job_A174	-0.0428	0.7371	-0.0581	0.9537	-1.4876	1.4019

5.3.5 | Model Diagnostics

It is important to validate the logistic regression model to ensure its validity and goodness of fit before it can be used for practical applications. The following measures are used to validate a logistic regression model:

1. Wald's test (a Chi-square test) for checking the statistical significance of individual predictor (feature) variables. This is equivalent to t -test in the MLR (Multiple Linear Regression) model.
2. Likelihood ratio test for checking the statistical significance of the overall model (LLR p -value is reported in the output). Likelihood ratio test is also used for variable (feature) selection.
3. Pseudo R^2 : It is a measure of goodness of the model. It is called pseudo R^2 because it does not have the same interpretation of R^2 as in the MLR model.

The model summary suggests that as per Wald's test, only 8 features are statistically significant at a significant value of $\alpha = 0.05$, as p -values are less than 0.05. p -value for likelihood ratio test (almost 0.00) indicates that the overall model is statistically significant.

Defining a method `get_significant_vars()` that takes the model object as an input and returns the list of significant variables after filtering out the variable with corresponding p -value less than 0.05.

```
def get_significant_vars( lm ):
    #Store the p-values and corresponding column names in a dataframe
    var_p_vals_df = pd.DataFrame( lm.pvalues )
    var_p_vals_df['vars'] = var_p_vals_df.index
    var_p_vals_df.columns = ['pvals', 'vars']
    # Filter the column names where p-value is less than 0.05
    return list( var_p_vals_df[var_p_vals_df.pvals <= 0.05]['vars'] )
```

Invoking the above method `get_significant_vars()` and passing the `logit_model`, we can get the list of significant variables.

```
significant_vars = get_significant_vars( logit_model )

significant_vars
[ 'duration',
  'amount',
  'inst_rate',
  'age',
  'checkin_acc_A13',
  'checkin_acc_A14',
  'credit_history_A34',
  'savings_acc_A65' ]
```

Only the features '`duration`', '`amount`', '`inst_rate`', '`age`', '`checkin_acc_A13`', '`checkin_acc_A14`', '`credit_history_A34`', '`savings_acc_A65`' have come out as significant variables. Now, we can build a logistic regression using only the significant variables.

```
final_logit = sm.Logit( y_train,
                      sm.add_constant( X_train [significant_vars] ) ).fit()
```

Optimization terminated successfully.
 Current function value: 0.511350
 Iterations 6

```
final_logit.summary2()
```

Model:	Logit	Pseudo R-squared:	0.161
Dependent Variable:	status	AIC:	733.8898
Date:	2018-05-06 18:50	BIC:	774.8495
No. Observations:	700	Log-Likelihood:	-357.94
Df Model:	8	LL-Null:	-426.75
Df Residuals:	691	LLR p-value:	7.4185e-26
Converged:	1.0000	Scale:	1.0000
No. Iterations:	6.0000		

	Coef.	Std.Err.	z	P > z	[0.025	0.975]
const	-0.8969	0.4364	-2.0551	0.0399	-1.7523	-0.0415
duration	0.0197	0.0098	2.0033	0.0451	0.0004	0.0390
amount	0.0001	0.0000	2.3205	0.0203	0.0000	0.0002
inst_rate	0.2811	0.0929	3.0264	0.0025	0.0991	0.4632
age	-0.0216	0.0089	-2.4207	0.0155	-0.0392	-0.0041
checkin_acc_A13	-0.8038	0.4081	-1.9697	0.0489	-1.6037	-0.0040
checkin_acc_A14	-1.5452	0.2187	-7.0649	0.0000	-1.9738	-1.1165
credit_history_A34	-0.8781	0.2319	-3.7858	0.0002	-1.3327	-0.4235
savings_acc_A65	-0.5448	0.2581	-2.1108	0.0348	-1.0507	-0.0389

The negative sign in coefficient value indicates that as the value of this variable increases, the probability of being a bad credit decreases, that is, $P(Y_i = 1)$ decreases. A positive sign means that the probability of being a bad credit increases as the corresponding value of the variable increases. Some observations from the model output are:

1. The log of odds ratio or probability of being a bad credit increases as *duration*, *amount*, *inst_rate* increases. For example, one unit change in duration results in 0.019723 unit change in log of odds ratio [refer to UD Kumar (2017) for detailed discussion].
2. The probability of being a bad credit decreases as age increases. This means that older people tend to pay back their credits on time compared to younger people.

5.3.6 | Predicting on Test Data

We will use the *final_logit* model and the significant features to predict the class probability. The *predict()* method will return the predicted probabilities for each observation in the test dataset. We will store the actual predicted class and predicted probabilities in a dataframe to compare and calculate various accuracy measures.

```
y_pred_df=pd.DataFrame( {"actual": y_test,
                           "predicted_prob": final_logit.predict(
                           sm.add_constant( X_test[significant_vars]))})
```

We can print the predictions of few test samples randomly using the *sample* method of *DataFrame*.

```
y_pred_df.sample(10, random_state = 42)
```

	actual	predicted_prob
557	1	0.080493
798	0	0.076653
977	0	0.345979
136	0	0.249919
575	0	0.062264
544	0	0.040768
332	1	0.833093
917	1	0.370667
678	0	0.388392
363	0	0.088952

To understand how many observations the model has classified correctly and how many it has not, a cut-off probability needs to be assumed. Assume that the cut-off probability is 0.5; then all observations with predicted probability of more than 0.5 will be predicted as bad credits and rest all as good credits. We will explore techniques on how to find the optimal cut-off probability later in *Section 5.3.10: Finding Optimal Classification Cut-off*. For now, let us assume it to be 0.5.

Now iterate through predicted probability of each observation using *map()* and tag the observation as bad credit (1) if probability value is more than 0.5 or as good credit (0) otherwise.

```
y_pred_df['predicted'] = y_pred_df.predicted_prob.map(
                           lambda x: 1 if x > 0.5 else 0)

y_pred_df.sample(10, random_state = 42)
```

	actual	predicted_prob	predicted
557	1	0.080493	0
798	0	0.076653	0
977	0	0.345979	0
136	0	0.249919	0
575	0	0.062264	0
544	0	0.040768	0
332	1	0.833093	1
917	1	0.370667	0
678	0	0.388392	0
363	0	0.088952	0

The *actual* column in the DataFrame depicts the actual label of the credit in the test set, while predicted column depicts what the model has predicted by taking 0.5 as cut-off probability value. For observation 557, the model predicts very low probability (0.08) of being a bad credit whereas it is actually a bad credit. The model has wrongly classified this one. But the model predicts high probability (0.833) of being a bad credit for observation 332, which is actually a bad credit. The model correctly predicted the class in this case.

It can be noticed from *actual* and *predicted* columns that some classifications are correct and some are wrong. For better understanding of this confusion, we can build a confusion matrix, which is discussed next.

5.3.7 | Creating a Confusion Matrix

Confusion matrix (also known as error matrix or classification table) is a matrix formed by checking the actual values and predicted values of observations in the dataset. To create a confusion matrix, first import *matplotlib.pyplot* and *seaborn* libraries to render all plots and charts.

```
import matplotlib.pyplot as plt
import seaborn as sn
%matplotlib inline
```

sklearn.metrics module provides a method to build a confusion matrix if actual and predicted values are passed as inputs parameters. For better understanding, this matrix can be plotted as a heatmap. *seaborn's heatmap* function can render the confusion matrix for easier interpretation. Define a method *draw_cm()* that takes actual and predicted values, creates the confusion matrix, and draws the heatmap.

```
from sklearn import metrics
def draw_cm( actual, predicted ):
    ## Cret
    cm = metrics.confusion_matrix( actual, predicted, [1,0] )
    sn.heatmap(cm, annot=True, fmt='%.2f',
               xticklabels = ["Bad credit", "Good Credit"],
               yticklabels = ["Bad credit", "Good Credit"] )
```

```
plt.ylabel('True label')
plt.xlabel('Predicted label')
plt.show()
```

Invoke the method *draw_cm()* to plot the results of our model predictions.

```
draw_cm( y_pred_df.actual,
          y_pred_df.predicted )
```

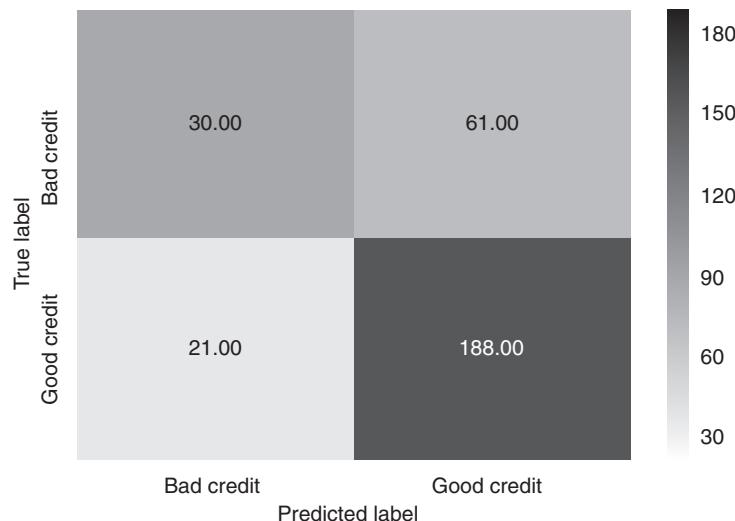


FIGURE 5.2 Confusion matrix with cut-off probability of 0.5.

In the confusion matrix in Figure 5.2, the columns represent the predicted label (class), while the rows represent the actual label (class). For example, out of 91 (i.e., 30 + 61) bad credits, only 30 have been classified correctly as bad credits and rest 61 have been classified as good credits when the cut-off probability is 0.5.

Each row represents the actual bad credit and good credit observations present in the test dataset and each column represents predicted values of outcome variable. We can note the following:

1. Left-top quadrant represents actual bad credit and is correctly classified as bad credit. This is called True Positives (TP).
2. Left-down quadrant represents actual good credit and is incorrectly classified as bad credit. This is called False Positives (FP).
3. Right-top quadrant represents actual bad credit and is incorrectly classified as good credit. This is called False Negatives (FN).
4. Right-down quadrant represents actual good credit and is correctly classified as good credit. This is called True Negatives (TN).

5.3.8 | Measuring Accuracies

In classification, the model performance is often measured using concepts such as sensitivity, specificity, precision, and F-score. The ability of the model to correctly classify positives and negatives is called **sensitivity** (also known as recall or true positive rate) and **specificity** (also known as true negative rate), respectively. The terminologies sensitivity and specificity originated in medical diagnostics.

Sensitivity or Recall (True Positive Rate)

Sensitivity is the conditional probability that the predicted class is positive given that the actual class is positive. Mathematically, sensitivity is given by

$$\text{Sensitivity} = \frac{TP}{TP + FN} \quad (5.3)$$

Specificity (True Negative Rate)

Specificity is the conditional probability that the predicted class is negative given that the actual class is negative. Mathematically, specificity is given by

$$\text{Specificity} = \frac{TN}{TN + FP} \quad (5.4)$$

Precision

Precision is the conditional probability that the actual value is positive given that the prediction by the model is positive. Mathematically, precision is given by

$$\text{Precision} = \frac{TP}{TP + FP} \quad (5.5)$$

F-Score

F-Score is a measure that combines precision and recall (harmonic mean between precision and recall). Mathematically, F-Score is given by

$$\text{F-Score} = \frac{2 \times \text{Recall} \times \text{Precision}}{\text{Recall} + \text{Precision}} \quad (5.6)$$

classification_report() method in *sklearn.metrics* gives a detailed report of precision, recall, and F-score for each class.

```
print( metrics.classification_report( y_pred_df.actual,
                                      y_pred_df.predicted ) )
```

	precision	recall	f1-score	support
0	0.76	0.90	0.82	209
1	0.59	0.33	0.42	91
avg/total	0.70	0.73	0.70	300

The model is very good at identifying the good credits ($Y = 0$), but not very good at identifying bad credits. This is the result for cut-off probability of 0.5%. This can be improved by choosing the right cut-off probability.

We can plot the distributions of predicted probability values for good and bad credits to understand how well the model can distinguish bad credits from good credits.

```
plt.figure( figsize = (8,6) )
# Plotting distribution of predicted probability values for bad credits
sn.distplot( y_pred_df[y_pred_df.actual == 1]["predicted_prob"],
              kde=False, color = 'b',
              label = 'Bad Credit')

# Plotting distribution of predicted probability values for good credits
sn.distplot( y_pred_df[y_pred_df.actual == 0]["predicted_prob"],
              kde=False, color = 'g',
              label = 'Good Credit')

plt.legend()
plt.show()
```

We can use a chart to understand how the distributions of predicted probabilities for bad credit and good credit look like (Figure 5.3). Larger the overlap between predicted probabilities for different classes, higher will be the misclassifications.

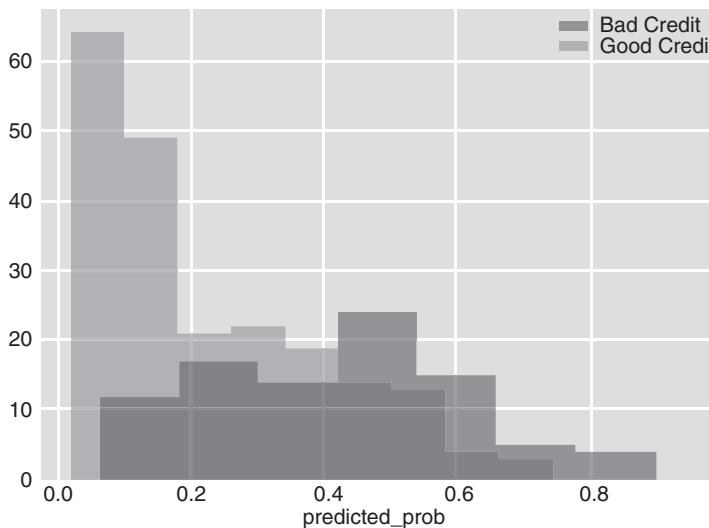


FIGURE 5.3 Distribution of predicted probability values by the model for both good and bad credits.

5.3.9 | Receiver Operating Characteristic (ROC) and Area Under the Curve (AUC)

Receiver operating characteristic (ROC) curve can be used to understand the overall performance (worth) of a logistic regression model (and, in general, of classification models) and used for model

selection. The term has its origin in electrical engineering when electrical signals were used for predicting enemy objects (such as submarines and aircraft) during World War II. Given a random pair of positive and negative class records, ROC gives the proportions of such pairs that will be correctly classified.

ROC curve is a plot between sensitivity (true positive rate) on the vertical axis and 1 – specificity (false positive rate) on the horizontal axis. We will write a method *draw_roc()* which takes the actual classes and predicted probability values and then draws the ROC curve (Figure 5.4).

metrics.roc_curve() returns different threshold (cut-off) values and their corresponding false positive and true positive rates. Then these values can be taken and plotted to create the ROC curve. *metrics.roc_auc_score()* returns the area under the curve (AUC).

```
def draw_roc( actual, probs ):
    # Obtain fpr, tpr, thresholds
    fpr,
    tpr,
    thresholds = metrics.roc_curve( actual,
                                    probs,
                                    drop_intermediate = False )
auc_score = metrics.roc_auc_score( actual, probs )
plt.figure(figsize=(8, 6))
# Plot the fpr and tpr values for different threshold values
plt.plot( fpr, tpr, label='ROC curve (area = %0.2f)' % auc_score )
# draw a diagonal line connecting the origin and top right most point
plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
# Setting x and y labels
plt.xlabel('False Positive Rate or [1 - True Negative Rate]')
plt.ylabel('True Positive Rate')
plt.legend(loc="lower right")
plt.show()

return fpr, tpr, thresholds
```

```
fpr, tpr, thresholds = draw_roc( y_pred_df.actual,
                                y_pred_df.predicted_prob)
```

The diagonal line in Figure 5.4 represents the case of not using a model (no discrimination between positive and negative); the area below the diagonal line is equal to 0.5 (it is a right-angle triangle, area of right-angle triangle is $0.5ab$, where a and b are the lengths of the sides which is equal to 1 in this case). Sensitivity and/or specificity are likely to change when the cut-off probability is changed. The line above the diagonal line in Figure 5.4 captures how sensitivity and (1 – specificity) change when the cut-off probability is changed. Model with higher AUC is preferred and AUC is frequently used for model selection.

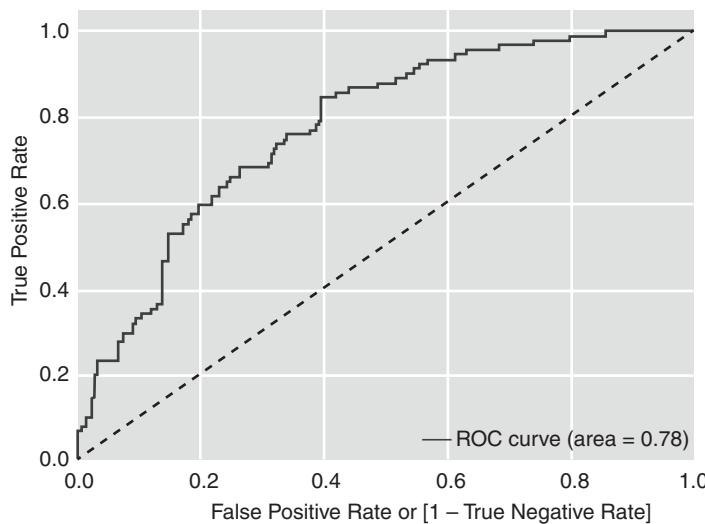


FIGURE 5.4 ROC curve.

As a thumb rule, AUC of at least 0.7 is required for practical application of the model. AUC greater than 0.9 implies an outstanding model. Caution should be exercised while selecting models based on AUC, especially when the data is imbalanced (i.e., dataset which has less than 10% positives). In case of imbalanced datasets, the AUC may be very high (greater than 0.9); however, either sensitivity or specificity values may be poor.

For this example, the AUC is 0.78, which implies the model is fairly good. The AUC can also be obtained by calling `roc_auc_score` from `sklearn.metrics`.

```
auc_score = metrics.roc_auc_score( y_pred_df.actual,
                                  y_pred_df.predicted_prob )
round( float( auc_score ), 2 )
```

0.78

5.3.10 | Finding Optimal Classification Cut-off

While using logistic regression model, one of the decisions that a data scientist has to make is to choose the right classification cut-off probability (P_c). The overall accuracy, sensitivity, and specificity will depend on the chosen cut-off probability. The following two methods are used for selecting the cut-off probability:

1. Youden's index
2. Cost-based approach

5.3.10.1 Youden's Index

Sensitivity and specificity change when we change the cut-off probability. Youden's index (Youden, 1950) is a classification cut-off probability for which the following function is maximized (also known as J-statistic):

$$\text{Youden's Index} = \text{J-Statistic} = \underset{p}{\text{Max}} [\text{Sensitivity}(p) + \text{Specificity}(p) - 1] \quad (5.7)$$

We already know that sensitivity is also known as True Positive Rate (TPR) and specificity is known as True Negative Rate (TNR). That is, select the cut-off probability for which $(\text{TPR} + \text{TNR} - 1)$ is maximum.

`draw_roc()` has returned `tpr` and `fpr` values, which we have stored in variables `tpr`, `fpr`, respectively. The variable `thresholds` captures the corresponding cut-off probabilities. We can take difference of `tpr` and `fpr` and then sort the values in descending order. The `thresholds` value, for which Eq. (5.7) is maximum, should be the optimal cut-off.

```
tpr_fpr = pd.DataFrame( { 'tpr': tpr,
                           'fpr': fpr,
                           'thresholds': thresholds } )

tpr_fpr['diff'] = tpr_fpr.tpr - tpr_fpr.fpr
tpr_fpr.sort_values('diff', ascending = False)[0:5]
```

	fpr	thresholds	tpr	diff
159	0.397129	0.221534	0.846154	0.449025
160	0.401914	0.216531	0.846154	0.444240
161	0.406699	0.215591	0.846154	0.439455
158	0.397129	0.223980	0.835165	0.438036
165	0.421053	0.207107	0.857143	0.436090

From the above result, the optimal cut-off is 0.22. We can now classify all the observations beyond 0.22 predicted probability as bad credits and others as good credits. We will capture these new classes in `predicted_new` variable and then draw a new confusion matrix (Figure 5.5).

```
y_pred_df['predicted_new'] = y_pred_df.predicted_prob.map(
    lambda x: 1 if x > 0.22 else 0)
```

```
draw_cm( y_pred_df.actual,
          y_pred_df.predicted_new)
```

Now print the report with cut-off probability of 0.22.

```
print(metrics.classification_report( y_pred_df.actual,
                                      y_pred_df.predicted_new ))
```

	precision	recall	f1-score	support
0	0.90	0.60	0.72	209
1	0.48	0.85	0.61	91
avg / total	0.77	0.68	0.69	300

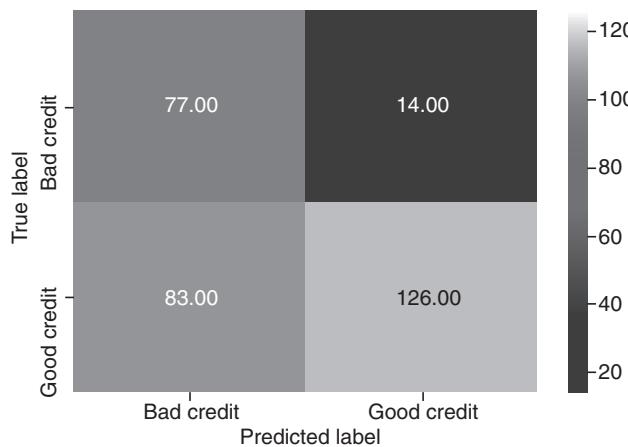


FIGURE 5.5 Confusion matrix with optimal cut-off using Yoden's index.

With cut-off probability of 0.22, the model is able to classify the bad credits better and the F1-score for bad credits ($Y = 1$) has also improved to 0.61.

5.3.10.2 Cost-Based Approach

As the cost of false negatives and false positives is not same, the optimal classification cut-off probability can also be determined using cost-based approach, which finds the cut-off where the total cost is minimum. In the cost-based approach, we assign penalty cost for misclassification of positives and negatives and find the total cost for a cut-off probability.

Assuming cost of a false positive is C_1 and that of a false negative is C_2 , total cost will be

$$\text{Total cost} = \text{FN} \times C_1 + \text{FP} \times C_2$$

The optimal cut-off probability is the one which minimizes the total penalty cost. We will write a method `get_total_cost()` to return the penalty cost for a specific cut-off probability. The method also takes the relative cost of false negatives and false positives. The function will take actual and predicted values, find the number of false positives (FPs) and false negatives (FNs), and return the total cost.

```
def get_total_cost( actual, predicted, cost_FPs, cost_FNs ):
    # Get the confusion matrix and calculate cost
    cm = metrics.confusion_matrix( actual, predicted, [1,0] )
    cm_mat = np.array( cm )
    return cm_mat[0,1] * cost_FNs + cm_mat[1,0] * cost_FPs
```

Create a DataFrame which will capture the cost against different cut-off probability values.

```
cost_df = pd.DataFrame( columns = [ 'prob', 'cost' ] )
```

Let us assume that false negatives (predicting a bad credit to be a good credit) are five times costlier than false positives (predicting a good credit to be a bad credit). Deciding the costs will require domain knowledge.

We can calculate the penalty cost for each cut-off probability values between 0.1 and 0.5 with incremental values of 0.01 and sort the costs in ascending order to find the cut-off probability at which the penalty cost is minimum.

```
idx = 0

## Iterate cut-off probability values between 0.1 and 0.5
for each_prob in range(10, 50):
    cost = get_total_cost( y_pred_df.actual,
                           y_pred_df.predicted_prob.map(
                               lambda x: 1 if x > (each_prob/100)      else 0), 1, 5 )
    cost_df.loc[idx] = [(each_prob/100), cost]
    idx += 1

cost_df.sort_values( 'cost', ascending = True )[0:5]
```

	Prob	Cost
4	0.14	150.0
12	0.22	153.0
2	0.12	154.0
10	0.20	154.0
9	0.19	156.0

The lowest cost is achieved at cut-off probability of 0.14 if false negatives are assumed to be five times costlier than false positives. So, let us predict everything beyond 0.14 as bad credit and below 0.14 as good credit.

```
y_pred_df['predicted_using_cost'] = y_pred_df.predicted_prob.map(
    lambda x: 1 if x > 0.14 else 0)

draw_cm( y_pred_df.actual,
          y_pred_df.predicted_using_cost )
```

As shown in Figure 5.6, the model has reduced the false negatives to only 6. This is because of high cost false negatives (5 times) compared to false positives. The number of false positives have increased to 120.

5.4 | GAIN CHART AND LIFT CHART

Gain chart and lift chart are two measures that are used for measuring the benefits of using the logistic regression model (in general, analytical model) and are used in business contexts such as target marketing. In target marketing or marketing campaigns, customers' responses to campaign are usually very low (in many cases the customers who respond to marketing campaigns are less than 1%). Another example of such low conversion is response to advertisement on the Internet (such as Google Adwords

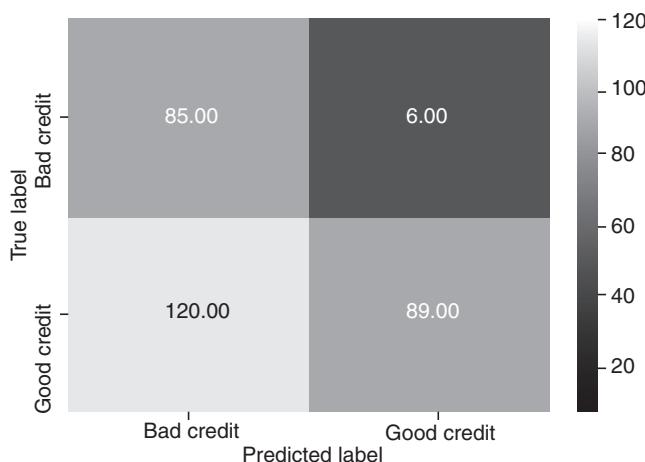


FIGURE 5.6 Confusion matrix with cost-based optimal cut-off.

and mobile advertisements). The organization incurs cost for each customer contact and hence would like to minimize the cost of marketing campaign and at the same time achieve the desired response level from the customers. The gain and lift charts are obtained using the following steps:

1. Predict the probability $Y = 1$ (positive) using the logistic regression (LR) model and arrange the observation in the decreasing order of predicted probability [i.e., $P(Y = 1)$].
2. Divide the datasets into deciles. Calculate the number of positives ($Y = 1$) in each decile and cumulative number of positives up to a decile.
3. Gain is the ratio between cumulative number of the positive observations up to a decile to total number of positive observations in the data. Gain chart is a chart drawn between gain on the vertical axis and decile on the horizontal axis.
4. Lift is the ratio of the number of positive observations up to decile i using the LR model to the expected number of positives up to that decile i based on a random model (not using a model). Lift chart is the chart between lift on the vertical axis and the corresponding decile on the horizontal axis.

$$\text{Gain} = \frac{\text{Cumulative number of positive observations upto decile } i}{\text{Total number of positive observations in the data}}$$

$$\text{Lift} = \frac{\text{Cumulative number of positive observations upto decile } i \text{ using LR model}}{\text{Cumulative number of positive observations upto decile } i \text{ based on random model}}$$

To illustrate the gain and lift charts, we will be using the *bank marketing dataset* available at the University of California, Irvine (UCI) machine learning repository². The data describes a problem in which a bank is interested in predicting which customers may respond to their direct marketing campaign to open a term deposit with the bank. The response variable $Y = 1$ implies that the customer opens a term deposit after the campaign and $Y = 0$ otherwise. The marketing campaign is based on the phone calls.

We will use the smaller dataset *bank.csv*. The description of the features is provided in Table 5.2.

² The dataset is taken from <https://archive.ics.uci.edu/ml/datasets/bank+marketing> (<https://archive.ics.uci.edu/ml/datasets/bank+marketing>). The description of all the variables are also available at the same location.

TABLE 5.2 Data description for bank marketing dataset

Variable	Variable Type	Description
age	numeric	Age of the client who is the target of this marketing exercise
job	categorical	type of job (categories: 'admin.', 'blue-collar', 'entrepreneur', 'housemaid', 'management', 'retired', 'self-employed', 'services', 'student', 'technician', 'unemployed', 'unknown')
marital	categorical	marital status (categories: "married", "divorced", "single"; note: "divorced" means divorced or widowed)
education	categorical	education qualification (categories: "unknown", "secondary", "primary", "tertiary")
default	categorical	customer has credit in default? (categories: 'no', 'yes')
balance	numerical	average yearly balance, in euros
housing-loan	categorical	has housing loan? (categories: 'no', 'yes')
personal-loan	categorical	has personal loan? (categories: 'no', 'yes')
previous-campaign	numerical	number of contacts performed before this campaign and for this client
subscribed	categorical	has the client subscribed a term deposit? (binary: "yes", "no")

5.4.1 | Loading and Preparing the Dataset

The dataset *bank.csv* contains records with comma separated values and can be read using pandas' *read_csv()* method.

Using the following code, we can read and display the first 5 records.

```
import pandas as pd
bank_df = pd.read_csv('bank.csv')
bank_df.head(5)
```

	age	job	marital	education	default	balance	housing-loan	personal-loan	current-campaign	previous-campaign	subscribed
0	30	unemployed	married	primary	no	1787	no	no	1	0	no
1	33	services	married	secondary	no	4789	yes	yes	1	4	no
2	35	management	single	tertiary	no	1350	yes	no	1	1	no
3	30	management	married	tertiary	no	1476	yes	yes	4	0	no
4	59	blue-collar	married	secondary	no	0	yes	no	1	0	no

```
bank_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4521 entries, 0 to 4520
Data columns (total 11 columns):
Age                      4521    non-null   int64
Job                      4521    non-null   object
```

```

Marital           4521    non-null   object
Education         4521    non-null   object
Default           4521    non-null   object
Balance            4521    non-null   int64
housing-loan      4521    non-null   object
personal-loan     4521    non-null   object
current-campaign 4521    non-null   int64
previous-campaign 4521    non-null   int64
Subscribed        4521    non-null   object
dtypes: int64(4), object(7)
memory usage: 388.6+ KB

```

The dataset has a total of 4521 observations, out of which 521 customers subscribed the term deposit (approximately 11.5%) and the remaining 4000 did not subscribe the term deposit. Except *age*, *balance*, *current-campaign*, and *previous-campaign*, rest all are (object) categorical features.

Let us capture the independent variables into the list *X_features*.

```

X_features = list( bank_df.columns )
X_features.remove( 'subscribed' )
X_features

```

```

['age',
 'job',
 'marital',
 'education',
 'default',
 'balance',
 'housing-loan',
 'personal-loan',
 'current-campaign',
 'previous-campaign']

```

Encode the categorical features into dummy variables using the following code:

```

encoded_bank_df = pd.get_dummies( bank_df[X_features],
                                 drop_first = True )

```

The outcome variable *subscribed* is set to *yes* or *no*. This needs to be encoded as 1 (yes) and 0 (no).

```

Y = bank_df.subscribed.map( lambda x: int( x == 'yes' ) )
X = encoded_bank_df

```

Here the dataset is not split into training and test datasets for simplicity as our objective is primarily to understand gain and lift chart.

5.4.2 | Building the Logistic Regression Model

We will start with building a logistic regression model to predict probability of customer subscribing to term deposits using the features available in X .

```
logit_model = sm.Logit( Y, sm.add_constant( X ) ).fit()
```

```
Optimization terminated successfully.  
    Current function value: 0.335572  
    Iterations 7
```

```
logit_model.summary2()
```

Model:	Logit	Pseudo R-squared:	0.061
Dependent Variable:	subscribed	AIC:	3082.2384
Date:	2018-02-26 21:42	BIC:	3236.2341
No. Observations:	4521	Log-Likelihood:	-1517.1
Df Model:	23	LL-Null:	-1615.5
Df Residuals:	4497	LLR p-value:	1.4866e-29
Converged:	1.0000	Scale:	1.0000
No. Iterations:	7.0000		

	Coef.	Std.Err.	z	P > z	[0.025	0.975]
const	-1.7573	0.3799	-4.6251	0.0000	-2.5019	-1.0126
age	0.0078	0.0058	1.3395	0.1804	-0.0036	0.0191
balance	-0.0000	0.0000	-0.2236	0.8231	-0.0000	0.0000
current-campaign	-0.0905	0.0238	-3.8042	0.0001	-0.1371	-0.0439
previous-campaign	0.1414	0.0212	6.6569	0.0000	0.0998	0.1830
job_blue-collar	-0.3412	0.2000	-1.7060	0.0880	-0.7331	0.0508
job_entrepreneur	-0.2900	0.3161	-0.9175	0.3589	-0.9096	0.3295
job_housemaid	-0.0166	0.3339	-0.0497	0.9603	-0.6711	0.6379
job_management	-0.0487	0.1984	-0.2455	0.8061	-0.4375	0.3401
job_retired	0.5454	0.2503	2.1794	0.0293	0.0549	1.0360
job_self-employed	-0.2234	0.2895	-0.7715	0.4404	-0.7909	0.3441
job_services	-0.2248	0.2245	-1.0012	0.3167	-0.6648	0.2152
job_student	0.3888	0.3181	1.2223	0.2216	-0.2346	1.0122

(Continued)

	Coef.	Std.Err.	z	$P > z $	[0.025	0.975]
job_technician	-0.2101	0.1874	-1.1213	0.2622	-0.5773	0.1571
job_unemployed	-0.3723	0.3336	-1.1162	0.2643	-1.0261	0.2815
job_unknown	0.3193	0.4620	0.6913	0.4894	-0.5861	1.2248
marital_married	-0.4012	0.1440	-2.7857	0.0053	-0.6835	-0.1189
marital_single	-0.0463	0.1676	-0.2763	0.7823	-0.3749	0.2822
education_secondary	0.2128	0.1680	1.2670	0.2052	-0.1164	0.5420
education_tertiary	0.3891	0.1935	2.0103	0.0444	0.0098	0.7684
education_unknown	-0.1956	0.2927	-0.6682	0.5040	-0.7693	0.3781
default_yes	0.2286	0.3670	0.6228	0.5334	-0.4908	0.9479
housing-loan_yes	-0.5355	0.1024	-5.2273	0.0000	-0.7362	-0.3347
personal-loan_yes	-0.7139	0.1689	-4.2268	0.0000	-1.0449	-0.3829

Find out which are significant variables using the method `get_significant_vars()` defined in *Section 5.3.5: Model Diagnostics* and build a model with only the significant variables.

```
significant_vars = get_significant_vars(logit_model)
significant_vars
```

```
['const',
 'current-campaign',
 'previous-campaign',
 'job_retired',
 'marital_married',
 'education_tertiary',
 'housing-loan_yes',
 'personal-loan_yes']
```

Setting `X_features` to only significant variables and building a logistic regression model with the significant features.

```
X_features = ['current-campaign',
               'previous-campaign',
               'job_retired',
               'marital_married',
               'education_tertiary',
               'housing-loan_yes',
               'personal-loan_yes']
```

```
logit_model_2 = sm.Logit( Y, sm.add_constant( X[X_features] ) ).fit()
```

```
Optimization terminated successfully.
    Current function value: 0.337228
    Iterations 7
```

Printing the model summary.

```
logit_model_2.summary2()
```

Model:	Logit	Pseudo R-squared:	0.056
Dependent Variable:	subscribed	AIC:	3065.2182
Date:	2018-02-26 21:42	BIC:	3116.5501
No. Observations:	4521	Log-Likelihood:	-1524.6
Df Model:	7	LL-Null:	-1615.5
Df Residuals:	4513	LLR p-value:	8.1892e-36
Converged:	1.0000	Scale:	1.0000
No. Iterations:	7.0000		

	Coef.	Std.Err.	z	P> z	[0.025	0.975]
const	-1.4754	0.1133	-13.0260	0.0000	-1.6974	-1.2534
current-campaign	-0.0893	0.0236	-3.7925	0.0001	-0.1355	-0.0432
previous-campaign	0.1419	0.0211	6.7097	0.0000	0.1004	0.1833
job_retired	0.8246	0.1731	4.7628	0.0000	0.4853	1.1639
marital_married	-0.3767	0.0969	-3.8878	0.0001	-0.5667	-0.1868
education_tertiary	0.2991	0.1014	2.9500	0.0032	0.1004	0.4978
housing-loan_yes	-0.5834	0.0986	-5.9179	0.0000	-0.7767	-0.3902
personal-loan_yes	-0.7025	0.1672	-4.2012	0.0000	-1.0302	-0.3748

p_value for LLR (Likelihood Ratio test) shows (less than 0.05) that the overall model is significant. We will predict the probabilities of the same observations as we have not split the dataset. But if readers have split, then they can use the test dataset for predicting the probabilities.

```
y_pred_df = pd.DataFrame( { 'actual': Y,
                             'predicted_prob': logit_model_2.predict(
                                sm.add_constant( X[X_features] ) ) } )
```

Now sort the observations by their predicted probabilities in the descending order.

```
sorted_predict_df = y_pred_df[ [ 'predicted_prob',
                                 'actual' ] ].sort_values('predicted_prob',
                               ascending = False)
```

After sorting, we will segment all the observations into deciles. First we will find the number of observations in each decile by dividing the total number of observations by 10.

```
num_per_decile = int( len( sorted_predict_df ) / 10 )
print("Number of observations per decile:", num_per_decile)
```

Number of observations per decile: 452

The function `get_deciles()` takes a DataFrame and segments the observations into deciles and marks each observation with the decile number it belongs to. The DataFrame with sorted probabilities should be passed to this function.

```
def get_deciles(df):
    # Set first decile
    df['decile'] = 1

    idx = 0
    # Iterate through all 10 deciles
    for each_d in range(0, 10):
        # Setting each 452 observations to one decile in sequence
        df.iloc[idx:idx+num_per_decile, df.columns.get_loc('decile')] = each_d
        idx += num_per_decile

    df['decile'] = df['decile'] + 1

    return df
```

Now invoke the above method with `sorted_predict_df` as a parameter.

```
deciles_predict_df = get_deciles( sorted_predict_df )
```

The following code displays the first 10 observations in the DataFrame `sorted_predict_df`.

```
deciles_predict_df[0:10]
```

	predicted_prob	actual	decile
3682	0.864769	0	1
97	0.828031	0	1
3426	0.706809	0	1
1312	0.642337	1	1
3930	0.631032	1	1
4397	0.619146	0	1

	predicted_prob	actual	decile
2070	0.609129	0	1
3023	0.573199	0	1
4080	0.572364	0	1
804	0.559350	0	1

Calculating Gain

To calculate the gain, we need to find how many subscriptions (how many 1's) are available in each decile. For this, the *actual* column value can be summed for each decile.

```
gain_lift_df = pd.DataFrame()
deciles_predict_df.groupby(
    'decile')[['actual']].sum().reset_index()
gain_lift_df.columns = ['decile', 'gain']
```

And then the cumulative sum for each subsequent decile divided by the total number of 1's available will give the *gain percentage*.

```
gain_lift_df['gain_percentage'] = (100 * gain_lift_df.gain.
cumsum() / gain_lift_df.gain.sum())
```

```
gain_lift_df
```

	decile	gain	gain_percentage
0	1	125	23.992322
1	2	83	39.923225
2	3	73	53.934741
3	4	53	64.107486
4	5	31	70.057582
5	6	46	78.886756
6	7	37	85.988484
7	8	28	91.362764
8	9	25	96.161228
9	10	20	100.000000

So, if we target only the first 50% of the customers, we have almost 70% subscriptions. The change in gain percentage can be plotted for better understanding (see Figure 5.7).

```
import matplotlib.pyplot as plt
import seaborn as sn
%matplotlib inline

plt.figure( figsize = (8,4))
plt.plot( gain_lift_df['decile'],
          gain_lift_df['gain_percentage'], '-')

plt.show()
```

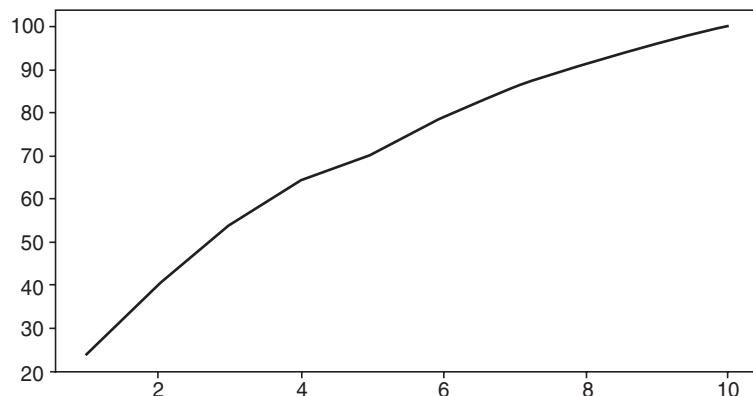


FIGURE 5.7 Gain chart.

The chart in Figure 5.7 shows the increase in gain against deciles. It can be observed that the marginal increase in gain reduces significantly as deciles progress.

Calculating Lift

Lift can be calculated by dividing gain percentage by the decile number.

```
gain_lift_df['lift'] = ( gain_lift_df.gain_percentage
                        / ( gain_lift_df.decile * 10 ) )
gain_lift_df
```

	decile	gain	gain_percentage	lift
0	1	125	23.992322	2.399232
1	2	83	39.923225	1.996161

(Continued)

	decile	gain	gain_percentage	lift
2	3	73	53.934741	1.797825
3	4	53	64.107486	1.602687
4	5	31	70.057582	1.401152
5	6	46	78.886756	1.314779
6	7	37	85.988484	1.228407
7	8	28	91.362764	1.142035
8	9	25	96.161228	1.068458
9	10	20	100.000000	1.000000

The *gain_lift_df* depicts the lift value against decile numbers.

```
plt.figure( figsize = (8,4))
plt.plot( gain_lift_df['decile'], gain_lift_df['lift'], '-' )
plt.show()
```

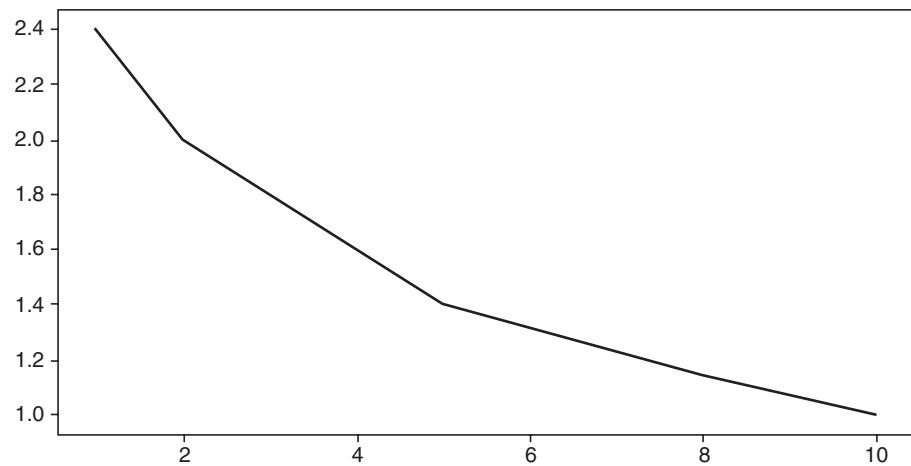


FIGURE 5.8 Lift chart.

As shown in Figure 5.8 per the lift values, targeting customers using the model can capture 2.39 times the number of subscribers compared to a random model in decile 1, 1.99 times up to decile 2, and so on and so forth.

5.5 | CLASSIFICATION TREE (DECISION TREE LEARNING)

Decision Tree Learning or Classification Trees are a collection of divide and conquer problem-solving strategies that use tree-like (inverted tree in which the root is at the top) structure to predict the value of an outcome variable. The tree starts with the root node consisting of the complete data and thereafter

uses intelligent strategies to split the nodes (parent node) into multiple branches (thus, creating children nodes). The original data is divided into subsets in this process. This is done to create more homogenous groups at the children nodes. It is one of the most powerful predictive analytics techniques used for generating business rules.

Classification and Regression Tree (CART) is one of the classification tree techniques. CART is an umbrella term; we will call it a classification tree if the outcome variable value is discrete and a regression tree if the outcome variable value is continuous.

Classification tree uses various impurity measures such as the *Gini Impurity Index* and *Entropy* to split the nodes. Regression tree, on the other hand, splits the node that minimizes the *Sum of Squared Errors* (SSE).

The following steps are used to generate classification and regression trees:

1. Start with the complete training data in the root node.
2. Decide on the measure of impurity, that is, either Gini Impurity Index or Entropy. Search for a predictor variable that minimizes the impurity when the parent node is split into children nodes. This happens when the original data is divided into two subsets using a predictor variable such that it results in the maximum reduction in the impurity in the case of a discrete dependent variable or maximum reduction in SSE in the case of a continuous dependent variable.
3. Repeat step 2 for each subset of the data (for each internal node) using the independent variables until
 - (a) All the dependent variables are exhausted.
 - (b) The stopping criteria are met. Few stopping criteria used are number of levels of tree from the root node, minimum number of observations in parent/child node (e.g., 10% of the training data), and minimum reduction in impurity index.
4. Generate business rules for the leaf (terminal) nodes of the tree.

sklearn.tree.DecisionTreeClassifier provides decision tree algorithm to create the decision tree. It takes the following key parameters:

1. *criterion*: string – The function to measure the quality of a split. Supported criteria are “gini” for the Gini impurity and “entropy” for the information gain. Default is *gini*.
2. *max_depth*: int – The maximum depth of the tree.
3. *min_samples_split*: int or float – The minimum number of samples required to split an internal node. If *int*, then number of samples or if *float*, percentage of total number of samples. Default is 2.
4. *in_samples_leaf*: int or float – The minimum number of samples required to be at a leaf node.

Detailed documentation is available at <http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html>

Let us build the model on the credit rating dataset.

5.5.1 | Splitting the Dataset

Set *encoded_credit_df* as *X* and the column *status* as *Y* and split the data into train and test sets using 70:30 splitting criteria.

```
Y = credit_df.status
X = encoded_credit_df
```

```
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test = train_test_split(X, Y,
                                                test_size = 0.3,
                                                random_state = 42 )
```

5.5.2 | Building Decision Tree Classifier using Gini Criteria

We have already read the dataset using `credit_df` variable. We will set `criterion` to `gini`, `max_depth` to 3, and keep default values for other parameters. In *Section 5.5.6: Finding Optimal Criteria and max_depth*, we will explore the most optimal criteria and `max_depth`.

```
from sklearn.tree import DecisionTreeClassifier
clf_tree = DecisionTreeClassifier(criterion = 'gini',
                                  max_depth = 3)

clf_tree.fit( X_train, y_train )

DecisionTreeClassifier(class_weight=None, criterion='gini', max_
depth=3,
max_features=None, max_leaf_nodes=None,
min_impurity_decrease=0.0, min_impurity_split=None,
min_samples_leaf=1, min_samples_split=2,
min_weight_fraction_leaf=0.0, presort=False, random_state=None,
splitter='best')
```

5.5.3 | Measuring Test Accuracy

Using the model to predict the probability of a bad credit on the test set and obtain ROC AUC score.

```
tree_predict = clf_tree.predict( X_test )
metrics.roc_auc_score( y_test, tree_predict )
```

0.58

5.5.4 | Displaying the Tree

To visualize the tree, use `graphviz` software. For this the following software need to be installed on your machine.

1. GraphViz (as per the OS and version you are using)
2. pip install pydotplus

Then export the tree model to a file using `export_graphviz()` function.

```

from sklearn.tree import export_graphviz
import pydotplus as pdot
from IPython.display import Image

# Export the tree into odt file
export_graphviz( clf_tree,
                  out_file = "chd_tree.odt",
                  feature_names = X_train.columns,
                  filled = True )

# Read the create the image file
chd_tree_graph = pdot.graphviz.graph_from_dot_file( 'chd_tree.odt' )
chd_tree_graph.write_jpg( 'chd_tree.png' )
# Render the png file
Image(filename='chd_tree.png')

```

From the decision tree diagram in Figure 5.9, it can be interpreted that

1. At the top node, there are 700 observations of which 491 are good credits and 209 are bad credits. The corresponding *Gini* index is 0.419. Later in the section, the *Gini* impurity index is explained.
2. *checkin_account_A14* is the most important feature for splitting good and bad credits in the dataset when compared to other features and hence, chosen as the top splitting criteria.
3. The first rule (*checkin_account_A14 < 0.5*) means if the customer has *checkin_account_A14* account or not.
4. This rule has split the dataset into two subsets represented by the second level nodes. On the left node, there are 425 samples (i.e., not having *checkin_account_A14*) and on the right node, there are 275 samples (i.e. having *checkin_account_A14*).
5. The nodes represented by dark shades depict good credits, while the nodes represented by light shades are bad credits.
6. One of the rules can be interpreted as: If the customer does not have *checkin_account_A14* and credit *duration* is less than 33 and does not have *saving_acc_A65*, then there is high probability of being a bad credit. There are 70 records in the dataset that satisfy these conditions and 48 of them have bad credit.
7. Another rule: If the customer has *checkin_account_A14* and *Inst_plans_A143* and age is more than 23.5, then there is a high probability of being good credit.

5.5.5 | Understanding Gini Impurity

Gini measurement is the probability of a random sample being classified correctly if we randomly pick a label according to the distribution in a branch. *Gini* impurity can be computed by summing the probability p_i of an item with label i being chosen times the probability $1 - p_i$ of a mistake in categorizing that item over all classes. It reaches its minimum (zero) when all cases in the node belong to a specific category. The *Gini* impurity index for a classification problem with C classes is given by

$$Gini(k) = \sum_{i=1}^C p_i(k)(1 - p_i(k)) \quad (5.8)$$

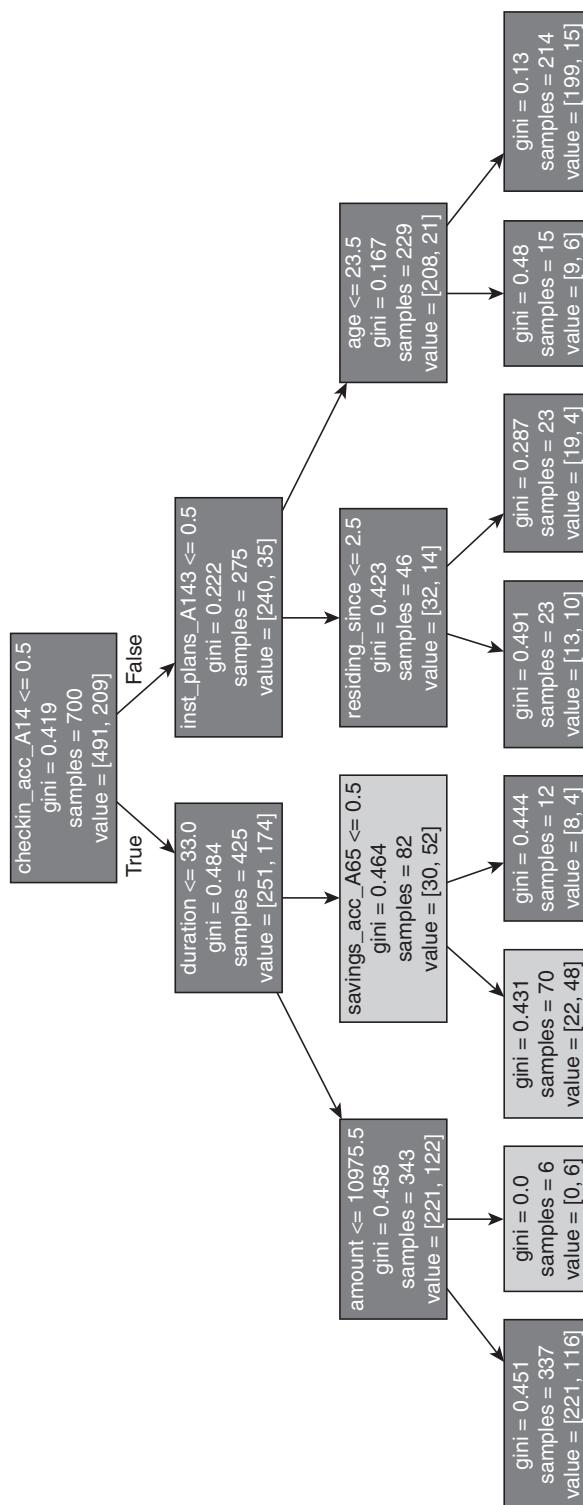


FIGURE 5.9 Decision tree graph created using Gini impurity splitting criteria.

So, in the top node the probability of finding good credit is 491/700 and finding bad credit is 209/700. Gini impurity for the top node in the above decision tree will be

```
gini_node_1 = 1 - pow(491/700, 2) - pow(209/700, 2)
print(round(gini_node_1, 4))
```

0.419

The Gini index is 0.419 as shown in Figure 5.9.

5.5.6 | Building Decision Tree using Entropy Criteria

Entropy is another popular measure of impurity that is used in classification trees to split a node. Assume that there are J classes labelled 1, 2, ..., J . The entropy at node k is given by

$$\text{Entropy}(k) = -\sum_{j=1}^J p(j|k) \log_2(j|k) \quad (5.9)$$

The value of entropy lies between 0 and 1, with a higher entropy indicating a higher impurity at the node. The following codes show how to build a classification tree using entropy splitting criteria and then displaying the tree classifier using Graphviz.

```
clf_tree_entropy = DecisionTreeClassifier(criterion = 'entropy',
                                         max_depth = 3)
clf_tree_entropy.fit(X_train, y_train)

# Export the tree into odt file
export_graphviz(clf_tree_entropy,
                 out_file = "chd_tree_entropy.odt",
                 feature_names = X_train.columns)

# Read the create the image file
chd_tree_graph = pdot.graphviz.graph_from_dot_file('chd_tree_entropy.odt')
chd_tree_graph.write_jpg('chd_tree_entropy.png')
# Render the png file
Image(filename='chd_tree_entropy.png')
```

From the decision tree diagram in Figure 5.10, it can be interpreted that

1. The nodes represented by dark shades depict majority good credits, while the nodes represented by light shades are majority bad credits.
2. The rules generated at the top two level nodes are same as rules generated by decision tree built using Gini index. The rules at the third level nodes are different.
3. One of the rules can be interpreted as: If the customer has *checkin_account_A14*, *Inst_plans* code is A143 and credit history code is A34, then there is a high probability of being good credit. There are 96 customers who satisfy these criteria and 94 of them have good credit.

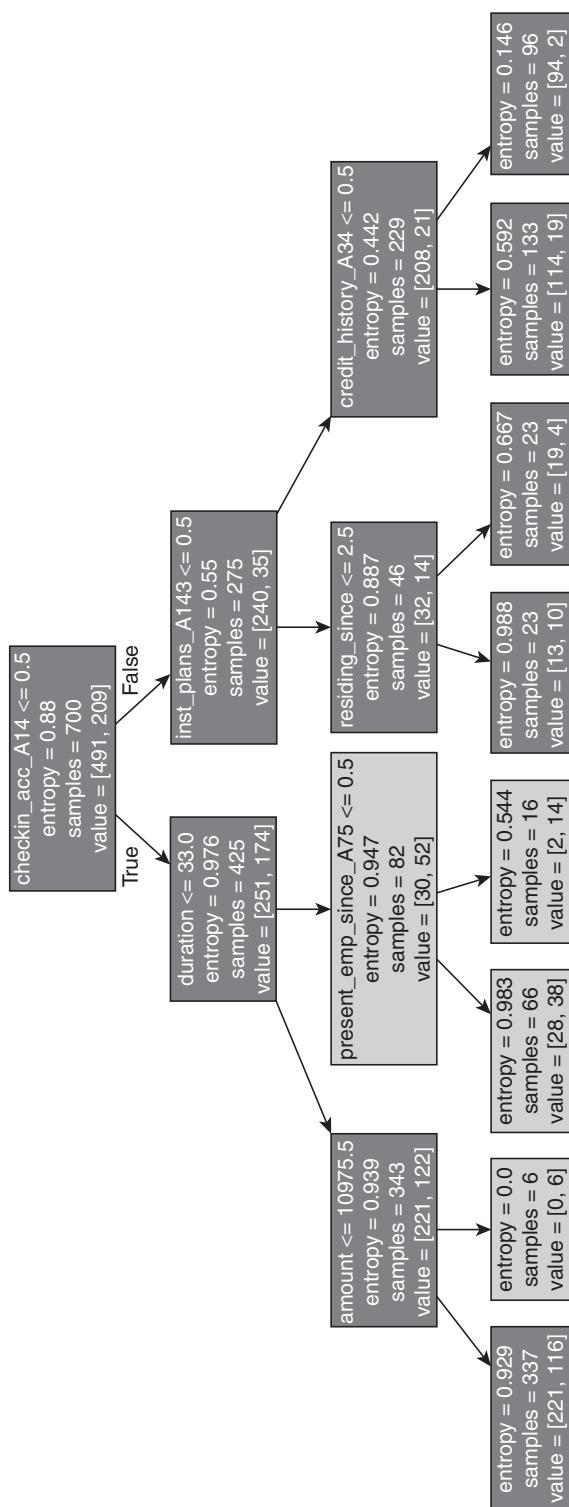


FIGURE 5.10 Decision tree graph created using entropy splitting criteria.

Calculating Entropy

As shown in Figure 5.10, at the top node there are 700 observations of which 491 are good credits and 209 are bad credits. At the top of the node, the probability of finding good credit is 491 and not finding bad credit is 209. So, entropy at the node is given by [as per Eq. (5.9)]

```
import math

entropy_node_1 = - (491/700)*math.log2(491/700) - (209/700)*math.log2(209/700)
print(round(entropy_node_1, 2))
```

0.88

The entropy at the top node is 0.88, as shown in the Figure 5.10.

Measuring Test Accuracy using AUC

```
tree_predict = clf_tree_entropy.predict(X_test)
metrics.roc_auc_score(y_test, tree_predict)
```

0.5763

5.5.7 | Finding Optimal Criteria and Max Depth

We need to search through various combinations of criteria and tree depths to find optimal criteria and depth where the tree gives highest accuracy.

sklearn.model_selection provides a feature called *GridSearchCV* which searches through a set of possible hyperparameter values and reports the most optimal one. *GridSearchCV* needs to be provided with the accuracy measure to be used for searching for the optimal value. For classification models, *roc_auc* values are mostly used. It uses *k*-fold cross validation to measure and validate the accuracy.

GridSearchCV can be used for any machine learning model and can search through multiple hyperparameters of the model. It takes the following arguments:

1. estimator – A scikit-learn model which implements the estimator interface.
2. param_grid – A dictionary with parameter names (string) as keys and lists of parameter settings to try as values.
3. scoring – Is a string; is the accuracy measure, i.e. *roc_auc*.
4. cv – Is an integer; gives the number of folds in *k*-fold. (More on *k*-fold cross-validation in *Chapter 6: Advanced Machine Learning*.)

Configuring the grid search to search for optimal parameters

1. Splitting criteria: gini or entropy.
2. Maximum depth of decision tree ranging from 2 to 10.

The searching of optimal parameter will be validated using 10-fold cross validation and the most optimal parameter will be chosen based on ROC AUC score.

```

from sklearn.model_selection import GridSearchCV
tuned_parameters = [{‘criterion’: [‘gini’, ‘entropy’],
                     ‘max_depth’: range(2,10)}]
clf_tree = DecisionTreeClassifier()
clf = GridSearchCV(clf_tree,
                   tuned_parameters,
                   cv=10,
                   scoring=‘roc_auc’)
clf.fit(X_train, y_train)

GridSearchCV(cv=10, error_score=‘raise’,
             estimator=DecisionTreeClassifier(class_weight=None,
                                              criterion=‘gini’, max_depth=None,
                                              max_features=None, max_leaf_nodes=None,
                                              min_impurity_decrease=0.0, min_impurity_split=None,
                                              min_samples_leaf=1, min_samples_split=2,
                                              min_weight_fraction_leaf=0.0, presort=False, random_state=None,
                                              splitter=‘best’),
             fit_params=None, iid=True, n_jobs=1,
             param_grid=[{‘max_depth’: range(2, 10), ‘criterion’: [‘gini’,
                                              ‘entropy’]}],
             pre_dispatch=‘2*n_jobs’, refit=True, return_train_score=‘warn’,
             scoring=‘roc_auc’, verbose=0)

```

Grid search returns *bestscore* and corresponding *bestparams* after searching through all combinations.

```
clf.best_score_
```

```
0.68242993197278912
```

```
clf.best_params_
```

```
{‘criterion’: ‘gini’, ‘max_depth’: 2}
```

The tree with *gini* criteria and *max_depth* = 2 is the best model. Finally, we can build a model with these parameters and measure the accuracy of the test. This is left to the readers for trying out. Refer to *Section 5.5.2: Building Decision Tree Classifier using Gini Criteria*.

5.5.8 | Benefits of Decision Tree

The benefits of building a decision tree classifier are as follows:

1. Rules generated are simple and interpretable. Trees can be visualized.
2. Work well with both numerical and categorical data. Do not require data to be normalized or creation of dummy variables.
3. Rules can help create business strategies.

CONCLUSION

1. In classification problem, the dependent variable Y takes finite discrete values. Logistic regression is one of the most popular techniques used for solving classification problems.
2. Logistic regression provides the probability of occurrence of the event; final class is usually predicted using a classification cut-off probability.
3. Decision trees generate rules that can be interpreted and business can create strategies around it.
4. The accuracy of an LR model is measured using metrics such as sensitivity, specificity, F-score, precision, and area under the ROC curve. Final model selection may be done using any of these metrics.
5. The optimal classification cut-off probability is usually calculated using Youden's Index or cost-based approach.
6. Gain and lift charts are two approaches used while solving classification problems with imbalanced datasets.

EXERCISES**Answer Questions 1 to 10 using the SAheart Dataset.**

The dataset SAheart.data is taken from the link below

<http://www-stat.stanford.edu/~tibs/ElemStatLearn/datasets/SAheart.data>

The dataset contains records of males in a heart-disease high-risk region of the Western Cape, South Africa. There are roughly two controls per case of CHD. Many of the CHD positive men have undergone blood pressure reduction treatment and other programs to reduce their risk factors after their CHD event. In some cases, the measurements were made after these treatments. These data are taken from a larger dataset, described in Rousseauw et al. (1983), *South African Medical Journal*. It is a tab separated file (.csv) and contains the following columns:

- *sbp* – Systolic blood pressure
- *tobacco* – Cumulative tobacco (kg)
- *ldl* – Low density lipoprotein cholesterol
- *adiposity*
- *famhist* – Family history of heart disease (Present, Absent)
- *typea* – type-A behavior
- *obesity*
- *alcohol* – Current alcohol consumption
- *age* – Age at onset
- *chd* – Response, coronary heart disease

1. Build a logistic regression model to predict the probability of getting *chd* (Coronary Heart Disease), that is, $P(chd = 1)$. Use all other columns except *chd* as features to build the model.
2. Find out which features are statistically significant in the logistic regression model built in Question 1. Build a new logistic regression model using only the significant features.
3. From the logistic regression model parameters, determine which parameters affect the probability of *chd* positively and negatively.

4. Calculate Youden's index for all possible cut-off probabilities ranging from 0.1 to 0.5 with an interval of 0.01. Find the optimal cut-off where Youden's index is maximum.
5. Build a confusion matrix based on the cut-off probability found in Question 4 and report the precision and recall of the model for *chd* cases (i.e., *chd* = 1).
6. Find optimal cut-off probability using cost-based approach using the cost of FPs and FNs as defined below.
 - Cost of predicting “*chd*” as “No *chd*” (FNs) cases is 5 times more than predicting “No *chd*” as “*chd*” (FPs).
 - Then find the precision and recall of the model for *chd* = 1.
7. Build a decision tree classifier model to predict the probability of using Gini index and max depth as 4.
8. Compare the decision tree classifier and logistic regression model performance. Explain which accuracy score can be used to find out which model is better in classifying *chd* from *no chd* cases.
9. Plot the decision tree and find out the most important splitting criteria at the top node. Explain how the Gini index is derived at the top node of the decision tree classifier.
10. Find the optimal *max_depth* for decision tree classifier if Gini index is used as splitting criteria. Search possible depths ranging from 3 to 10 and use ROC and AUC as scoring to find the optimal *max_depth*.

Answer Questions 11 to 15 using the HR Dataset.

The dataset *hr_data.csv* contains samples of candidates that were part of a recruitment process of a particular client of ScaleneWorks. ScaleneWorks supports several information technology (IT) companies in India with their talent acquisition. One of the challenge they face is about 30% of the candidates who accept the jobs offer, do not join the company. This leads to huge loss of revenue and time as the companies initiate the recruitment process again to fill the workforce demand. ScaleneWorks wants to find out if a model can be built to predict the likelihood of a candidate joining the company. If the likelihood is high, then the company can go ahead and offer the jobs to the candidates.

The dataset contains several attributes about candidates along with a column (or variable) that indicates if the candidate finally joined the company or not.

Here is the description of candidate's attributes:

- *Candidate* – Reference number; it is a unique number to identify the candidate
- *DOJ extended* – Binary variable identifying whether candidate asked for date of joining extension (Yes/No)
- *Duration to accept the offer* – Number of days taken by the candidate to accept the offer (Scale variable)
- *Notice period* – Notice period to be served in the parting company before candidate can join this company (Scale variable)
- *Offered band* – Band offered to the candidate based on experience, performance in interview rounds (C0/C1/C2/C3/C4/C5/C6)
- *Percentage hike expected* – Percentage hike expected by the candidate (Scale variable)
- *Percentage hike offered* – Percentage hike offered by the company (Scale variable)
- *Joining bonus* – Binary variable indicating if joining bonus was given or not (Yes/no)
- *Gender* – Gender of the candidate (Male/Female)
- *Candidate source* – Source from which resume of the candidate was obtained (Employee referral/Agency/ Direct)

- *REX* (in Yrs.) – Relevant years of experience of the candidate for the position offered (Scale variable)
 - *LOB* – Line of business for which offer was rolled out (Categorical variable)
 - *Date of Birth* – Date of birth of the candidate
 - *Joining location* – Company location for which the offer was rolled out for the candidate to join (Categorical variable)
 - *Candidate relocation status* – Binary variable indicating whether the candidate has to relocate from one city to another city for joining (Yes/No)
 - *HR Status* – Final joining status of the candidate (Joined/Not Joined)
11. Build a logistic regression model to predict the probability of a candidate joining the company. Assume “Not Joined” as positive cases and “Joined” as negative cases.
12. Find the significant features from the above model and build another logistic regression model with only the significant features.
13. Assume the following costs to find optimal cut-off probability to determine if a candidate will join or not.
- Cost of predicting “Not Joining” as “Joining” (FPs) cases is 3 times more than predicting “Joining” as “Not Joining” (FNs).
14. Build a confusion matrix based on the cut-off probability found in Question 13 and report the precision and recall of the model for joining cases.
15. HR wants to understand the key parameters effecting the joining of candidates. So, build a decision tree with optimal parameters and provide some rules to HR for building strategies to ensure candidates offered job most likely will join the company in future.

REFERENCES

1. U Dinesh Kumar (2017). *Business Analytics: The Science of Data-Driven Decision Making*, Wiley India, India.
2. UCI Machine Learning Repository: <https://archive.ics.uci.edu/ml/>
3. Youden W J (1950). Index for Rating Diagnostic Tests, *Cancer*, Vol. 3, No. 1, pp. 32–35.
4. Scikit-learn documentation: <http://scikit-learn.org/>
5. Statsmodel: <https://www.statsmodels.org/stable/index.html>
6. Pandas Library: <https://pandas.pydata.org/>
7. Matplotlib Library: <https://matplotlib.org/>
8. Seaborn Library: <https://seaborn.pydata.org/>
9. Rousseauw J, du Plessis J, Benade A, Jordaan P, Kotze J, and Ferreira J (1983). Coronary Risk Factor Screening in Three Rural Communities, *South African Medical Journal*, Vol. 64, pp. 430–436.



CHAPTER

6

Advanced Machine Learning

LEARNING OBJECTIVES

After completing this chapter, you will be able to

- Understand the foundations of machine learning algorithms.
- Learn the difference between supervised and unsupervised learning algorithms.
- Understand and develop the gradient descent optimization algorithm.
- Apply machine learning algorithms available in *Scikit-learn (sklearn)* to regression and classification problems.
- Understand the problems of underfitting and overfitting and use of regularization to control overfitting.
- Understanding how to apply ensemble techniques such as Random Forest, Bagging and Boosting.
Know what is a hyperparameter and how to search for optimal hyperparameters.
- Learn feature selection using machine learning models.

6.1 | OVERVIEW

Machine learning algorithms are a subset of artificial intelligence (AI) that imitates the human learning process. Humans learn through multiple experiences how to perform a task. Similarly, machine learning algorithms develop multiple models (usually using multiple datasets) and each model is analogous to an experience. For example, consider someone trying to learn tennis. Getting the service right requires a lot of practice, and more so to learn to serve an ace (serve such that the opponent player is unable to reach the ball). To master the service in tennis (especially ace), a player must probably practice several hundred times; each practice session is a learning. In machine learning algorithms, we develop several models which can run into several hundred and each data and model is treated as learning opportunity. Mitchell (2006) defined machine learning as follows:

Machine learns with respect to a particular task T, performance metric P following experience E, if the system reliably improves its performance P at task T following experience E.

Let the task T be a classification problem. To be more specific, consider a customer's propensity to buy a product. The performance P can be measured through several metrics such as overall accuracy, sensitivity, specificity, and area under the receive operating characteristic curve (AUC). The experience E is analogous to different classifiers generated in machine learning algorithms such as random forest (in random forest several trees are generated, and each tree is used for classification of a new case).

The major difference between statistical learning and machine learning is that statistical learning depends heavily on validation of model assumptions and hypothesis testing, whereas the objective of machine learning is to improve prediction accuracy. For example, while developing a regression model, we check for assumptions such as normality of residuals, significance of regression parameters and so on. However, in the case of the random forest using classification trees, the most important objective is the accuracy/performance of the model.

In this chapter, we will discuss the following two ML algorithms:

- Supervised Learning:** In supervised learning, the datasets have the values of input variables (feature values) and the corresponding outcome variable. The algorithms learn from the training dataset and predict the outcome variable for a new record with values of input variables. Linear regression and logistic regression are examples of supervised learning algorithms.
- Unsupervised Learning:** In this case, the datasets will have only input variable values, but not the output. The algorithm learns the structure in the inputs. Clustering and factor analysis are examples of unsupervised learning and will be discussed in *Chapter 7*.

6.1.1 | How Machines Learn?

In supervised learning, the algorithm learns using a function called loss function, cost function or error function, which is a function of predicted output and the desired output. If $h(X_i)$ is the predicted output and y_i is the desired output, then the loss function is

$$L = \frac{1}{n} \sum_{i=1}^n [h(X_i) - y_i]^2 \quad (6.1)$$

where n is the total number of records for which the predictions are made. The function defined in Eq. (6.1) is a sum of squared error (SSE). SSE is the loss function for a regression model. The objective is to learn the values of parameters (aka feature weights) that minimize the cost function. Machine learning uses optimization algorithms which can be used for minimizing the loss function. Most widely used optimization technique is called the **Gradient Descent**.

In the next section, we will discuss a regression problem and understand how gradient descent algorithm minimizes the loss function and learn the model parameters.

6.2 | GRADIENT DESCENT ALGORITHM

In this section, we will discuss how gradient descent (GD) algorithm can be used for estimating the values of regression parameters, given a dataset with inputs and outputs.

The functional form of a simple linear regression model is given by

$$Y_i = \beta_0 + \beta_1 X_i + \varepsilon_i \quad (6.2)$$

where β_0 is called bias or intercept, β_1 is the feature weight or regression coefficient, ε_i is the error in prediction.

The predicted value of Y_i is written as \widehat{Y}_i and it is given by

$$\widehat{Y}_i = \widehat{\beta}_0 + \widehat{\beta}_1 X_i \quad (6.3)$$

where $\widehat{\beta}_0$ and $\widehat{\beta}_1$ are the estimated values of β_0 and β_1 . The error is given by

$$\epsilon_i = Y_i - \hat{Y}_i = (Y_i - \hat{\beta}_0 - \hat{\beta}_1 X_i) \quad (6.4)$$

The **cost function** for the linear regression model is the total error (mean squared error) across all N records and is given by

$$\text{MSE} = \epsilon_{\text{mse}} = \frac{1}{N} \sum_{i=1}^N (Y_i - \hat{\beta}_0 - \hat{\beta}_1 X_i)^2 \quad (6.5)$$

The error is a function of β_0 and β_1 . It is pure convex function and has a global minimum as shown in Figure 6.1. The gradient descent algorithm starts at a random starting value (for β_0 and β_1) and moves towards the optimal solution as shown in Figure 6.1.

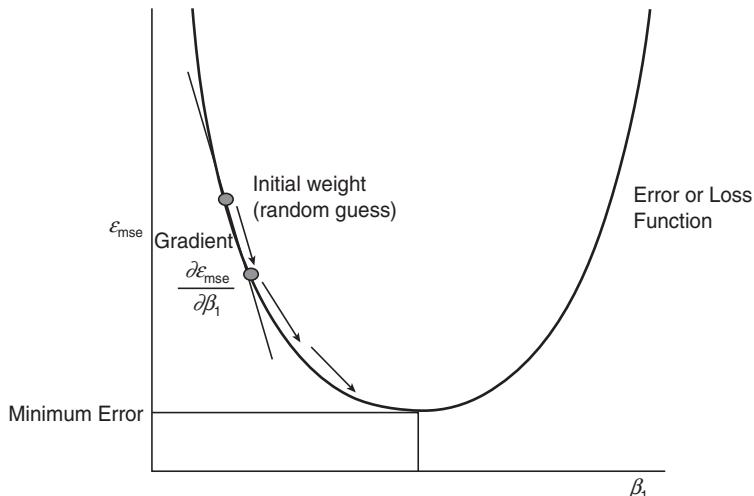


FIGURE 6.1 Cost function for linear regression model.

Gradient descent finds the optimal values of β_0 and β_1 that minimize the loss function using the following steps:

1. Randomly guess the initial values of β_0 (bias or intercept) and β_1 (feature weight).
2. Calculate the estimated value of the outcome variable \hat{Y}_i for initialized values of bias and weights.
3. Calculate the mean square error function (MSE).
4. Adjust the β_0 and β_1 values by calculating the gradients of the error function.

$$\beta_0 = \beta_0 - \alpha \times \frac{\partial \epsilon_{\text{mse}}}{\partial \beta_0} \quad (6.6)$$

$$\beta_1 = \beta_1 - \alpha \times \frac{\partial \epsilon_{\text{mse}}}{\partial \beta_1} \quad (6.7)$$

where α is the learning rate (a hyperparameter). The value of α is chosen based on the magnitude of update needed to be applied to the bias and weights at each iteration.

The partial derivatives of MSE with respect to β_0 and β_1 are given by

$$\frac{\partial \mathcal{E}_{\text{mse}}}{\partial \beta_0} = -\frac{2}{N} \sum_{i=1}^N (Y_i - \beta_0 - \beta_1 X_i) = -\frac{2}{N} \sum_{i=1}^N (Y_i - \hat{Y}_i) \quad (6.8)$$

$$\frac{\partial \mathcal{E}_{\text{mse}}}{\partial \beta_1} = -\frac{2}{N} \sum_{i=1}^N (Y_i - \beta_0 - \beta_1 X_i) = -\frac{2}{N} \sum_{i=1}^N (Y_i - \hat{Y}_i) \times X_i \quad (6.9)$$

5. Repeat steps 1 to 4 for several iterations until the error stops reducing further or the change in cost is infinitesimally small.

The values of β_0 and β_1 at the minimum cost points are best estimates of the model parameters.

6.2.1 | Developing a Gradient Descent Algorithm for Linear Regression Model

For better understanding the GD algorithm, we will implement the GD algorithm using the dataset *Advertising.csv*. The dataset contains the examples of advertisement spends across multiple channels such as Radio, TV, and Newspaper, and the corresponding sales revenue generated at different time periods. The dataset is taken from a chapter in the book titled “*Introduction to Statistical Learning*” by James *et al.* (2013). The dataset has the following elements:

1. TV – Spend on TV advertisements
2. Radio – Spend on radio advertisements
3. Newspaper – Spend on newspaper advertisements
4. Sales – Sales revenue generated

For predicting future sales using spends on different advertisement channels, we can build a regression model.

6.2.1.1 Loading the Dataset

Load the dataset using Pandas’ library.

```
import pandas as pd
import numpy as np

import warnings
warnings.filterwarnings('ignore')

sales_df = pd.read_csv('Advertising.csv')
# Printing first few records
sales_df.head()
```

	Unnamed: 0	TV	Radio	Newspaper	Sales
0	1	230.1	37.8	69.2	22.1
1	2	44.5	39.3	45.1	10.4
2	3	17.2	45.9	69.3	9.3
3	4	151.5	41.3	58.5	18.5
4	5	180.8	10.8	58.4	12.9

6.2.1.2 Set X and Y Variables

For building a regression model, the inputs *TV*, *Radio*, and *Newspaper* are taken as *X* features and *Sales* *Y* is taken as the outcome variable.

```
X = sales_df[['TV', 'Radio', 'Newspaper']]
Y = sales_df['Sales']
```

6.2.1.3 Standardize X and Y

It is important to convert all variables into one scale. This can be done by subtracting mean from each value of the variable and dividing by the corresponding standard deviation of the variable.

```
Y = np.array( (Y - Y.mean()) / Y.std() )
X = X.apply( lambda rec: (rec - rec.mean()) / rec.std(),
            axis = 0 )
```

6.2.1.4 Implementing the Gradient Descent Algorithm

To implement the steps explained in Section 6.2, *Gradient Descent*, a set of following utility methods need to be implemented:

1. **Method 1:** Method to randomly initialize the bias and weights.
2. **Method 2:** Method to calculate the predicted value of *Y*, that is, *Y* given the bias and weights.
3. **Method 3:** Method to calculate the cost function from predicted and actual values of *Y*.
4. **Method 4:** Method to calculate the gradients and adjust the bias and weights.

Method 1: Random Initialization of the Bias and Weights

The method randomly initializes the bias and weights. It takes the number of weights that need to be initialized as a parameter.

```

import random

#dim - is the number of weights to be initialized besides the bias
def initialize( dim ):
    # For reproducible results, the seed is set to 42.
    # Reader can comment the following two lines
    # and try other initialization values.
    np.random.seed(seed=42)
    random.seed(42)
    #Initialize the bias.
    b = random.random()
    #Initialize the weights.
    w = np.random.rand( dim )

    return b, w

```

To initialize the bias and 3 weights, as we have three input variables *TV*, *Radio* and *Newspaper*, we can invoke the *initialize()* method as follows:

```

b, w = initialize( 3 )
print( "Bias: ", b, " Weights: ", w )

Bias: 0.6394267984    Weights: [0.37454012 0.95071431 0.73199394]

```

Method 2: Predict YValues from the Bias and Weights

Calculate the *Y* values for all the inputs, given the bias and weights. We will use matrix multiplication of weights with input variable values. *matmul()* method in *numpy* library can be used for matrix multiplication. Each row of *X* can be multiplied with the weights column to produce the predicted outcome variable.

```

# Inputs:
# b - bias
# w - weights
# X - the input matrix

def predict_Y( b, w, X ):
    return b + np.matmul( X, w )

```

Now calculate the predicted values after initializing bias and weights.

```

b, w = initialize( 3 )
Y_hat = predict_Y( b, w, X )
Y_hat[0:10]

array([ 3.23149557,  1.70784873,  2.82476076,  2.75309026,  0.92448558,
       3.17136498,  0.62234399, -0.34935444, -2.313095,   -0.76802983])

```

Method 3: Calculate the Cost Function – MSE

Compute mean squared error (MSE) by

1. Calculating differences between the estimated \hat{Y} and actual Y .
2. Calculating the square of the above residuals, and sum over all records.
3. Dividing it with the number of observations.

```
import math

# Inputs
# Y - Actual values of y
# Y_hat - predicted value of y
def get_cost( Y, Y_hat ):
    # Calculating the residuals - difference between actual and
    # predicted values
    Y_resid = Y - Y_hat
    # Matrix multiplication with self will give the square values
    # Then take the sum and divide by number of examples to
    # calculate mean
    return np.sum( np.matmul( Y_resid.T, Y_resid ) ) / len( Y_resid )
```

Invoking `get_cost()` after initializing the bias and weights and calculating predicted values for outcome variable.

```
b, w = initialize( 3 )
Y_hat = predict_Y( b, w, X)
get_cost( Y, Y_hat )
```

1.5303100198505895

Method 4: Update the Bias and Weights

This is the most important method, where the bias and weights are adjusted based on the gradient of the cost function. The bias and weights will be updated in a method `update_beta()` using the following gradients [Eqs. (6.6) and (6.7)]:

$$\beta_0 = \beta_0 - \alpha \times \frac{\partial \mathcal{E}_{\text{mse}}}{\partial \beta_0}$$

$$\beta_1 = \beta_1 - \alpha \times \frac{\partial \mathcal{E}_{\text{mse}}}{\partial \beta_1}$$

where α is the learning parameter that decides the magnitude of the update to be done to the bias and weights.

The parameters passed to the function are:

1. x, y : the input and output variables
2. y_{hat} : predicted value with current bias and weights
3. b_0, w_0 : current bias and weights
4. *learning rate*: learning rate to adjust the update step

```
def update_beta( x, y, y_hat, b_0, w_0, learning_rate ):

    #gradient of bias
    db = (np.sum( y_hat - y ) * 2) / len(y)
    #gradient of weights
    dw = (np.dot( ( y_hat - y ), x ) * 2 ) / len(y)
    #update bias
    b_1 = b_0 - learning_rate * db
    #update beta
    w_1 = w_0 - learning_rate * dw

    #return the new bias and beta values
    return b_1, w_1
```

The following is an example of updating bias and weights once after initializing. The learning parameter used is 0.01.

```
b, w = initialize( 3 )
print( "After Initialization - Bias: ", b, " Weights: ", w )
Y_hat = predict_Y( b, w, X )
b, w = update_beta( X, Y, Y_hat, b, w, 0.01 )
print( "After first update - Bias: ", b, " Weights: ", w )
```

```
After initialization - Bias: 0.6394267984578837 Weights: [0.37454
012 0.95071431 0.73199394]
After first update - Bias: 0.6266382624887261 Weights: [0.3807909
3 0.9376953 0.71484883]
```

6.2.1.5 Finding the Optimal Bias and Weights

The updates to the bias and weights need to be done iteratively, until the cost is minimum. It can take several iterations and is time-consuming. There are two approaches to stop the iterations:

1. Run a fixed number of iterations and use the bias and weights as optimal values at the end these iterations.
2. Run iterations until the change in cost is small, that is, less than a predefined value (e.g., 0.001).

We will define a method *run_gradient_descent()*, which takes *alpha* and *num_iterations* as parameters and invokes methods like *initialize()*, *predict_Y()*, *get_cost()*, and *update_beta()*.

Also, inside the method,

1. variable `gd_iterations_df` keeps track of the cost every 10 iterations.
2. default value of 0.01 for the learning parameter and 100 for number of iterations will be used.

```
def run_gradient_descent( X,
                         Y,
                         alpha = 0.01,
                         num_iterations = 100):

    # Initialize the bias and weights
    b, w = initialize( X.shape[1] )

    iter_num = 0
    # gd_iterations_df keeps track of the cost every 10 iterations
    gd_iterations_df = pd.DataFrame(columns = ['iteration', 'cost'])
    result_idx = 0

    # Run the iterations in loop
    for each_iter in range(num_iterations):
        # Calculate predicted value of y
        Y_hat = predict_Y( b, w, X )
        # Calculate the cost
        this_cost = get_cost( Y, Y_hat )
        # Save the previous bias and weights
        prev_b = b
        prev_w = w
        # Update and calculate the new values of bias and weights
        b, w = update_beta( X, Y, Y_hat, prev_b, prev_w, alpha)
        # For every 10 iterations, store the cost i.e. MSE
        if( iter_num % 10 == 0 ):
            gd_iterations_df.loc[result_idx] = [iter_num, this_cost]
            result_idx = result_idx + 1

        iter_num += 1

    print( "Final estimate of b and w: ", b, w )
    #return the final bias, weights and the cost at the end
    return gd_iterations_df, b, w
```

```
gd_iterations_df, b, w = run_gradient_descent( X, Y, alpha =
0.001, num_iterations = 200 )
```

Final estimate of b and w: 0.42844895817391493 [0.48270238 0.752659
69 0.46109174]

Let us print the cost per every 10 iterations.

```
gd_iterations_df[0:10]
```

	iteration	cost
0	0.0	1.530310
1	10.0	1.465201
2	20.0	1.403145
3	30.0	1.343996
4	40.0	1.287615
5	50.0	1.233868
6	60.0	1.182630
7	70.0	1.133780
8	80.0	1.087203
9	90.0	1.042793

It can be noted that the cost is reducing at the end of each iteration.

6.2.1.6 Plotting the Cost Function against the Iterations

The value of the cost function at the end of each iteration can be visualized by plotting cost at every 10 iterations using the following commands:

```
import matplotlib.pyplot as plt
import seaborn as sn
%matplotlib inline

plt.plot( gd_iterations_df['iteration'], gd_iterations_df['cost'] );
plt.xlabel("Number of iterations")
plt.xlabel("Cost or MSE")

Text(0.5, 0,'Cost or MSE')

print( "Final estimates of b and w: ", b, w )
```

```
Final estimates of b and w: 0.42844895817391493 [0.48270238 0.75265
969 0.46109174]
```

The weights are standardized values as we have used standardized values of Y and X. This means that for one standard deviation change in TV spend, the sales revenue changes by 1.55599903 standard deviations and so on.

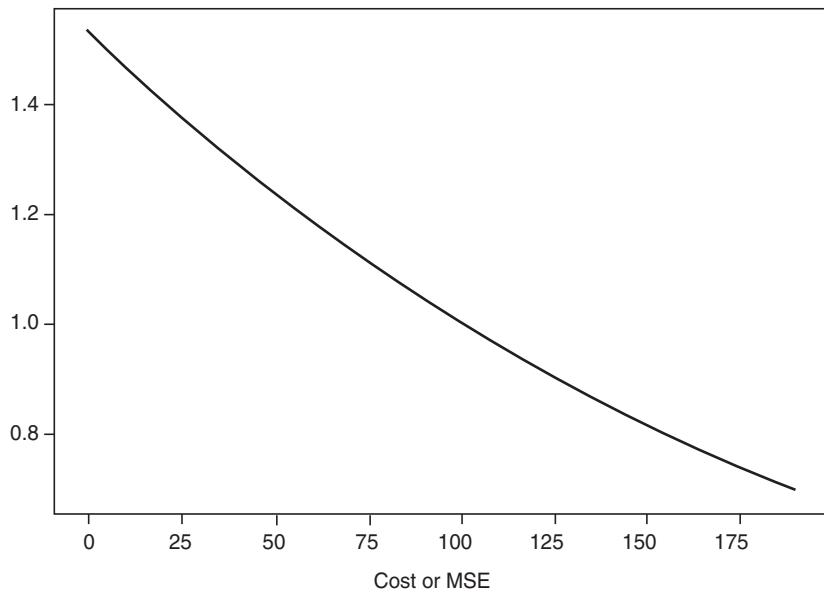


FIGURE 6.2 Cost or MSE at the end of iterations.

From Figure 6.2, it can be noticed that the cost is still reducing and has not reached the minimum point. We can run more iterations and verify if the cost is reaching a minimum point or not.

```
alpha_df_1, b, w = run_gradient_descent(X, Y, alpha = 0.01,
                                         num_iterations = 2000)
```

```
Final estimate of b and w: 2.7728016698178713e-16
[ 0.75306591  0.5 3648155 -0.00433069]
```

What happens if we change the learning parameter and use smaller value (e.g., 0.001)?

```
alpha_df_2, b, w = run_gradient_descent(X, Y, alpha = 0.001,
                                         num_iterations = 2000)
```

```
Final estimate of b and w: 0.011664695556930518 [0.74315125 0.52779
959 0.01171703]
```

Now we plot the cost after every iteration for different learning rate parameters (alpha values).

```
plt.plot(alpha_df_1['iteration'], alpha_df_1['cost'], label =
          "alpha = 0.01");
plt.plot(alpha_df_2['iteration'], alpha_df_2['cost'], label =
          "alpha = 0.001");
```

```

plt.legend();
plt.ylabel('Cost');
plt.xlabel('Number of Iterations');
plt.title('Cost Vs. Iterations for different alpha values');

```

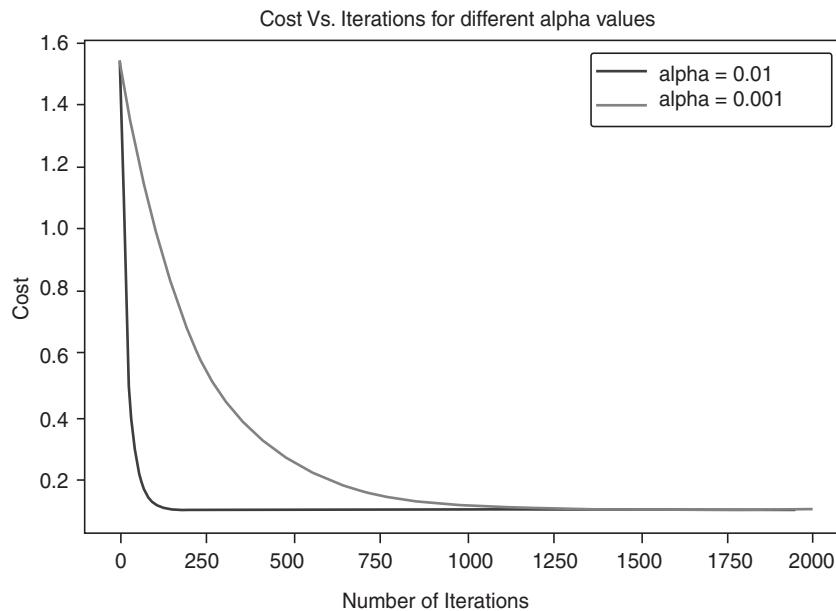


FIGURE 6.3 Cost or MSE at the end of iterations for different alpha values.

The plot in Figure 6.3 shows that the learning is faster for *alpha* value 0.01 compared to 0.001. For smaller values, the learning could be slower whereas higher learning rate could lead to skipping the minima of cost function. It is imperative to search for the optimal learning parameter.

6.3 | SCIKIT-LEARN LIBRARY FOR MACHINE LEARNING

The implementation of gradient descent algorithm in the previous section using *numpy* library is only for our understanding of how it works. But in practice, we will use the *scikit-learn* library in Python, which is primarily an open-source Python library for building machine learning models. *scikit-learn* provides a comprehensive set of algorithms for the following kind of problems:

1. Regression
2. Classification
3. Clustering

scikit-learn also provides an extensive set of methods for data pre-processing and feature selection. We will discuss many algorithms in this chapter that are available within *scikit-learn*. *scikit-learn* will be referred as *sklearn* going forward.

6.3.1 | Steps for Building Machine Learning Models

The steps to be followed for building, validating a machine learning model and measuring its accuracy are as follows:

1. Identify the features and outcome variable in the dataset.
2. Split the dataset into training and test sets.
3. Build the model using training set.
4. Predict outcome variable using a test set.
5. Compare the predicted and actual values of the outcome variable in the test set and measure accuracy using measures such as mean absolute percentage error (MAPE) or root mean square error (RMSE).

6.3.1.1 *Splitting Dataset into Train and Test Datasets*

The following commands can be used for splitting the dataset using 70:30 ratio. That is, 70% for training and 30% for the test set. Usually, data scientists use a range of 60% to 80% for training and the rest for testing the model.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    sales_df[["TV", "Radio",
              "Newspaper"]], 
    sales_df.Sales, test_size=0.3,
    random_state = 42)
```

The following commands can be used for finding the number of records sampled into training and test sets.

```
len( X_train )
```

140

```
len( X_test )
```

60

6.3.1.2 *Building Linear Regression Model with Train Dataset*

Linear models are included in *sklearn.linear_model* module. We will use *LinearRegression* method for building the model and compare with the results we obtained through our own implementation of gradient descent algorithm.

```
from sklearn.linear_model import LinearRegression
```

Steps for building a model in *sklearn* are

1. Initialize the model.
2. Invoke *fit()* method on the model and pass the input (*X*) and output(*Y*) values.
3. *fit()* will run the algorithm and return the final estimated model parameters.

```
# Initializing the model
linreg = LinearRegression()
# Fitting training data to the model
linreg.fit( X_train, y_train )
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1,
normalize=False)
```

After the model is built, the model parameters such as intercept (bias) and coefficients (weights) can be obtained using the following commands:

```
linreg.intercept_
```

```
2.708949092515912
```

```
linreg.coef_
```

```
array([0.04405928, 0.1992875, 0.00688245])
```

To associate the coefficient values with the variable names, we can use *zip()* in Python. *zip()* returns a dictionary with variable names mapped to coefficient values.

```
list( zip( ["TV", "Radio", "Newspaper"], list( linreg.coef_ ) ) )
```

```
[('TV', 0.04405928),
 ('Radio', 0.1992874968989395),
 ('Newspaper', 0.0068824522222754)]
```

The estimated model is

$$Sales = 2.708 + 0.044 * TV + 0.199 * Radio + 0.006 * Newspaper$$

The weights are different than what we estimated earlier. This is because we have not standardized the values in this model. The model indicates that for every unit change in TV spending, there is an increase of 0.044 units in sales revenue.

6.3.1.3 Making Prediction on Test Set

sklearn provides *predict()* method for all ML models, which takes the *X* values and predicts the outcome variable *Y* as shown in the following code:

```
# Predicting the y value from the test set
y_pred = linreg.predict( X_test )
```

To compare the actual and predicted values of the outcome variable and the residuals, we will create and store these values in a DataFrame. The residual here refers to the difference between the actual and the predicted values.

```
# Creating DataFrame with 3 columns named: actual, predicted and residuals
# to store the respective values
test_pred_df = pd.DataFrame( { 'actual': y_test,
                               'predicted': np.round( y_pred, 2 ),
                               'residuals': y_test - y_pred } )
# Randomly showing the 10 observations from the DataFrame
test_pred_df.sample(10)
```

	actual	predicted	residuals
126	6.6	11.15	-4.553147
170	8.4	7.35	1.049715
95	16.9	16.57	0.334604
195	7.6	5.22	2.375645
115	12.6	13.36	-0.755569
38	10.1	10.17	-0.070454
56	5.5	8.92	-3.415494
165	11.9	14.30	-2.402060
173	11.7	11.63	0.068431
9	10.6	12.18	-1.576049

6.3.1.4 Measuring Accuracy

Root Mean Square Error (RMSE) and R-squared are two key accuracy measures for *Linear Regression Models* as explained in Chapter 4.

sklearn.metrics package provides methods to measure various metrics. For regression models, *mean_squared_error* and *r2_score* can be used to calculate MSE and R-squared values, respectively.

```
## Importing metrics from sklearn
from sklearn import metrics
```

R-Squared Value

The R-squared value is calculated on the training data, which is the amount of variance in Y that can be explained by the model. `metrics.r2_score()` takes the actual and the predicted values of Y to compute R-squared value. The following command can be used for calculating R-squared value:

```
# y_train contains the actual value and the predicted value is
# returned from predict() method after passing the X values of the
# training data.
r2 = metrics.r2_score( y_train, linreg.predict(X_train) )
print("R Squared: ", r2)
```

R Squared: 0.9055159502227753

The model explains 90% of the variance in Y .

RMSE Calculation

`metrics.mean_squared_error()` takes actual and predicted values and returns MSE.

```
# y_pred contains predicted value of test data
mse = metrics.mean_squared_error( y_test, y_pred )
```

RMSE value can be calculated by the square root of `mse` using the following command:

```
# Taking square root of MSE and then round off to two decimal values
rmse = round( np.sqrt(mse), 2 )
print("RMSE: ", rmse)
```

RMSE: 1.95

The RMSE value indicates the model prediction has a standard deviation of 1.95.

To understand the model error in detail, we need to understand the components of the error term and how to deal with those components for improving model performance. In the next section, we will discuss these components in detail.

6.3.2 | Bias-Variance Trade-off

Model errors can be decomposed into two components: *bias* and *variance*. Understanding these two components is key to diagnosing model accuracies and avoiding model overfitting or underfitting. High bias can lead to building underfitting model, whereas high variance can lead to overfitting models.

Let us take an example to understand bias and variance in detail. The dataset *curve.csv* has records with two variables *x* and *y*, where *y* depends on *x*.

```
# Reading the file curve.csv and printing first few examples
curve = pd.read_csv( "curve.csv" )
curve.head()
```

	<i>x</i>	<i>y</i>
0	2	-1.999618
1	2	-1.999618
2	8	-3.978312
3	9	-1.969175
4	10	-0.957770

As there are only two variables, we can observe the relationship by plotting them using scatter plot (Figure 6.4).

```
plt.scatter( curve.x, curve.y );
plt.xlabel("x values")
plt.ylabel("y values")
```

```
Text(0,0.5,'y values')
```

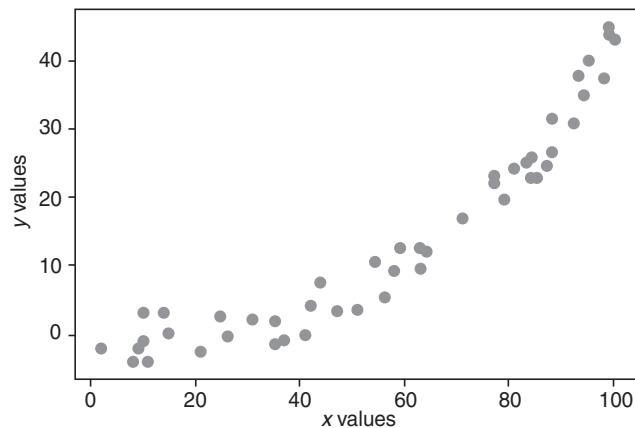


FIGURE 6.4 Scatter plot between *x* and *y*.

It can be observed from Figure 6.4 that the relation between *y* and *x* is not linear and looks like some polynomial. But we are not sure of the degree of the polynomial form. We need to try various polynomial forms of *x* and verify which model fits the data best.

To explore various polynomial forms, `polyfit()` from `numpy` library can be used. `polyfit()` takes X and Y values, and the degree of x features to be used to fit a model. Degree 1 means only value of x is used to predict y, whereas degree 2 means x and x^2 are used to predict y.

The following commands can be used for implementing a generic method `fit_poly()`, which takes degree as a parameter and builds a model with all required polynomial terms.

```
# Input
# degree - polynomial terms to be used in the model
def fit_poly( degree ):
    # calling numpy method polyfit
    p = np.polyfit( curve.x, curve.y, deg = degree )
    curve['fit'] = np.polyval( p, curve.x )
    # draw the regression line after fitting the model
    sn.regplot( curve.x, curve.y, fit_reg = False )
    # Plot the actual x and y values
    return plt.plot( curve.x, curve.fit, label='fit' )
```

Fitting a mode with degree = 1.

$$y = \beta_1 x_1 + \varepsilon$$

```
fit_poly( 1 );
## Plotting the model form and the data
plt.xlabel("x values")
plt.ylabel("y values");
```

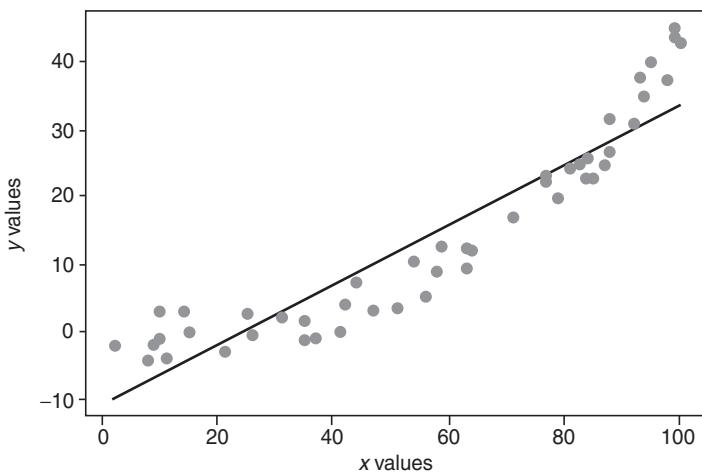


FIGURE 6.5 Linear regression model between x and y.

Linear model ($y = \beta_0 + \beta_1 x_1$) is shown in Figure 6.5. It is evident from the figure that the regression model does not seem to fit the data well. It is not able to learn from data as a simplistic form of model is assumed. This is a case of underfitting or bias.

Let us fit a regression model with a polynomial term, that is, square of x .

$$y = \beta_1 x_1 + \beta_2 x_1^2 + \varepsilon_i$$

```
fit_poly( 2 );
plt.xlabel("x values")
plt.ylabel("y values");
```

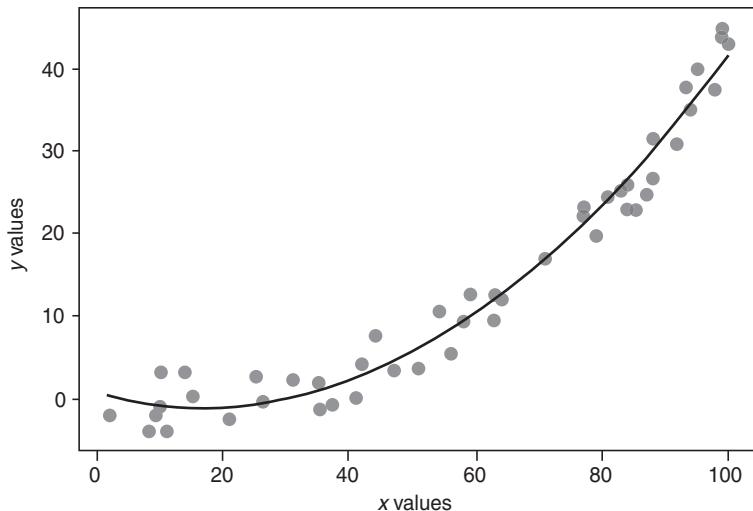


FIGURE 6.6 Linear regression model between y and polynomial terms of x of degrees 1 and 2.

Figure 6.6 shows the fitted model which includes x_1 and x_1^2 . The regression line seems to fit data better than the previous model. But we are not sure if this is the best model. We can try building a model with all polynomial terms up to degrees ranging from 1 to 10 as shown below:

$$y = \beta_1 x_1 + \beta_2 x_1^2 + \beta_3 x_1^3 + \beta_4 x_1^4 + \beta_5 x_1^5 + \beta_6 x_1^6 + \beta_7 x_1^7 + \beta_8 x_1^8 + \beta_9 x_1^9 + \beta_{10} x_1^{10} + \varepsilon_i$$

```
fit_poly( 10 );
plt.xlabel("x values")
plt.ylabel("y values");
```

It can be observed from Figure 6.7 that as we build models with more and more polynomial terms, the model starts to fit every data point in the training set. This is the case of overfitting. The model with polynomial terms up to degree 10 will be sensitive to any changes in training examples. Any addition and removal of a single observation from the dataset can alter the model parameters significantly.

An underfitting model has a large error because of high bias, and an overfitting model has a large error because of high variance. An optimal model will be somewhere between an underfitting and an overfitting model, and will have low bias and low variance. This can be observed by comparing RMSE in training and test sets.

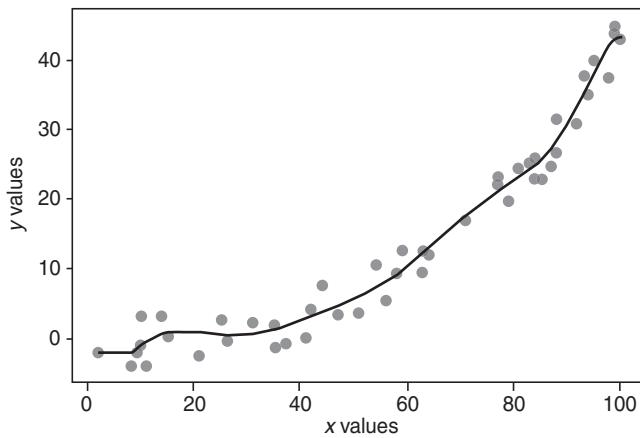


FIGURE 6.7 Linear regression model between y and polynomial terms of x of degree ranging from 1 to 10.

We use the following code to build models with degrees ranging from 1 to 15 and storing the degree and error details in different columns of a DataFrame named `rmse_df`.

1. `degree`: Degree of the model (number of polynomial terms).
2. `rmse_train`: RMSE error on train set.
3. `rmse_test`: RMSE error on test set.

```
# Split the dataset into 60:40 into training and test set
train_X, test_X, train_y, test_y = train_test_split( curve.x,
                                                    curve.y,
                                                    test_size=0.40,
                                                    random_state=100 )

# Define the dataframe store degree and rmse for training and test set
rmse_df = pd.DataFrame( columns = ["degree", "rmse_train",
                                    "rmse_test"] )

# Define a method to return the rmse given actual and predicted values.

def get_rmse( y, y_fit ):
    return np.sqrt( metrics.mean_squared_error( y, y_fit ) )

# Iterate from degree 1 to 15
for i in range( 1, 15 ):
    # fitting model
    p = np.polyfit( train_X, train_y, deg = i )
    # storing model degree and rmse on train and test set
    rmse_df.loc[i-1] = [ i,
                        get_rmse(train_y, np.polyval(p, train_X)),
                        get_rmse(test_y, np.polyval(p, test_X)) ]
```

Print the *rmse_df* records using the following code:

```
rmse_df
```

	degree	rmse_train	rmse_test
0	1.0	5.226638	5.779652
1	2.0	2.394509	2.755286
2	3.0	2.233547	2.560184
3	4.0	2.231998	2.549205
4	5.0	2.197528	2.428728
5	6.0	2.062201	2.703880
6	7.0	2.039408	2.909237
7	8.0	1.995852	3.270892
8	9.0	1.979322	3.120420
9	10.0	1.976326	3.115875
10	11.0	1.964484	3.218203
11	12.0	1.657948	4.457668
12	13.0	1.656719	4.358014
13	14.0	1.642308	4.659503

Now we can plot the train and test errors against the degree of the models for better understanding.

```
# Plotting the rmse for training set in red color
plt.plot( rmse_df.degree,
          rmse_df.rmse_train,
          label='RMSE on Training Set',
          color = 'r' )

# Plotting the rmse for test set in green color
plt.plot( rmse_df.degree,
          rmse_df.rmse_test,
          label='RMSE on Test Set',
          color = 'g' )

# Mention the legend
plt.legend(bbox_to_anchor=(1.05, 1),
           loc=2,
           borderaxespad=0.);

plt.xlabel("Model Degrees")
plt.ylabel("RMSE");
```

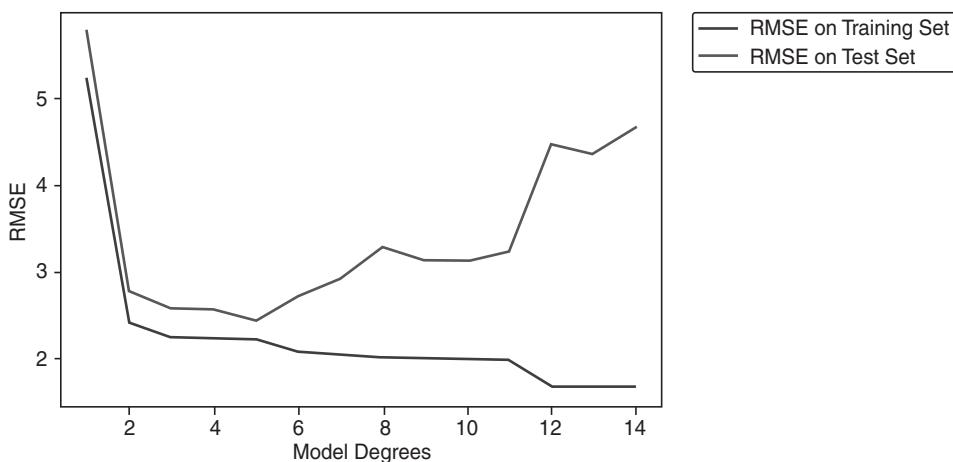


FIGURE 6.8 Model accuracy on training and test sets against the model complexity.

Three key observations from Figure 6.8 are as follows:

1. Error on the test set are high for the model with complexity of degree 1 and degree 15.
2. Error on the test set reduces initially, however increases after a specific level of complexity.
3. Error on the training set decreases continuously.

The degree can also be associated with complexity. It can be observed that as complexity (number of polynomial terms in the model) increases, the model starts to fit to train and test data well. But beyond a specific level of complexity (i.e., complexity level 5), the training error continues to reduce, but the test error starts to increase again. This is a point where the model starts to overfit the training set and stops generalizing. So, the optimal model complexity is 5, which has low bias and low variance.

6.3.3 | K-Fold Cross-Validation

K-fold cross-validation is a robust validation approach that can be adopted to verify if the model is overfitting. The model, which generalizes well and does not overfit, should not be very sensitive to any change in underlying training samples. K-fold cross-validation can do this by building and validating multiple models by resampling multiple training and validation sets from the original dataset.

The following steps are used in K-fold cross-validation:

1. Split the training data set into K subsets of equal size. Each subset will be called a fold. Let the folds be labelled as f_1, f_2, \dots, f_K . Generally, the value of K is taken to be 5 or 10.
2. For $i = 1$ to K
 - (a) Fold f_i is used as validation set and all the remaining $K - 1$ folds as training set.
 - (b) Train the model using the training set and calculate the accuracy of the model in fold f_i .

Calculate the final accuracy by averaging the accuracies in the test data across all K models. The average accuracy value shows how the model will behave in the real world. The variance of these accuracies is an indication of the robustness of the model.

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
Iteration 1	train	train	train	train	test
Iteration 2	train	train	train	test	train
Iteration 3	train	train	test	train	train
Iteration 4	train	test	train	train	train
Iteration 5	test	train	train	train	train

FIGURE 6.9 K-fold cross-validation.

Figure 6.9 depicts the process of K-fold cross-validation with 5 folds.

6.4 | ADVANCED REGRESSION MODELS

We will be using the IPL dataset (used in Chapter 4) to discuss advanced regression models. First, we will build a linear regression model to understand the shortcomings and then proceed to advanced regression models.

6.4.1 | Building Linear Regression Model

The model will predict *SOLD PRICE* of a player based on past performance measures of the players.

6.4.1.1 Loading IPL Dataset

Load the dataset and display information about the dataset using the following commands:

```
ipl_auction_df = pd.read_csv('IPL IMB381IPL2013.csv')
ipl_auction_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 130 entries, 0 to 129
Data columns (total 26 columns):
Sl.NO.          130    non-null   int64
PLAYER NAME     130    non-null   object
AGE             130    non-null   int64
COUNTRY         130    non-null   object
TEAM            130    non-null   object
PLAYING ROLE   130    non-null   object
T-RUNS          130    non-null   int64
T-WKTS          130    non-null   int64
ODI-RUNS-S     130    non-null   int64
ODI-SR-B       130    non-null   float64
```

```

ODI-WKTS      130    non-null   int64
ODI-SR-BL     130    non-null   float64
CAPTAINCY EXP 130    non-null   int64
RUNS-S        130    non-null   int64
HS            130    non-null   int64
AVE           130    non-null   float64
SR-B          130    non-null   float64
SIXERS        130    non-null   int64
RUNS-C        130    non-null   int64
WKTS          130    non-null   int64
AVE-BL         130    non-null   float64
ECON           130    non-null   float64
SR-BL          130    non-null   float64
AUCTION YEAR 130    non-null   int64
BASE PRICE    130    non-null   int64
SOLD PRICE    130    non-null   int64
dtypes: float64(7), int64(15), object(4)
memory usage: 26.5+ KB

```

We will use only a subset of features for building the model. The list *X_features* is initialized with the name of the features to be used as described below.

```

X_features = ['AGE', 'COUNTRY', 'PLAYING ROLE', 'T-RUNS', 'T-WKTS',
               'ODI-RUNS-S', 'ODI-SR-B', 'ODI-WKTS', 'ODI-SR-BL',
               'CAPTAINCY EXP', 'RUNS-S', 'HS', 'AVE', 'SR-B', 'SIX-
               ERS', 'RUNS-C', 'WKTS', 'AVE-BL', 'ECON', 'SR-BL']

```

Out of these, there are four categorical features that need to be encoded into dummy features using OHE (One Hot Encoding). The details of encoding categorical features are already discussed in Chapter 4.

```

# Initialize a list with the categorical feature names.
categorical_features = ['AGE', 'COUNTRY', 'PLAYING ROLE',
                        'CAPTAINCY EXP']
# get_dummies() is invoked to return the dummy features.
ipl_auction_encoded_df = pd.get_dummies( ipl_auction_df[X_features],
                                           columns = categorical_
                                           features,
                                           drop_first = True )

```

To display all feature names along with the new dummy features we use the following code:

```
ipl_auction_encoded_df.columns
```

```
Index(['T-RUNS', 'T-WKTS', 'ODI-RUNS-S', 'ODI-SR-B', 'ODI-WKTS',
       'ODI-SR-BL', 'RUNS-S', 'HS', 'AVE', 'SR-B', 'SIXERS', 'RUNS-',
       'WKTS', 'AVE-BL', 'ECON', 'SR-BL', 'AGE_2', 'AGE_3',
       'COUNTRY_BAN', 'COUNTRY_ENG', 'COUNTRY_IND', 'COUNTRY_NZ',
       'COUNTRY_PAK', 'COUNTRY_SA', 'COUNTRY_SL', 'COUNTRY_WI',
       'COUNTRY_ZIM', 'PLAYING ROLE_Batsman', 'PLAYING ROLE_Bowler',
       'PLAYING ROLE_W. Keeper', 'CAPTAINCY EXP_1'],
      dtype='object')
```

We will create two variables X and Y for building models. So, initialize X to values from all the above features and Y to *SOLD PRICE*, the outcome variable.

```
X = ipl_auction_encoded_df
Y = ipl_auction_df['SOLD PRICE']
```

6.4.1.2 Standardization of X and Y

Standardization is the process of bringing all features or variables into one single scale (normalized scale). This can be done by subtracting mean from the values and dividing by the standard deviation of the feature or variable. Standardizing helps to manage difference in scales of measurements of different variables. *StandardScaler* available in *sklearn.preprocessing* package provides this functionality.

```
from sklearn.preprocessing import StandardScaler

## Initializing the StandardScaler
X_scaler = StandardScaler()
## Standardize all the feature columns
X_scaled = X_scaler.fit_transform(X)

## Standardizing Y explicitly by subtracting mean and
## dividing by standard deviation
Y = (Y - Y.mean()) / Y.std()
```

6.4.1.3 Split the Dataset into Train and Test

Split the dataset into train and test with 80:20 split. *random_state* (seed value) is set to 42 for reproducibility of exact results.

```
from sklearn.model_selection import train_test_split
```

```
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled,
    Y,
    test_size=0.2,
    random_state = 42)
```

6.4.1.4 Build the Model

SGDRegressor in *sklearn.linear_model* is a variation of gradient descent algorithm for building linear regression model. Gradient descent algorithm uses all the training examples to learn to minimize the cost function, whereas *SGDRegressor* (Stochastic Gradient Descent) uses a subset of examples in each iteration for learning.

sklearn.linear_model also provides *LinearRegression* to build a linear regression model. We can initialize the *LinearRegression* class and then call *fit()* and pass *X_train* and *y_train* to build the model.

```
from sklearn.linear_model import LinearRegression
```

```
linreg = LinearRegression()
linreg.fit(X_train, y_train)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1,
normalize=False)
```

Let us print the coefficient using the following code:

```
linreg.coef_
```

```
array([-0.43539611, -0.04632556,  0.50840867, -0.03323988,  0.2220377,
       -0.05065703,  0.17282657, -0.49173336,  0.58571405, -0.11654753,
       0.24880095,  0.09546057,  0.16428731,  0.26400753, -0.08253341,
      -0.28643889, -0.26842214, -0.21910913, -0.02622351,  0.24817898,
       0.18760332,  0.10776084,  0.04737488,  0.05191335,  0.01235245,
       0.00547115, -0.03124706,  0.08530192,  0.01790803, -0.05077454,
       0.18745577])
```

The sign of the coefficients indicates positive or negative effect on a player's *SOLD PRICE*. We will store the beta coefficients and respective column names in a DataFrame and then sort the coefficient values in descending order to observe the effects.

```
## The dataframe has two columns to store feature name and the
## corresponding coefficient values
columns_coef_df = pd.DataFrame( { 'columns': ipl_auction_encoded_
                                    .columns,
                                    'coef': linreg.coef_ } )
```

```
## Sorting the features by coefficient values in descending order
sorted_coef_vals = columns_coef_df.sort_values( 'coef',
                                                ascending=False)
```

6.4.1.5 Plotting the Coefficient Values

The feature names and the coefficients are plotted using a bar plot to observe the effect (Figure 6.10). The vertical axis is the feature name and the horizontal axis is the coefficient value. As the features are standardized, the magnitude of the values also indicates the effect on outcome i.e. *SOLD PRICE*. The following commands are used to plot the coefficients of values:

```
plt.figure( figsize = ( 8, 6 ) ) ## Creating a bar plot
sn.barplot(x="coef", y="columns", data=sorted_coef_vals);
plt.xlabel("Coefficients from Linear Regression")
plt.ylabel("Features")
```

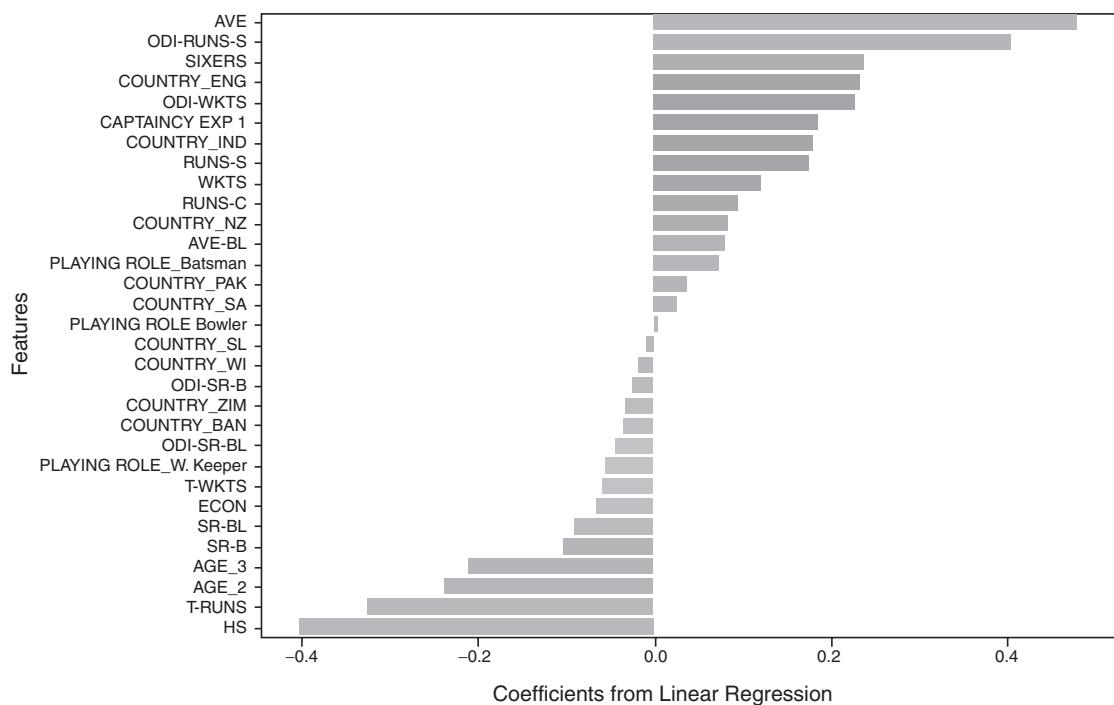


FIGURE 6.10 Bar plot depicting coefficient values of features in the model, sorted in descending order.

Few observations from Figure 6.10 are as follows:

1. AVE, ODI-RUNS-S, SIXERS are top three highly influential features which determine the player's *SOLD PRICE*.

2. Higher ECON, SR-B and AGE have negative effect on *SOLD PRICE*.
3. Interestingly, higher test runs (T-Runs) and highest score (HS) have negative effect on the *SOLD PRICE*. Note that few of these counter-intuitive sign for coefficients could be due to multicollinearity. For example, we expect SR-B (batting strike rate) to have a positive effect on the *SOLD PRICE*.

6.4.1.6 Calculate RMSE

We can calculate the RMSE on training and test sets to understand the model's ability to predict *SOLD PRICE*. We will develop an utility method *get_train_test_rmse()* to calculate and print the RMSE of train and test sets for comparison. It will take the model as a parameter. This will be used in all the models that we will be discussing in this chapter.

```
from sklearn import metrics

# Takes a model as a parameter
# Prints the RMSE on train and test set
def get_train_test_rmse( model ):
    # Predicting on training dataset
    y_train_pred = model.predict( X_train )
    # Compare the actual y with predicted y in the training dataset
    rmse_train = round(np.sqrt(metrics.mean_squared_error( y_train,
                                                          y_train_
                                                          pred)),3)

    # Predicting on test dataset
    y_test_pred = model.predict( X_test )
    # Compare the actual y with predicted y in the test dataset
    rmse_test = round(np.sqrt(metrics.mean_squared_error(y_test,
                                                          y_test_
                                                          pred)),3)
    print( "train: ", rmse_train, " test:", rmse_test )
```

Invoke the method *get_train_test_rmse()* with the model *linreg* as a parameter to get train and test accuracy.

```
get_train_test_rmse( linreg )
```

train: 0.679 test: 0.749

RMSE on the training set is 0.679, while it is 0.749 on the test set. A good model that generalizes well needs to have a very similar error on training and test sets. Large difference indicates that the model may be overfitting to the training set. Most widely used approach to deal with model overfitting is called *Regularization*, which will be discussed in the next section.

6.4.2 | Applying Regularization

One way to deal with overfitting is regularization. It is observed that overfitting is typically caused by inflation of the coefficients. To avoid overfitting, the coefficients should be regulated by penalizing

potential inflation of coefficients. Regularization applies penalties on parameters if they inflate to large values and keeps them from being weighted too heavily.

The coefficients are penalized by adding the coefficient terms to the cost function. If the coefficients become large, the cost increases significantly. So, the optimizer controls the coefficient values to minimize the cost function. Following are the two approaches that can be used for adding a penalty to the cost function:

1. **L1 Norm:** Summation of the absolute value of the coefficients. This is also called Least Absolute Shrinkage and Selection Operator (LASSO Term) (Tibshirani, 1996). The corresponding cost function is given by

$$\mathcal{E}_{\text{mse}} = \frac{1}{N} \sum_{i=1}^N (Y_i - (\beta_0 + \beta_1 X_1 + \dots + \beta_n X_n))^2 + \alpha \sum_{i=0}^n |\beta_i| \quad (6.10)$$

where α is the multiplier.

2. **L2 Norm:** Summation of the squared value of the coefficients. This is called *Ridge Term* (Hoerl A E and Kennard Kennard, 1970). The cost function is given by

$$\mathcal{E}_{\text{mse}} = \frac{1}{N} \sum_{i=1}^N (Y_i - (\beta_0 + \beta_1 X_1 + \dots + \beta_n X_n))^2 + \alpha \sum_{i=1}^n \beta_i^2 \quad (6.11)$$

Ridge term distributes (smoothens) the coefficient values across all the features, whereas LASSO seems to reduce some of the coefficients to zero. Features with coefficients value as zero can be treated as features with no contribution to the model. So, LASSO can also be used for feature selection, that is, remove features with zero coefficients, thereby reducing the number of features.

Figure 6.11 depicts the LASSO and Ridge constraint applied to the cost function (James *et al.*, 2013).

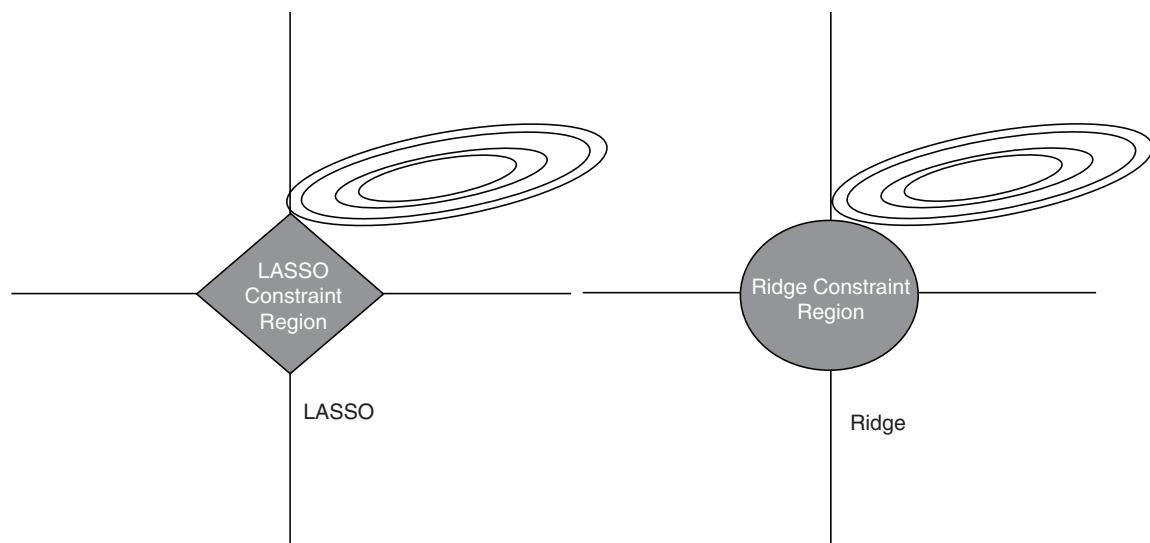


FIGURE 6.11 Effect of LASSO and Ridge terms on the cost function.

6.4.2.1 Ridge Regression

sklearn.linear_model provides Ridge regression for building linear models by applying L2 penalty.

Ridge regression takes the following parameters:

1. *alpha (α)* – float – is the regularization strength; regularization strength must be a positive float. Regularization improves the estimation of the parameters and reduces the variance of the estimates. Larger values of alpha imply stronger regularization.
2. *max_iter* – int (integer) – is the maximum number of iterations for the gradient solver.

```
# Importing Ridge Regression
from sklearn.linear_model import Ridge

# Applying alpha = 1 and running the algorithms for maximum of 500
# iterations
ridge = Ridge(alpha = 1, max_iter = 500)
ridge.fit( X_train, y_train )

Ridge(alpha=1, copy_X=True, fit_intercept=True, max_iter=500,
normalize=False, random_state=None, solver='auto', tol=0.001)

get_train_test_rmse( ridge )
```

train: 0.68 test: 0.724

The difference in RMSE on train and test has reduced because of penalty effect. The difference can be reduced by applying a stronger penalty. For example, apply α value as 2.0.

```
ridge = Ridge(alpha = 2.0, max_iter = 1000)
ridge.fit( X_train, y_train )
get_train_test_rmse( ridge )
```

train: 0.682 test: 0.706

The difference in model accuracy on training and test has reduced. We need to calculate the optimal value for α . This can be achieved in many ways. Multiple values of α can be tested before arriving at the optimal value. The parameters which can be tuned are called hyperparameters in machine learning. Here α is a hyperparameter.

sklearn.model_selection.GridSearchCV can help search for the optimal value and will be discussed later in the chapter. For now, let us assume the optimal value for α is 2.0.

6.4.2.2 LASSO Regression

sklearn.linear_model provides LASSO regression for building linear models by applying L1 penalty. Two key parameters for LASSO regression are:

1. *alpha* – float – multiplies the L1 term. Default value is set to 1.0.
2. *max_iter* – int – Maximum number of iterations for gradient solver.

```
# Importing LASSO Regression
from sklearn.linear_model import Lasso

# Applying alpha = 1 and running the algorithms for maximum of 500
# iterations
lasso = Lasso(alpha = 0.01, max_iter = 500)
lasso.fit( X_train, y_train )

Lasso(alpha=0.01, copy_X=True, fit_intercept=True, max_iter=500,
       normalize=False, positive=False, precompute=False, random_
       state=None, selection='cyclic', tol=0.0001, warm_start=False)

get_train_test_rmse( lasso )
```

train: 0.688 test: 0.698

It can be noticed that the model is not overfitting and the difference between train RMSE and test RMSE is very small. LASSO reduces some of the coefficient values to 0, which indicates that these features are not necessary for explaining the variance in the outcome variable.

We will store the feature names, coefficient values in a DataFrame and then filter the features with zero coefficients.

```
## Storing the feature names and coefficient values in the DataFrame
lasso_coef_df = pd.DataFrame( { 'columns': ipl_auction_encoded_
                               .columns,
                               'coef': lasso.coef_ } )
```



```
## Filtering out coefficients with zeros
lasso_coef_df[lasso_coef_df.coef == 0]
```

	coef	columns
1	0.0	T-WKTS
3	0.0	ODI-SR-B
13	0.0	AVE-BL
28	0.0	PLAYING ROLE_Bowler

The LASSO regression indicates that the features listed under “columns” are not influencing factors for predicting the *SOLD PRICE* as the respective coefficients are 0.0.

6.4.2.3 Elastic Net Regression

ElasticNet regression combines both L1 and L2 regularizations to build a regression model. The corresponding cost function is given by

$$\mathcal{E}_{\text{mse}} = \frac{1}{N} \sum_{i=1}^N (Y_i - (\beta_0 + \beta_1 X_1 + \cdots + \beta_n X_n))^2 + \gamma \sum_{i=0}^n |\beta_i| + \sigma \sum_{i=1}^n \beta_i^2 \quad (6.12)$$

While building *ElasticNet* regression model, both hyperparameters σ (L2) and γ (L1) need to be set. *ElasticNet* takes the following two parameters:

1. *alpha* – Constant that multiplies the penalty terms. Default value is set to 1.0. ($\text{alpha} = \sigma + \gamma$)
2. *l1_ratio* – The ElasticNet mixing parameter, with $0 \leq l1_ratio \leq 1$.

$$l1_ratio = \frac{\gamma}{\sigma + \gamma}$$

where

l1_ratio = 0 implies that the penalty is an L2 penalty.

l1_ratio = 1 implies that it is an L1 penalty.

$0 < l1_ratio < 1$ implies that the penalty is a combination of L1 and L2.

In the example below, penalties applied are $\gamma = 0.01$ and $\sigma = 1.0$. So

$$\text{alpha} = \sigma + \gamma = 1.01 \text{ and } l1_ratio = \frac{\gamma}{\sigma + \gamma} = 0.0099$$

```
from sklearn.linear_model import ElasticNet
enet = ElasticNet(alpha = 1.01, l1_ratio = 0.001, max_iter = 500)
enet.fit( X_train, y_train )
get_train_test_rmse( enet )
```

train: 0.789 test: 0.665

As we can see, applying both the regularizations did not improve the model performance. It has become worse. In this case, we can choose to apply only L1 (LASSO) regularization, which seems to deal with the overfitting problem efficiently.

6.5 | ADVANCED MACHINE LEARNING ALGORITHMS

In this section, we will take a binary classification problem and explore it through machine learning algorithms such as K-Nearest Neighbors (KNN), Random Forest, and Boosting. Bank marketing dataset available at the University of California, Irvine machine learning repository (<https://archive.ics.uci.edu/ml/datasets/Bank+Marketing>) is used in this section for the demonstration of various techniques. The dataset is based on a telemarketing campaign carried out by a Portuguese bank for subscription of a term deposit. The data has several features related to the potential customers and whether they subscribed the term deposit or not (outcome). The objective, in this case, is to predict which customers may respond to their marketing campaign to open a term deposit with the bank. The response variable $Y = 1$ implies that the customer subscribed a term deposit after the campaign and 0 otherwise. The marketing campaign is based on phone calls. We have already explored this dataset in Chapter 5.

We can use the following commands for reading the dataset and printing a few records.

```
bank_df = pd.read_csv('bank.csv')
bank_df.head(5)
```

	age	job	marital	education	default	balance	housing-loan	personal-loan	current-campaign	previous-campaign	subscribed
0	30	unemployed	married	primary	no	1787	no	no	1	0	no
1	33	services	married	secondary	no	4789	yes	yes	1	4	no
2	35	management	single	tertiary	no	1350	yes	no	1	1	no
3	30	management	married	tertiary	no	1476	yes	yes	4	0	no
4	59	blue-collar	married	secondary	no	0	yes	no	1	0	no

The following commands can be used for displaying information about the dataset.

```
bank_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4521 entries, 0 to 4520
Data columns (total 11 columns):
Age            4521 non-null int64
Job             4521 non-null object
Marital         4521 non-null object
Education       4521 non-null object
Default         4521 non-null object
Balance         4521 non-null int64
housing-loan    4521 non-null object
personal-loan   4521 non-null object
current-campaign 4521 non-null int64
previous-campaign 4521 non-null int64
Subscribed      4521 non-null object
dtypes: int64(4), object(7)
memory usage: 388.6+ KB
```

The dataset has 4521 observations and 10 features excluding the *subscribed* column (outcome variable). The details of the columns are given in Chapter 5, *Classification*.

6.5.1 | Dealing with Imbalanced Datasets

One of the major problems in machine learning is imbalanced (or unbalanced) dataset. A dataset is imbalanced when there is no equal representation of all classes in the data. For example, in the bank marketing dataset the proportion of customers who responded to the telemarketing is approximately 11.5% and the remaining 88.5% did not respond. Thus, the representation of two classes (responded, did not respond) in the dataset is not equal. In the data, the *subscribed* column indicates whether or not the customer has opened (subscribed) a term deposit account with the bank after the marketing campaign.

1. *yes* – the customer has opened the account
2. *no* – the customer has not opened the account

We have to check the number of records in each class to understand the imbalance. A simple `value_counts()` on the column values will provide the answer. The following command can be used for counting the number of records in each class:

```
bank_df.subscribed.value_counts()
```

```
no      4000
yes     521
Name: subscribed, dtype: int64
```

The dataset is quite imbalanced. Both the classes are not equally represented. There are only 521 (11.5%) observations in which customers have subscribed as opposed to 4000 observations where customers have not subscribed. In such cases, the model may not be able to learn and may be biased towards the class that is over-represented.

Even if the model predicts that no customer will subscribe (all negatives), it will have an accuracy of more than 88%. This is called *Accuracy Paradox*. But the objective of building a model here is to identify the customers who will subscribe to the term deposit (i.e., increase the number of *True Positives*).

One approach to deal with imbalanced dataset is bootstrapping. It involves resampling techniques such as *upsampling* and *downsampling*.

1. *Upsampling*: Increase the instances of under-represented minority class by replicating the existing observations in the dataset. Sampling with replacement is used for this purpose and is also called *Oversampling*.
2. *Downsampling*: Reduce the instances of over-represented majority class by removing the existing observations from the dataset and is also called *Undersampling*.

`sklearn.utils` has *resample* method to help with upsampling. It takes three parameters:

1. The original sample set
2. *replace*: Implements resampling with replacement. If false, all resampled examples will be unique.
3. *n_samples*: Number of samples to generate.

In this case, the number of examples of *yes* cases will be increased to 2000.

```
## Importing resample from *sklearn.utils* package.
from sklearn.utils import resample

# Separate the case of yes-subscribes and no-subscribes
bank_subscribed_no = bank_df[bank_df.subscribed == 'no']
bank_subscribed_yes = bank_df[bank_df.subscribed == 'yes']

##Upsample the yes-subscribed cases.
df_minority_upsampled = resample(bank_subscribed_yes,
                                 replace=True,
                                 n_samples=2000)
```

```
# Combine majority class with upsampled minority class
new_bank_df = pd.concat([bank_subscribed_no, df_minority_upsampled])
```

After upsampling, the `new_bank_df` contains 4000 cases of `subscribed = no` and 2000 cases of `subscribed = yes` in the ratio of 67:33. Before using the dataset, the examples can be shuffled to make sure they are not in a particular order. `sklearn.utils` has a method `shuffle()`, which does the shuffling.

```
from sklearn.utils import
shuffle new_bank_df = shuffle(new_bank_df)
```

We now assign all the features column names to `X_features` variable.

```
# Assigning list of all column names in the DataFrame
X_features = list( new_bank_df.columns )
# Remove the response variable from the list
X_features.remove( 'subscribed' )
X_features
```

```
['age',
'job',
'marital',
'education',
'default',
'balance',
'housing-loan',
'personal-loan',
'current-campaign',
'previous-campaign']
```

The following command can be used to encode all the categorical features into dummy features and assign to `X`.

```
## get_dummies() will convert all the columns with data type as
## objects
encoded_bank_df = pd.get_dummies( new_bank_df[X_features],
                                 drop_first = True )
X = encoded_bank_df
```

The `subscribed` column values are string literals and need to be encoded as follows:

1. `yes` to 1
2. `no` to 0

```
# Encoding the subscribed column and assigning to Y
Y = new_bank_df.subscribed.map( lambda x: int( x == 'yes' ) )
```

Now split the dataset into train and test sets in 70:30 ratio, respectively.

```
from sklearn.model_selection import train_test_split
train_X, test_X, train_y, test_y = train_test_split(X,
                                                    Y,
                                                    test_size=0.3,
                                                    random_state=42)
```

6.5.2 | Logistic Regression Model

6.5.2.1 Building the Model

Logistic regression is discussed in detail in Chapter 5, *Classification*. Cost function for logistic regression is called log loss (log likelihood) or binary cross-entropy function and is given by

$$-\frac{1}{N} \sum_{i=1}^N (y_i \ln(p_i) + (1-y_i) \ln(1-p_i)) \quad (6.13)$$

where p_i is the probability that Y belongs to class 1. It is given by

$$p_i = P(y_i = 1) = \frac{e^Z}{1+e^Z}$$

$$Z = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_m X_m$$

where X_1, X_2, \dots, X_m are features.

LogisticRegression is imported from *sklearn.linear_model* package and is fitted with the training data.

```
from sklearn.linear_model import LogisticRegression
## Initializing the model
logit = LogisticRegression()
## Fitting the model with X and Y values of the dataset
logit.fit( train_X, train_y)
```

Now we use the model to predict on the test set. *pred_y* will be assigned to the predicted classes. Note that we are not predicting class directly, which in turn predicts probabilities and determines the class by using 0.5 as a cut-off probability (or an optimal cut-off).

```
pred_y = logit.predict(test_X)
```

6.5.2.2 Confusion Matrix

We develop a custom method *draw_cm()* to draw the confusion matrix. This method will be used to draw the confusion matrix from the models we discuss in the subsequent sections of the chapter. It takes the actual and predicted class labels to draw the confusion matrix. Usage and interpretation of a confusion matrix is already explained in detail in Chapter 5.

```

## Importing the metrics
from sklearn import metrics

## Defining the matrix to draw the confusion matrix from actual and
## predicted class labels
def draw_cm( actual, predicted ):
    # Invoking confusion_matrix from metric package. The matrix
    # will be oriented as [1,0] i.e. the classes with label 1 will be
    # represented by the first row and 0 as second row
    cm = metrics.confusion_matrix( actual, predicted, [1,0] )
    # Confusion will be plotted as heatmap for better visualization
    # The labels are configured to better interpretation from the plot
    sn.heatmap(cm, annot=True, fmt='.2f',
                xticklabels = ["Subscribed", "Not Subscribed"],
                yticklabels = ["Subscribed", "Not Subscribed"] )
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.show()

```

Plot the confusion matrix using the following command:

```
cm = draw_cm( test_y, pred_y )
```



FIGURE 6.12 Confusion matrix of logistic regression model.

Figure 6.12 shows that the model has very few true positives and a large number of false negatives. In this business context, the objective is to build a model that will have a high percentage of true positives.

6.5.2.3 Classification Report

The `classification_report()` function in `sklearn.metrics` gives a detailed report of precision, recall and F1-score for each class. The report is shown in Table 6.1.

```
print( metrics.classification_report( test_y, pred_y ) )
```

TABLE 6.1 Model performance metrics

	precision	recall	f1-score	Support
0	0.72	0.93	0.81	1225
1	0.62	0.25	0.35	575
avg/total	0.69	0.71	0.67	1800

From Table 6.1, we learn that recall for positive cases is only 0.25. Most of the cases have been predicted as negative.

6.5.2.4 Receiver Operating Characteristic Curve (ROC) and Area under ROC (AUC) Score

ROC and AUC are two important measures of model performance for classification problems. The model's `predict_proba()` method gives the predicted probabilities for the test examples and can be passed to `roc_auc_score()` along with actual class labels to obtain AUC score. AUC score is explained in detail in Chapter 5.

```
## Predicting the probability values for test cases
predict_proba_df = pd.DataFrame( logit.predict_proba( test_X ) )
predict_proba_df.head()
```

	0	1
0	0.505497	0.494503
1	0.799272	0.200728
2	0.646329	0.353671
3	0.882212	0.117788
4	0.458005	0.541995

As shown in the result printed above, the second column in the DataFrame `predict_proba_df` has the probability for class label 1.

Now we create a DataFrame `test_results_df` to store the actual labels and predicted probabilities for class label 1.

```
## Initializing the DataFrame with actual class labels
test_results_df = pd.DataFrame( { 'actual': test_y } )
test_results_df = test_results_df.reset_index()
## Assigning the probability values for class label 1
test_results_df['chd_1'] = predict_proba_df.iloc[:,1:2]
```

Let us print the first 5 records.

```
test_results_df.head(5)
```

	index	actual	chd_1
0	2022	0	0.494503
1	4428	0	0.200728
2	251	0	0.353671
3	2414	0	0.117788
4	4300	1	0.541995

The DataFrame *test_results_df* contains the test example index, the actual class label and predicted probabilities for class 1 in columns *index*, *actual* and *chd_1*, respectively.

The ROC AUC score can be obtained using *metrics.roc_auc_score()*.

```
# Passing actual class labels and predicted probability values
# to compute ROC AUC score.

auc_score = metrics.roc_auc_score(test_results_df.actual,
                                  test_results_df.chd_1)
round( float( auc_score ), 2 )
```

0.7

That is, AUC score is 0.7.

Plotting ROC Curve

To visualize the ROC curve, an utility method *draw_roc_curve()* is implemented, which takes the mode, test set and actual labels of test set to draw the ROC curve. It returns the *auc_score*, false positive rate (FPR), true positive rate (TPR) values for different threshold (cut-off probabilities) ranging from 0.0 to 1.0. This method will be used for all future ML models that we will be discussing in subsequent sections. ROC AUC curve is discussed in detail in Section 5.2.10 (ROC and AUC), Chapter 5.

The following custom method is created for plotting ROC curve and calculating the area under the ROC curve.

```
## The method takes the following three parameters
## model: the classification model
## test_X: X features of the test set
## test_y: actual labels of the test set
## Returns
## - ROC Auc Score
## - FPR and TPRs for different threshold values
def draw_roc_curve( model, test_X, test_y ):
    ## Creating and initializing a results DataFrame with actual
    ## labels
    test_results_df = pd.DataFrame( { 'actual': test_y } )
    test_results_df = test_results_df.reset_index()

    # predict the probabilities on the test set
    predict_proba_df = pd.DataFrame( model.predict_proba( test_X ) )

    ## selecting the probabilities that the test example belongs
    ## to class 1
    test_results_df['chd_1'] = predict_proba_df.iloc[:,1:2]

    ## Invoke roc_curve() to return fpr, tpr and threshold values.
    ## Threshold values contain values from 0.0 to 1.0
    fpr, tpr, thresholds = metrics.roc_curve( test_results_
                                              df.actual,
                                              test_results_
                                              df.chd_1,
                                              drop_intermediate =
                                              False )

    ## Getting roc auc score by invoking metrics.roc_auc_score method
    auc_score = metrics.roc_auc_score( test_results_df.actual,
                                       test_results_df.chd_1 )

    ## Setting the size of the plot
    plt.figure(figsize=(8, 6))
    ## Plotting the actual fpr and tpr values
    plt.plot(fpr, tpr, label = 'ROC curve (area = %0.2f)' % auc_score)
    ## Plotting the diagonal line from (0,1)
    plt.plot([0, 1], [0, 1], 'k--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    ## Setting labels and titles
    plt.xlabel('False Positive Rate or [1 - True Negative Rate]')
```

```

plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic example')
plt.legend(loc="lower right")
plt.show()

return auc_score, fpr, tpr, thresholds

```

The plot of ROC curve is shown in Figure 6.13. The corresponding AUC value is 0.70.

```

## Invoking draw_roc_curve with the logistic regression model
_, _, _, _ = draw_roc_curve( logit, test_X, test_y )

```

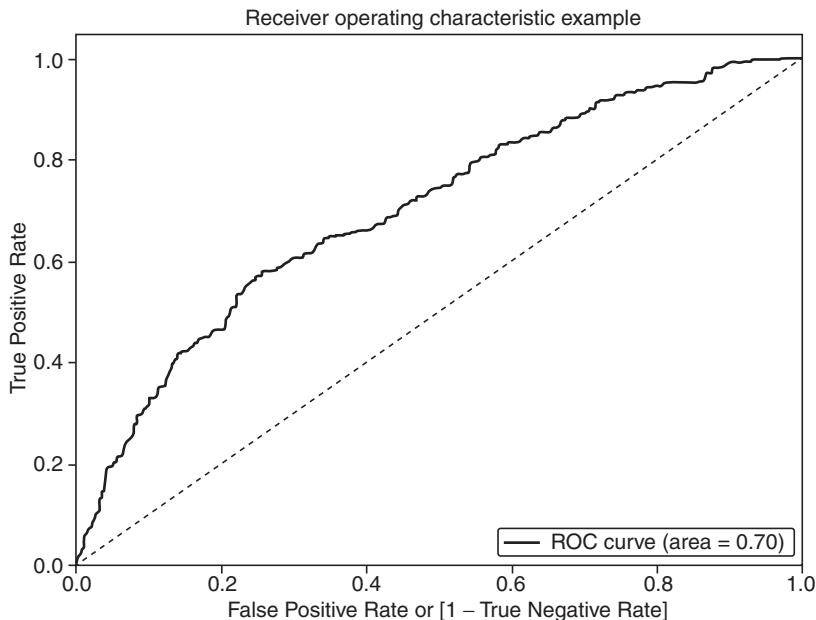


FIGURE 6.13 ROC AUC curve of logistic regression model.

6.5.3 | K-Nearest Neighbors (KNN) Algorithm

K-Nearest Neighbors (KNN) algorithm is a non-parametric, lazy learning algorithm used for regression and classification problems. Machine learning algorithms are of two types: parametric and non-parametric.

1. Parametric models estimate a fixed number of parameters from the data and strong assumptions of the data. The data is assumed to be following a specific probability distribution. Logistic regression is an example of a parametric model.
2. Non-parametric models do not make any assumptions on the underlying data distribution (such as normal distribution). KNN memorizes the data and classifies new observations by comparing the training data.

KNN algorithm finds observations in the training set, which are similar to the new observation. These observations are called neighbors. For better accuracy, a set of neighbors (K) can be considered for classifying a new observation. The class for the new observation can be predicted to be same class that majority of the neighbors belong to.

In Figure 6.14, observations belong to two classes represented by triangle and circle shapes. To find the class for a new observation, a set of neighbors, marked by the circle, are examined. As a majority of the neighbors belong to class B, the new observation is classified as *class B*.

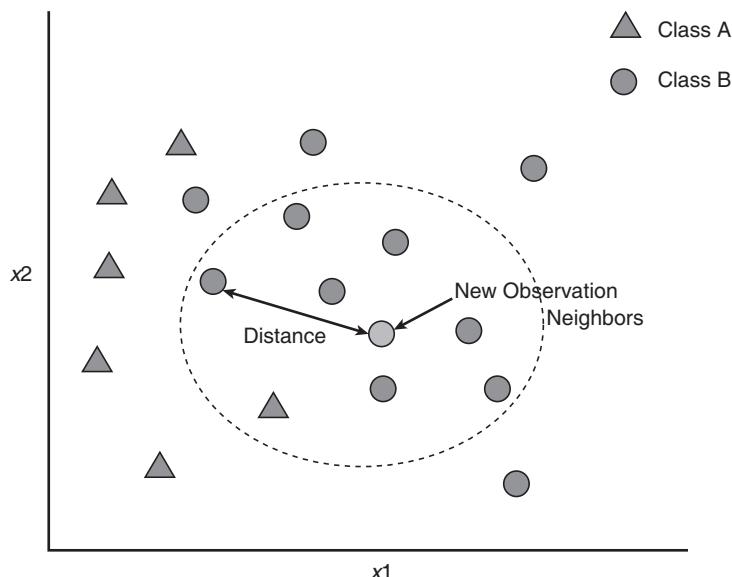


FIGURE 6.14 KNN algorithm.

The neighbors are found by computing distance between observations. Euclidean distance is one of the most widely used distance metrics. It is given by

$$D(O_1, O_2) = \sqrt{(X_{11} - X_{21})^2 + (X_{12} - X_{22})^2} \quad (6.14)$$

where O_1 and O_2 are two observations in the data. X_{11}, X_{21} are the values of feature X_1 for records 1 and 2, respectively, and X_{12} and X_{22} are the values of feature X_2 for records 1 and 2, respectively. Few more distance metrics such as Minkowski distance, Jaccard Coefficient and Gower's distance are used. For better understanding of these distances, refer to Chapter 14: *Clustering of Business Analytics: The Science of Data-Driven Decision Making* by U Dinesh Kumar (2017).

`sklearn.neighbors` provides `KNeighborsClassifier` algorithm for classification problems. `KNeighborsClassifier` takes the following parameters:

1. `n_neighbors`: int – Number of neighbors to use by default. Default is 5.
2. `metric`: string – The distance metrics. Default ‘Minkowski’. Available distance metrics in `sklearn` are discussed at the *Scikit-learn* site (see Reference section).
3. `weights` : str – Default is uniform where all points in each neighborhood are weighted equally. Else the distance which weighs points by the inverse of their distance.

Illustration of applying KNN algorithms to the bank marketing dataset is described below. The following default values are used to run the algorithm.

1. n_neighbors = 5
2. metric = 'minkowski'

```
## Importing the KNN classifier algorithm
from sklearn.neighbors import KNeighborsClassifier

## Initializing the classifier
knn_clf = KNeighborsClassifier()
## Fitting the model with the training set
knn_clf.fit( train_X, train_y )

KNeighborsClassifier(algorithm='auto', leaf_size=30,
metric='minkowski', metric_params=None, n_jobs=1, n_neighbors=5, p=2,
weights='uniform')
```

6.5.3.1 KNN Accuracy

We find ROC AUC score and draw the ROC curve (Figure 6.15).

```
## Invoking draw_roc_curve with the KNN model
_, _, _, _ = draw_roc_curve( knn_clf, test_X, test_y )
```

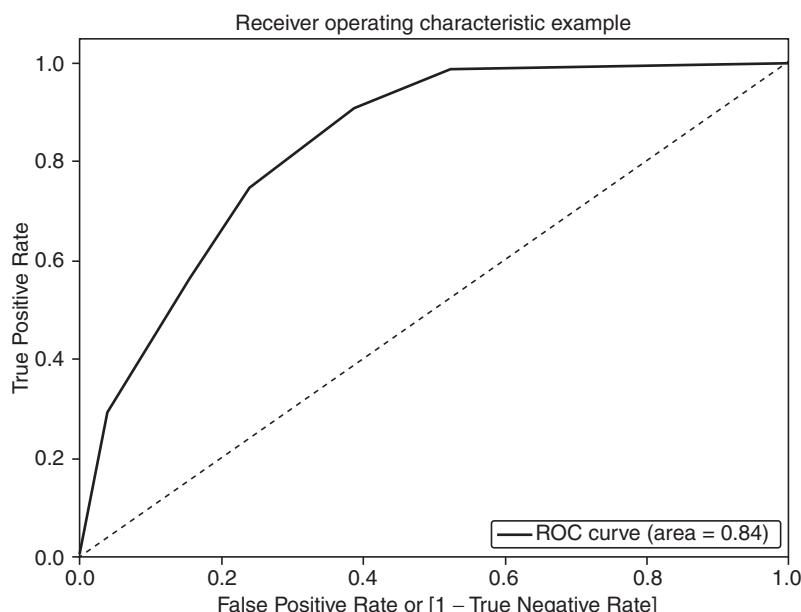


FIGURE 6.15 ROC AUC curve of KNN model.

As shown in Figure 6.15, KNN has AUC score of 0.84 and is better than the Logistic Regression Model. The confusion matrix can be created using the following commands:

```
## Predicting on test set
pred_y = knn_clf.predict(test_X)
## Drawing the confusion matrix for KNN model
draw_cm( test_y, pred_y )
```



FIGURE 6.16 Confusion matrix of KNN model.

Let us print the classification report for the KNN model.

```
print( metrics.classification_report( test_y, pred_y ) )
```

TABLE 6.2 Classification report for the KNN model

	precision	recall	f1-score	support
0	0.87	0.76	0.81	1225
1	0.59	0.75	0.66	575
avg/total	0.78	0.76	0.76	1800

As shown in Table 6.2, the **recall** of positive cases has improved from 0.25 (logistic regression model) to 0.75 in the KNN model. The above model accuracy is obtained by considering the default number of neighbors (i.e., $k = 5$). Can the accuracy of the model be improved by increasing or reducing the number of neighbors? In other words, what is the most optimal number of neighbors (K) to be considered for classification in this case? K in KNN is called *hyperparameter* and the process of finding optimal value for a hyperparameter is called *hyperparameter tuning*.

sklearn has a *GridSearch* mechanism in which one or multiple hyperparameters can be searched through for the most optimal values, where the model gives the highest accuracy. The search mechanism is a brute force approach, that is, evaluate all the possible values and find the most optimal ones.

6.5.3.2 GridSearch for Optimal Parameters

One of the problems in machine learning is the selection of optimal hyperparameters. *sklearn.model_selection* provides a feature called *GridSearchCV*, which searches through a set of given hyperparameter values and reports the most optimal one. *GridSearchCV* does *k*-fold cross-validation for each value of hyperparameter to measure accuracy and avoid overfitting.

GridSearchCV can be used for any machine learning algorithm to search for optimal values for its hyperparameters. *GridSearchCV* searches among a list of possible hyperparameter values and reports the best value based on accuracy measures. *GridSearchCV* takes the following parameters:

1. *estimator* – scikit-learn model, which implements estimator interface. This is the ML algorithm.
2. *param_grid* – A dictionary with parameter names (string) as keys and lists of parameter values to search for.
3. *scoring* – string – the accuracy measure. For example, ‘r2’ for regression models and ‘f1’, ‘precision’, ‘recall’ or ‘roc_auc’ for classification models.
4. *cv* – integer – the number of folds in K-fold.

In our example, the hyperparameters and corresponding set of values to search for are as follows:

1. *n_neighbhors* – All values from 5 to 10.
2. *metric* (for distance calculation) – ‘canberra’, ‘euclidean’, ‘minkowski’.

All possible combination of hyperparameters will be evaluated by *GridSearchCV*. Also, to measure the robustness of the model, each set of value will be evaluated by K-fold cross-validation. The accuracy measure reported will be the average accuracy across all the folds from K-fold cross-validation.

```
## Importing GridSearchCV
from sklearn.model_selection import GridSearchCV

## Creating a dictionary with hyperparameters and possible values
## for searching
tuned_parameters = [{ 'n_neighbors': range(5,10),
                      'metric': ['canberra', 'euclidean',
                                'minkowski'] }]

## Configuring grid search
clf = GridSearchCV(KNeighborsClassifier(),
                     tuned_parameters,
                     cv=10,
                     scoring='roc_auc')

## fit the search with training set
clf.fit(train_X, train_y)
```

```
GridSearchCV(cv=10, error_score='raise',
            estimator=KNeighborsClassifier(algorithm='auto', leaf_size=30,
                metric='minkowski',
                metric_params=None, n_jobs=1, n_neighbors=5, p=2,
                weights='uniform'),
            fit_params=None, iid=True, n_jobs=1,
            param_grid=[{'n_neighbors': range(5, 10), 'metric':
                ['Canberra', 'euclidean', 'minkowski']}],
            pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
            scoring='roc_auc', verbose=0)
```

Once the search is over, the best score and params (parameters) can be printed as follows.

```
clf.best_score_
```

0.8293

```
clf.best_params_
```

```
{'metric': 'canberra', 'n_neighbors': 5}
```

GridSearch suggests that the best combination of parameters is *n_neighbhors* = 5 and *canberra* distance and the corresponding *roc_auc* score is 0.826, which is slightly higher than *minkowski* distance. Grid search also reports the accuracy metrics for all combinations of hyperparameters it has searched through.

```
clf.grid_scores_
```

```
[mean : 0.82933, std: 0.01422, params: {'metric' : 'Canberra',
                                             'n_neighbors' : 5},
 mean : 0.81182, std: 0.01771, params: {'metric' : 'Canberra',
                                             'n_neighbors' : 6},
 mean : 0.80038, std: 0.01953, params: {'metric' : 'Canberra',
                                             'n_neighbors' : 7},
 mean : 0.79129, std: 0.01894, params: {'metric' : 'Canberra',
                                             'n_neighbors' : 8},
 mean : 0.78268, std: 0.01838, params: {'metric' : 'Canberra',
                                             'n_neighbors' : 9},
 mean : 0.79774, std: 0.01029, params: {'metric' : 'euclidean',
                                             'n_neighbors' : 5},
 mean : 0.77504, std: 0.01127, params: {'metric' : 'euclidean',
                                             'n_neighbors' : 6},
 mean : 0.75184, std: 0.01258, params: {'metric' : 'euclidean',
                                             'n_neighbors' : 7},
 mean : 0.73387, std: 0.01330, params: {'metric' : 'euclidean',
                                             'n_neighbors' : 8},
 mean : 0.71950, std: 0.01310, params: {'metric' : 'euclidean',
                                             'n_neighbors' : 9},
```

```

mean : 0.79774, std: 0.01029, params: {'metric' : 'minkowski',
                                         'n_neighbors' : 5},
mean : 0.77504, std: 0.01127, params: {'metric' : 'minkowski',
                                         'n_neighbors' : 6},
mean : 0.75184, std: 0.01258, params: {'metric' : 'minkowski',
                                         'n_neighbors' : 7},
mean : 0.73387, std: 0.01330, params: {'metric' : 'minkowski',
                                         'n_neighbors' : 8},
mean : 0.71950, std: 0.01310, params: {'metric' : 'minkowski',
                                         'n_neighbors' : 9} ]

```

It also reports the mean and standard deviation of accuracy scores from the cross-validation. Higher standard deviation could indicate overfitting models.

6.5.4 | Ensemble Methods

All the models discussed in this chapter so far predict the value of the outcome variable based on a single model. Ensemble methods, on the other hand, are learning algorithms that take a set of estimators or classifiers (models) and classify new data points using strategy such as majority vote. The majority voting could be based on counting simply the vote from each class or it could be weighted based on their individual accuracy measures. Ensemble methods are also used for regression problems, where the prediction of new data is simple average or weighted average of all the predictions from the set of regression models as shown in Figure 6.17.

Multiple datasets are needed for building multiple classifiers. However, in practice there is always one training set available. So instead of using the same training set across all classifiers, strategy such as bootstrapped samples (random samples with replacement) are drawn from the initial training set and given to each classifier. Sometimes bootstrapping involves sampling features (using a subset of features) along with sampling observations. Each resampled set contains a subset of features available in the original set. Sampling features help to find important features, which is discussed in the subsequent sections in detail. Records that are not part of a specific sample are used for testing the model accuracy. Such records are called *Out-of-Bag (OOB) records*.

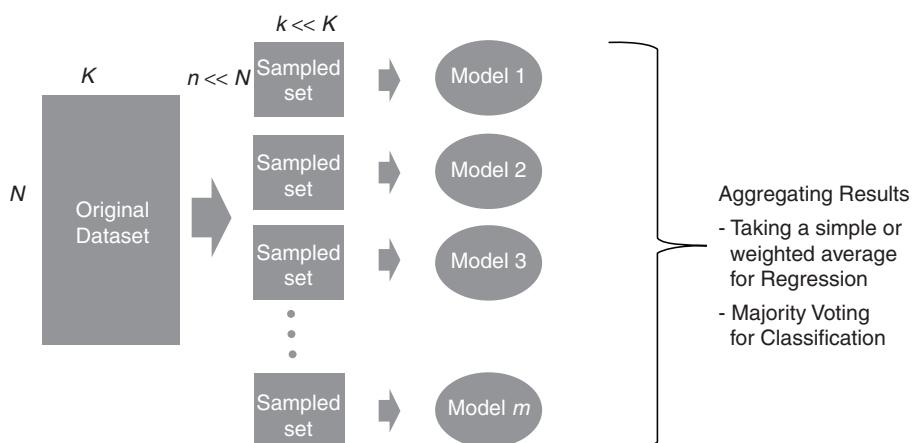


FIGURE 6.17 Ensemble technique.

In Figure 6.17, K and N , respectively, are number of features and records in the original set, while k and n , respectively, are the number of features and records in the sampled set (bootstrapped samples). The process of bootstrapping samples from original set to build multiple models and aggregating their results for final prediction is called *Bagging*. The term “bagging” comes from bootstrapping plus aggregating. One of the most widely used bagging technique is *Random Forest*.

6.5.5 | Random Forest

Random forest is one of the most popular ensemble techniques used in the industry due to its performance and scalability. A random forest is an ensemble of decision trees (classification and regression tree), where each decision tree is built from bootstrap samples (sampling with replacement) and randomly selected subset of features without replacement. The decision trees are normally grown deep (without pruning).

The number of estimators or models to be used in Random Forest can be tuned to increase the model accuracy. The hyperparameters in a Random Forest model are

1. Number of decision trees.
2. Number of records and features to be sampled.
3. Depth and search criteria (Gini impurity index or entropy).

RandomForestClassifier is available in *sklearn.ensemble* and takes the following parameters:

1. *n_estimators*: integer – The number of trees in the forest.
2. *criterion*: string – The function to measure the quality of a split, Gini or entropy. Default criteria used is Gini.
3. *max_feature*: int, float – The number of features to be used for each tree.
 - If int, then consider *max_features* for each tree.
 - If float, then *max_features* is a percentage and $\text{int}(\text{max_features} * \text{n_features})$ features are considered for each tree.
 - If “auto”, then *max_features* = $\sqrt{\text{n_features}}$.
 - If “sqrt”, then *max_features* = $\sqrt{\text{n_features}}$ (same as “auto”).
 - If “log2”, then *max_features* = $\log_2(\text{n_features})$.
4. *max_depth*: integer – The maximum depth of the tree.

6.5.5.1 Building Random Forest Model

Random forest model can be developed using the following commands. In this case, *max_depth* is 10 and *n_estimators* is 10.

```
## Importing Random Forest Classifier from the sklearn.ensemble
from sklearn.ensemble import RandomForestClassifier

## Initializing the Random Forest Classifier with max_dept and
## n_estimators
radm_clf = RandomForestClassifier(max_depth=10, n_estimators=10)
radm_clf.fit( train_X, train_y )
```

```
RandomForestClassifier(bootstrap=True, class_weight=None,
                      criterion='gini',
                      max_depth=15, max_features='auto', max_leaf_nodes=None,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=1, min_samples_split=2,
                      min_weight_fraction_leaf=0.0, n_estimators=20, n_jobs=1,
                      oob_score=False, random_state=None, verbose=0,
                      warm_start=False)
```

Now we find the ROC AUC score and draw the ROC curve using utility method *draw_roc_curve()* (Figure 6.18).

```
_ = draw_roc_curve( radm_clf, test_X, test_y );
```

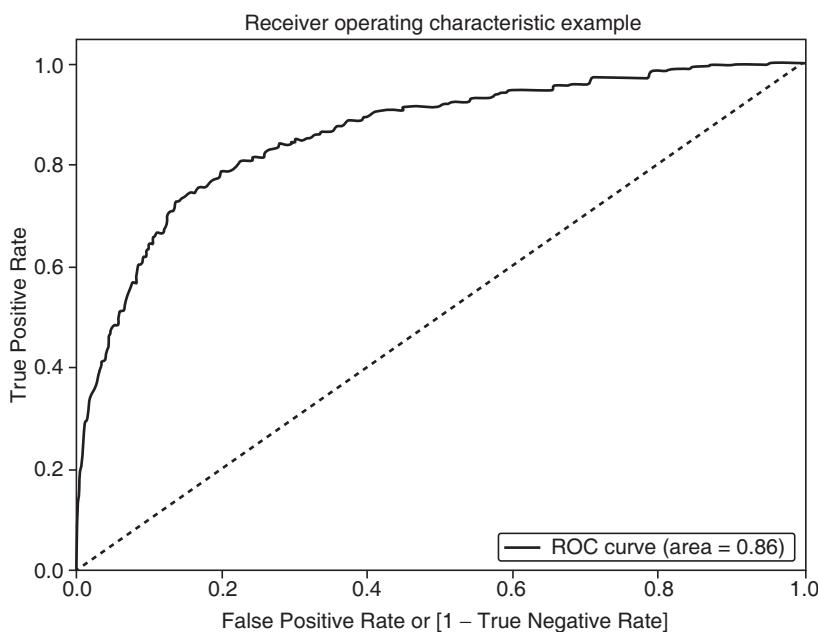


FIGURE 6.18 ROC AUC curve of random forest model with 10 decision trees.

AUC for the random forest model is 0.86 (Figure 6.18) and better compared to the KNN model. But we can still improve the accuracy by using grid search by fine-tuning the hyperparameters.

6.5.5.2 Grid Search for Optimal Parameters

We are not sure whether the hyperparameter used in the previous section for developing the random forest is optimal. We can use grid computing to find the best values for hyperparameters; one disadvantage of grid search is that it can be time-consuming. We will tune only the parameters *max_depth*, *n_estimators* and *max_features* and use a small set of values for search.

The commands for grid computing are provided below:

1. max_depth: 10 or 15
2. n_estimators: 10 or 20
3. max_features: sqrt or 0.2

```
## Configuring parameters and values for searched
tuned_parameters = [{`max_depth`: [10, 15],
                     `n_estimators`: [10, 20],
                     `max_features`: ['sqrt', 0.2]}]

## Initializing the RF classifier
radm_clf = RandomForestClassifier()

## Configuring search with the tunable parameters
clf = GridSearchCV(radm_clf,
                     tuned_parameters,
                     cv=5,
                     scoring='roc_auc')

## Fitting the training set
clf.fit(train_X, train_y)

GridSearchCV(cv=5, error_score='raise',
             estimator=RandomForestClassifier(bootstrap=True, class_
             weight=None, criterion='gini',
             max_depth=None, max_features='auto', max_leaf_nodes=None,
             min_impurity_decrease=0.0, min_impurity_split=None,
             min_samples_leaf=1, min_samples_split=2,
             min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=1,
             oob_score=False, random_state=None, verbose=0,
             warm_start=False),
             fit_params=None, iid=True, n_jobs=1,
             param_grid=[{`n_estimators`: [10, 20], `max_features`: ['sqrt',
             'auto'], `max_depth`: [10, 15]}],
             pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
             scorings='roc_auc', verbose=0)
```

Printing the best score and best parameters.

```
clf.best_score_
```

0.9413

AUC score has reached 0.94 with the following optimal values for the hyperparameters. This may change slightly for the readers because the folds are created randomly by the cross-validation strategy.

```
clf.best_params_
{'max_depth': 15, 'max_features': 'auto', 'n_estimators': 20}
```

The best accuracy of 0.9413 AUC score is given by a random forest model with 20 decision trees (estimators), a maximum number of features as auto (square root of number of total features) and max depth as 15.

6.5.5.3 Building the Final Model with Optimal Parameter Values

Following commands can be used for developing a new random forest model with optimal parameter values.

```
## Initializing the Random Forest Model with the optimal values
radm_clf = RandomForestClassifier(max_depth=15,
                                   n_estimators=20,
                                   max_features = 'auto')

## Fitting the model with the training set
radm_clf.fit( train_X, train_y )

RandomForestClassifier(bootstrap=True, class_weight=None,
                      criterion='gini',
                      max_depth=15, max_features= 'auto', max_leaf_nodes=None,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=1, min_samples_split=2,
                      min_weight_fraction_leaf=0.0, n_estimators=20, n_jobs=1,
                      oob_score=False, random_state=None, verbose=0,
                      warm_start=False)
```

6.5.5.4 ROC AUC Score

Following commands can be used for finding ROC AUC score and drawing the ROC curve using our utility method `draw_roc_curve` (Figure 6.19).

```
_ = draw_roc_curve( clf, test_X, test_y )
```

As shown in the figure, the AUC value for this random forest is 0.94.

6.5.5.5 Drawing the Confusion Matrix

Use the following commands to drawing the confusion matrix using the utility method `draw_cm()` (Figure 6.20).

```
pred_y = radm_clf.predict( test_X )
draw_cm( test_y, pred_y )
```

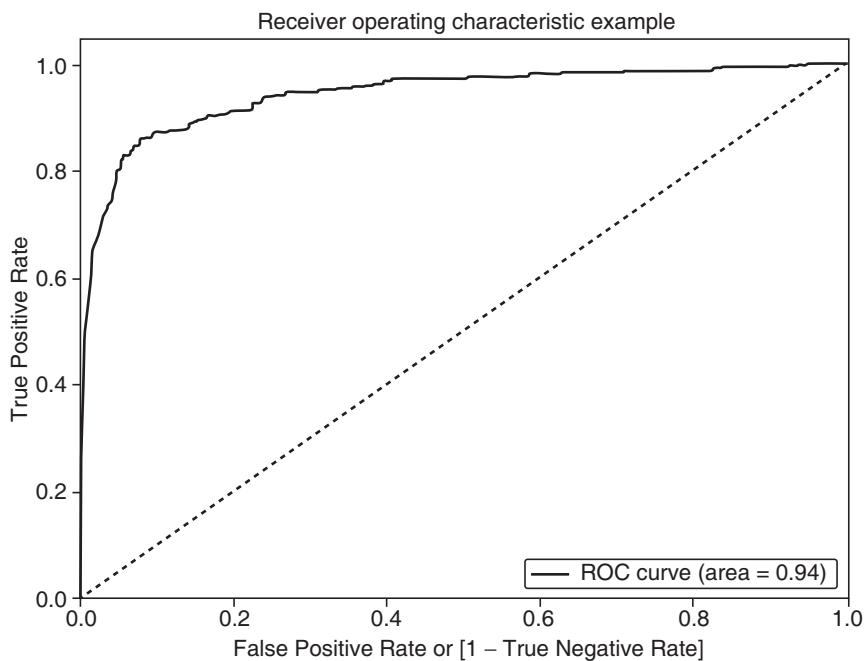


FIGURE 6.19 ROC AUC curve of random forest model with 10 decision trees.



FIGURE 6.20 Confusion matrix of random forest model.

The model can detect 464 out of 575 subscribed cases and there are only 75 false positives cases. There could be slight variation in these values since the trees are developed using random sampling with replacement.

Let us print the classification report as shown in Table 6.3.

```
print( metrics.classification_report( test_y, pred_y ) )
```

TABLE 6.3 Classification report

	Precision	recall	f1-score	support
0	0.91	0.94	0.93	1225
1	0.86	0.81	0.83	575
avg/total	0.90	0.90	0.90	1800

As shown in Table 6.3, the precision and recall for positive cases are 0.86 and 0.81, respectively, which are far better than what we obtained using logistic regression (Section 6.5.2) and KNN model (Section 6.5.3).

6.5.5.6 Finding Important Features

Random forest algorithm reports feature importance by considering feature usage over all the trees in the forest. This gives good insight into which features have important information with respect to the outcome variable. It uses “Gini impurity reduction” or “mean decrease impurity” for calculating the importance.

Feature importance is calculated for a feature by multiplying error reduction at any node by the feature with the proportion of samples reaching that node. Then the values are averaged over all the trees to find final feature importance.

In *sklearn*, the classifiers return a parameter called *featureimportances*, which holds the feature importance values. We can store these values along with the column names in a DataFrame and plot them in descending order of importance for better visualization and interpretation (Figure 6.21).

```
import numpy as np

# Create a dataframe to store the features and their corresponding
# importances
feature_rank = pd.DataFrame( { 'feature': train_X.columns,
                                'importance': radm_clf.feature_
                                importances_ } )

## Sorting the features based on their importances with most
## important feature at top.
feature_rank = feature_rank.sort_values('importance', ascending =
                                         False)

plt.figure(figsize=(8, 6))
# plot the values
sn.barplot( y = 'feature', x = 'importance', data = feature_rank );
```

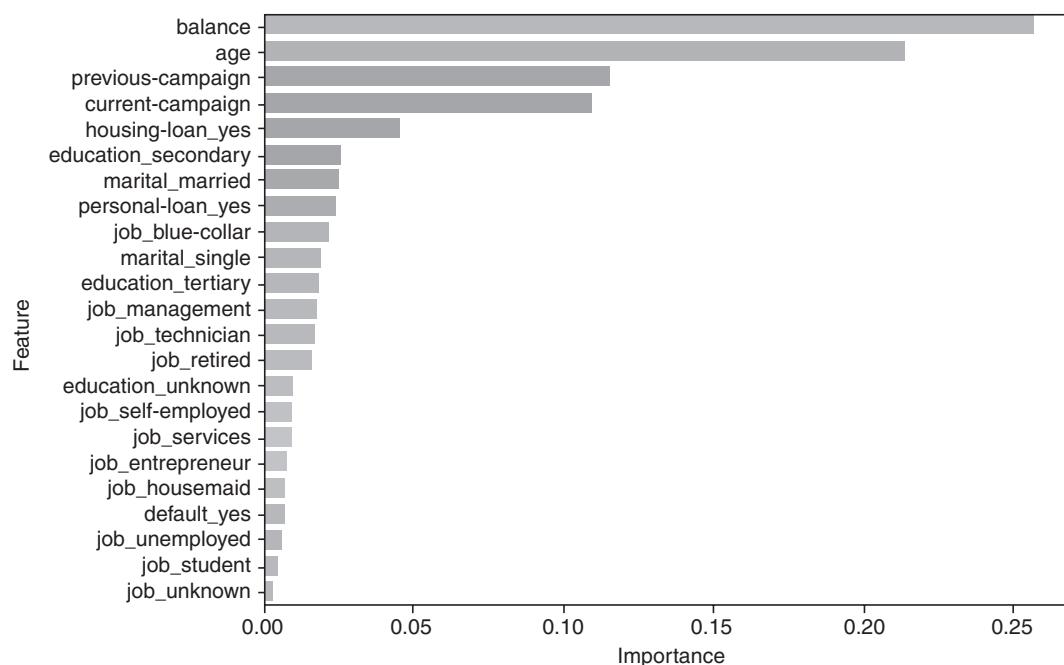


FIGURE 6.21 Features sorted by their importance values in the random forest model.

The top 5 features (Figure 6.21) are *balance*, *age*, *previous-campaign*, *current-campaign* and *house-loan_yes*. The importance score is normalized and shows the relative importance of features. The cumulative sum of features importance can show the amount of variance explained by the top five features.

Pandas' *cumsum()* method can be used to compute the cumulative sum, which is shown in Table 6.4.

```
feature_rank['cumsum'] = feature_rank.importance.cumsum() * 100
feature_rank.head(10)
```

TABLE 6.4 Feature importance from random forest model

	feature	importance	cumsum
1	balance	0.256938	25.693789
0	age	0.214292	47.122946
3	previous-campaign	0.115596	58.682579
2	current-campaign	0.109440	69.626538
21	housing-loan_yes	0.045655	74.192006
17	education_secondary	0.026044	76.796379
15	marital_married	0.025683	79.364678

TABLE 6.4 Continued

	feature	importance	cumsum
22	personal-loan_yes	0.024546	81.819316
4	job_blue-collar	0.021823	84.001666
16	marital_single	0.019638	85.965419

The top five features provide 75% of the information in the data with respect to the outcome variable. This technique can also be used for feature selection. Random forest being a black box model, cannot be interpreted. But it can be used to select a subset of features using feature importance criteria and build simpler models for interpretation.

6.5.6 | Boosting

Boosting is another popular ensemble technique which combines multiple *weak classifiers* into a single *strong classifier*. A weak classifier is one which is slightly better than random guessing. That is, the error is less than 50%. Any classification algorithm can be used for boosting and is called base classifier. Boosting builds multiple classifiers in a sequential manner as opposed to bagging, which can build classifiers in parallel. Boosting builds initial classifier by giving equal weights to each sample and then focuses on correctly classifying misclassified examples in subsequent classifiers. Two most widely used boosting algorithms are *AdaBoost* and *Gradient Boosting*.

6.5.6.1 AdaBoost

AdaBoost assigns weight to each record in a training set, which determines the probability that a specific record is selected for training a classifier. For first classifier, the weights of all examples will be equal (random sampling), but after training the first classifier, AdaBoost increases the weight for the misclassified records so that these records will have a higher probability of selection in the training set in anticipation that the next classifier will learn to classify them correctly.

After each classifier (usually a decision tree) is trained, the classifier's weight is computed based on its accuracy. More accurate classifiers are given more weight. The final prediction is based on the weighted sum of all classifiers and given by the following formula (Schapire, 1999):

$$F(X_i) = \text{Sign} \left(\sum_{k=1}^K \alpha_k f_k(X_i) \right) \quad (6.15)$$

where K is the total number of classifiers, $f_k(x_i)$ is the output of weak classifier k for input feature set X_i , α_k is the weight applied to classifier k and is given by

$$\alpha_k = \frac{1}{2} \ln \left(\frac{1 - \varepsilon_k}{\varepsilon_k} \right) \quad (6.16)$$

Here α_k is based on the classifier's error rate ε_k , which is the number of misclassified records over the training set divided by the total number of records of the training set. The final output $F(X_i)$ is just a linear combination of all of the weak classifiers, and final decision depends on the sign of this sum.

sklearn.ensemble.AdaBoostClassifier takes the following parameters to build an AdaBoost model:

1. *base_estimator*: object – The base estimator from which the boosted ensemble is built. The estimator should have the capability to utilize differential weights for samples (for example, decision tree or logistic regression).
2. *n_estimators*: integer – The maximum number of estimators at which boosting algorithm is terminated. In the case of perfect fit, the learning procedure is stopped early. Default is 50.
3. *learning_rate*: float – Learning rate shrinks the contribution of each classifier specified by *learning_rate*. There is a trade-off between *learning_rate* and *n_estimators*. Default value for *learning_rate* is 1.

The commands for building an AdaBoost model using Logistic Regression as base classifier and using 50 classifiers are as follows:

```
## Importing AdaBoost classifier
from sklearn.ensemble import AdaBoostClassifier

## Initializing logistic regression to use as base classifier
logreg_clf = LogisticRegression()

## Initializing AdaBoost classifier with 50 classifiers
ada_clf = AdaBoostClassifier(logreg_clf, n_estimators=50)

## Fitting AdaBoost model to training set
ada_clf.fit(train_X, train_y)

AdaBoostClassifier(algorithm='SAMME.R',
                   base_estimator=LogisticRegression(C=1.0, class_
                   weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, max_iter=100, multi_class='ovr',
                   n_jobs=1,
                   penalty='12', random_state=None, solver='liblinear',
                   tol=0.0001,
                   verbose=0, warm_start=False),
                   learning_rate=1.0, n_estimators=50, random_state=None)
```

Use the following command to find ROC AUC score. We then draw the ROC curve for the AdaBoost classifier using our custom method *draw_roc_curve()* (Figure 6.22).

```
_,'_,'_,'_ = draw_roc_curve(ada_clf, test_X, test_y)
```

As shown in Figure 6.22, the AUC for the AdaBoost is 0.71.

6.5.6.2 Gradient Boosting

AdaBoost focusses on the misclassified examples in subsequent classifiers, whereas Gradient Boosting focusses on residuals from previous classifiers and fits a model to the residuals. Gradient boosting algorithm repetitively leverages the patterns in residuals and strengthens the model with weak predictions and makes it better. Once it reaches a stage in which residuals do not have any pattern that could be modeled, it stops modeling residuals. It uses gradient descent algorithm in each stage to minimize the error. Gradient boosting typically uses decision tree as base classifier.

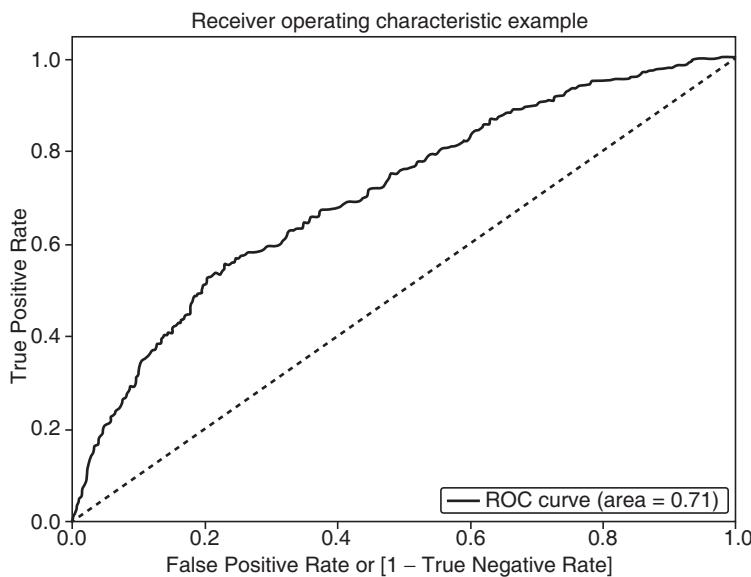


FIGURE 6.22 ROC AUC curve of AdaBoost model with 50 logistic regressions.

The following steps are used in gradient boosting:

1. Fit the first model to the data $Y = F_1(x)$.
2. Fit the next model to the residuals $\epsilon_1 = Y - F_1(x) = F_2(x)$.
3. The above steps are repeated until the residuals do not change significantly.
4. The final model is used to make the final predictions.

$$F(X_i) = \left(\sum_{k=1}^K \alpha_k F_k(X_i) \right)$$

where α_k is the learning rate or accuracy of model k .

`sklearn.ensemble.GradientBoostingClassifier` classifier takes the following key parameters:

1. `n_estimators`: int – The number of boosting stages to perform. Default value is 100. Gradient boosting is fairly robust to overfitting, so usually a large number of estimators result in better performance.
2. `max_depth`: integer – Maximum depth of the individual regression estimators. The maximum depth limits the number of levels from the root node in the tree.
3. `max_features`: int, float, string – The number of features to consider when looking for the best split. The applicable values are given in random forest section.

We will build a model with 500 estimators and `max_depth` value as 100.

```
## Importing Gradient Boosting classifier
from sklearn.ensemble import GradientBoostingClassifier

## Initializing Gradient Boosting with 500 estimators and max
## depth as 10.
```

```

gboost_clf = GradientBoostingClassifier( n_estimators=500,
                                         max_depth=10)

## Fitting gradient boosting model to training set
gboost_clf.fit(train_X, train_y)

GradientBoostingClassifier(criterion='friedman_mse', init=None,
                           learning_rate=0.1, loss= 'deviance', max_depth=10,
                           max_features=None, max_leaf_nodes=None,
                           min_impurity_decrease=0.0, min_impurity_split=None,
                           min_samples_leaf=1, min_samples_split=2,
                           min_weight_fraction_leaf=0.0, n_estimators=500,
                           presort='auto', random_state=None, subsample=1.0,
                           verbose=0,warm_start=False)

```

Now we find ROC AUC score and draw the ROC curve for the gradient boosting classifier using our utility method *draw_roc_curve()* (Figure 6.23).

```
_,' _,' _,' _ = draw_roc_curve( gboost_clf, test_X, test_y )
```

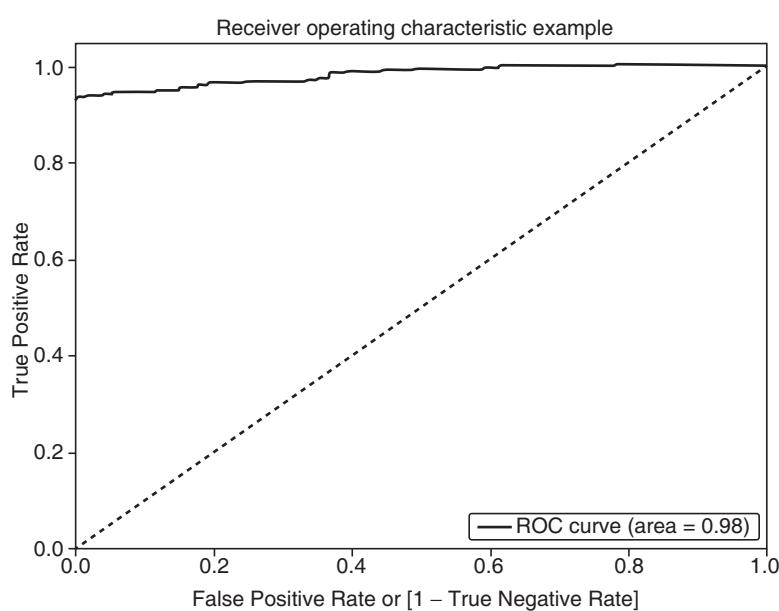


FIGURE 6.23 ROC AUC curve of gradient boosting model with 500 estimators.

The ROC curve of gradient boosting is shown in Figure 6.23. The corresponding AUC is 0.98.

The model has a 98% AUC score. It seems to be the best among all models we have built so far. But is it overfitting? We will run a 10-fold cross-validation to check the model's accuracy.

```
from sklearn.model_selection import cross_val_score
gboost_clf = GradientBoostingClassifier( n_estimators=500,
                                         max_depth=10)
cv_scores = cross_val_score( gboost_clf, train_X, train_y,
                             cv = 10, scoring = 'roc_auc' )
```

```
print(cv_scores)
print("Mean Accuracy: ", np.mean(cv_scores), " with standard
      deviation of:", np.std(cv_scores))
```

[0.96983951 0.96098506 0.9630729 0.95004276 0.98561151 0.97185641
0.97050897 0.97493263 0.97573346 0.96186505]

Mean Accuracy: 0.9684448270398314 with standard deviation of: 0.0
093498165767254

The model accuracy is consistent with a mean accuracy of 96.8% and a standard deviation of only 1%. This is a robust model. Now use the following commands to plot the confusion matrix and print the classification report.

```
gboost_clf.fit(train_X, train_y )
pred_y = gboost_clf.predict( test_X )
draw_cm( test_y, pred_y )
```

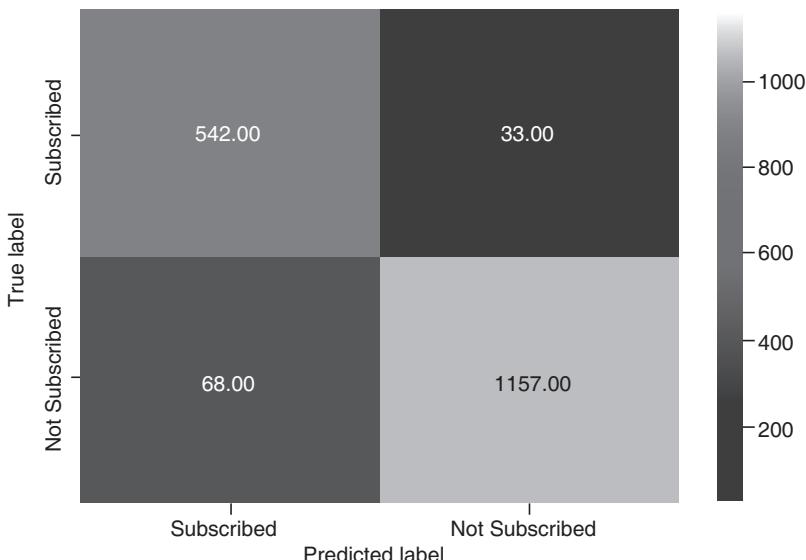


FIGURE 6.24 Confusion matrix of gradient boosting model.

As shown in Figure 6.24, the false negatives are only 33 and false positives are 68.

Now, let us print the classification report (shown in Table 6.5).

```
print( metrics.classification_report( test_y, pred_y ) )
```

TABLE 6.5 Classification report

	precision	recall	f1-score	support
0	0.97	0.94	0.96	1225
1	0.89	0.94	0.91	575
avg/total	0.95	0.94	0.94	1800

The model detects 542 out of 575 subscribed cases with recall reaching 94%. The readers can try *GridSearchCV* for finding optimal parameters and verify if the model can still be improved.

Like random forest algorithm, the boosting algorithm also provides feature importance based on how each feature has contributed to the accuracy of the model. Use the following commands to create a DataFrame to store feature names and feature importance from gradients boosting classifier and then sorting them by their importance values.

```
import numpy as np

# Create a dataframe to store the features and their corresponding
# importances
feature_rank = pd.DataFrame( { 'feature': train_X.columns,
                               'importance': gboost_clf.feature_
                               importances_ } )

## Sorting the features based on their importances with most
## important feature at top.
feature_rank = feature_rank.sort_values('importance', ascending =
                                         False)

plt.figure(figsize=(8, 6))
# plot the values
sn.barplot( y = 'feature', x = 'importance', data = feature_rank );
```

Gradient boosting also selected the features *balance*, *age*, and *current_campaign* as top features (Figure 6.25) which have maximum information about whether a customer will subscribe or not subscribe to the term deposits.

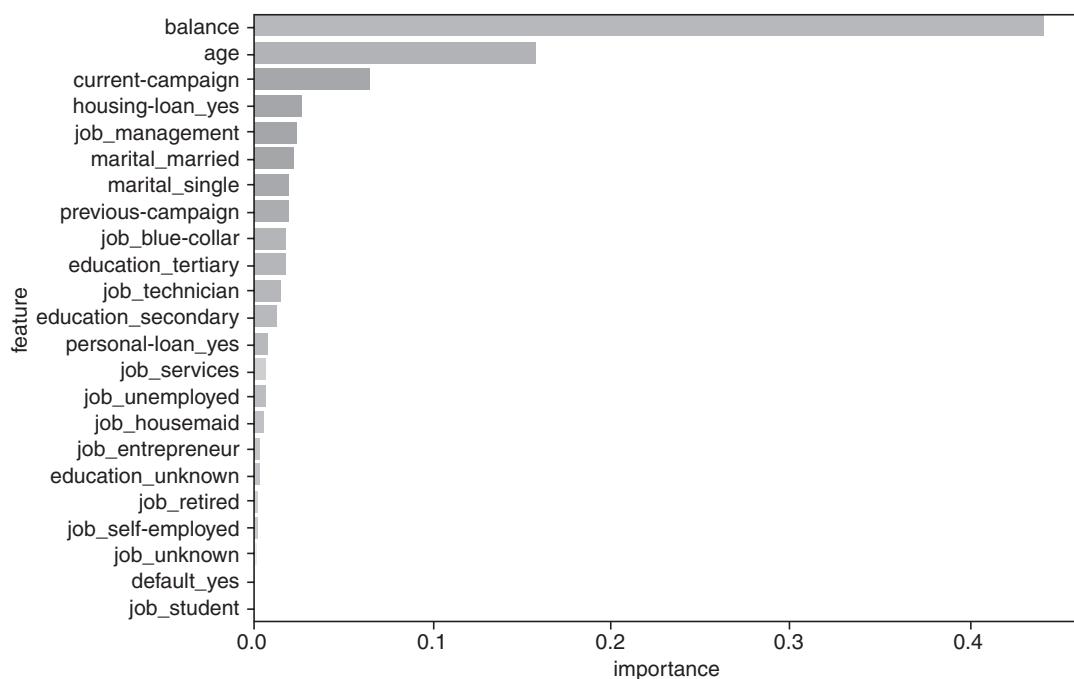


FIGURE 6.25 Feature sorted by their importance values in the gradient boosting model.

CONCLUSION

1. In this chapter, we learnt about how machines learn and the difference between supervised and unsupervised learning.
2. A model is built by defining a cost function and using an optimization algorithm to minimize the cost function.
3. Implementing a gradient descent algorithms to learn a regression model.
4. Building a machine learning model using Python's *sklearn* library.
5. Bias and variance trade-off and how to avoid model overfitting using regularization.
6. Advanced regression models such as LASSO, ridge, and elastic net models.
7. KNN model and finding optimal k values using grid search mechanism.
8. Ensemble techniques such as Random Forest and Boosting techniques: AdaBoost and Gradient Boosting.
9. Finding features importance using ensemble techniques.

EXERCISES

Answer Questions 1 to 8 using the *SAheart* Dataset.

The dataset *SAheart.data* is taken from the link below:

<http://www-stat.stanford.edu/~tibs/ElemStatLearn/datasets/SAheart.data>

The dataset contains records of males in a heart-disease high-risk region of the Western Cape, South Africa. There are roughly two controls per case of CHD. Many of the CHD positive men have undergone

blood pressure reduction treatment and other programs to reduce their risk factors after their CHD event. In some cases the measurements were made after these treatments. These data are taken from a larger dataset, described in Rousseauw et al. (1983), *South African Medical Journal*. It is a tab separated file (csv) and contains the following columns

- *sbp* – Systolic blood pressure
- *tobacco* – Cumulative tobacco (kg)
- *ldl* – Low density lipoprotein cholesterol
- *adiposity*
- *famhist* – Family history of heart disease (Present, Absent)
- *typea* – type-A behavior
- *obesity*
- *alcohol* – Current alcohol consumption
- *age* – Age at onset
- *chd* – Response, coronary heart disease

The *chd* = 1 and *chd* = 0 will be referred as *chd* and *no chd* cases, respectively.

1. Find out the number cases available in the dataset for *chd* and *no chd*. Is it an imbalanced dataset for classification modelling? Plot the findings using a bar plot, where *x* is *chd* or no *chd* and *y* gives the count of samples in each group.
2. To create a balanced dataset, do upsampling of *chd* cases to be same as number of *no chd* cases. For upsampling, use resampling with replacement technique.
3. Maintain two datasets – imbalanced dataset (original one) and balanced dataset – using the upsampling approach. Split each of the datasets into 80:20 splits for creating training and test sets.
4. Build a logistic regression model using both the imbalanced and balanced datasets and compare results. Use K-fold cross-validation ($k = 5$) and ROC AUC score for comparing the model performance.
5. Build a Random Forest model using balanced dataset. Use grid search mechanism to find the most optimal values of the following parameters:

- *n_estimators* = [50, 100, 200, 500]
- *max_depth* = [3, 5, 7, 9]
- *max_features* = [0.1, 0.2, 0.3, 0.5]

Find the best model based on ROC_AUC score.

6. Find features importance from the model built in Question 6 and report the top features which explain 95% of information or variance in the dataset.
7. Build a decision tree model using the features selected above and the following parameters:
 - *max_depth* = 4
 - Criterion = “entropy”

Interpret the rules generated by the decision tree. Explain how these rules can be used in the real world.

Answer Questions 8 to 15 using the *Earnings Manipulation* Dataset.

MCA Technology Solutions Private Limited was established in 2015 in Bangalore with an objective to integrate analytics and technology with business. MCA Technology Solutions helped its clients in areas such as customer intelligence, forecasting, optimization, risk assessment, web analytics, and text mining, and cloud solutions. Risk assessment vertical at MCA technology solutions focuses on problems such as fraud detection and credit scoring. Sachin Kumar, Director at MCA Technology Solutions, Bangalore was approached by one of his clients, a commercial bank, to assist them in detecting earning manipulators among the bank's customers. The bank provided business loans to small and medium enterprises and the value of loan ranged from INR 10 million to 500 million (\$1 = INR 66.82, August 16, 2016). The bank suspected that their customers may be involved in earnings' manipulations to increase their chance of securing a loan.

Saurabh Rishi, the Chief Data Scientist at MCA Technologies, was assigned the task of developing a use case for predicting earning manipulations. He was aware of models such as *Beneish* model that was used for predicting earning manipulations; however, he was not sure of its performance, especially in the Indian context. Saurabh decided to develop his own model for predicting earning manipulations using data downloaded from the Prowess database maintained by the Centre of Monitoring Indian Economy (CMIE). Daniel received information related to earning manipulators from Securities Exchange Board of India (SEBI) and the Lexis Nexis database. Data on 220 companies was collected to develop the model.

The data is available in the file “*Earnings Manipulation 220.csv*”.

Description of the columns is given in Table 6.6. In this table

- netPPE is net Property, Plant, and Equipment. Property Plant and Equipment is the value of all buildings, land, furniture, and other physical capital that a business has purchased to run a business.
- Subscript (t) means value in the current year financial statement and subscript ($t - 1$) means the value in the previous year financial statement.

TABLE 6.6 Columns and their descriptions in the dataset

Column	Description
Company Name	It is a serial number.
Year Ending	The year to which the financial statement belongs to.
Days Sales to Receivables Index (DSRI)	$\text{DSRI} = \frac{\frac{\text{Receivable}_{(t)}}{\text{Sales}_{(t)}}}{\frac{\text{Receivable}_{(t-1)}}{\text{Sales}_{(t-1)}}}$
Gross Margin Index (GMI)	$\text{GMI} = \frac{\frac{\text{Sales}_{(t-1)} - \text{Cost of Goods Sold}_{(t-1)}}{\text{Sales}_{(t-1)}}}{\frac{\text{Sales}_{(t)} - \text{Cost of Goods Sold}_{(t)}}{\text{Sales}_{(t)}}}$

(Continued)

TABLE 6.6 Continued

Column	Description
Asset Quality Index (AQI)	$AQI = \frac{\frac{1 - (Current\ Assets_{(t)} + netPPE_{(t)})}{Total\ Assets_{(t)}}}{\frac{1 - (Current\ Assets_{(t-1)} + netPPE_{(t-1)})}{Total\ Assets_{(t-1)}}}$
Sales Growth Index (SGI)	$SGI = \frac{Sales_{(t)}}{Sales_{(t-1)}}$
Depreciation Index (DEPI)	$DEPI = \frac{\frac{Depreciation\ Expense_{(t-1)}}{(Depreciation\ Expense_{(t-1)} + netPPE_{(t-1)})}}{\frac{Depreciation\ Expense_{(t)}}{(Depreciation\ Expense_{(t)} + netPPE_{(t)})}}$
Sales and General Administrative (SGAI)	$SGAI = \frac{\frac{SGAIExpense_{(t)}}{Sales_{(t)}}}{\frac{SGAIExpense_{(t-1)}}{Sales_{(t-1)}}}$
Accruals to Asset Ratio (ACCR)	$ACCR = \frac{Profit\ after\ Tax_{(t)} - Cash\ from\ Operations_{(t)}}{Total\ Assets_{(t)}}$
Leverage Index (LEVI)	$LEVI = \frac{\frac{(LTD_{(t)} + Current\ Liabilities_{(t)})}{Total\ Assets_{(t)}}}{\frac{(LTD_{(t-1)} + Current\ Liabilities_{(t-1)})}{Total\ Assets_{(t-1)}}}$
MANIPULATOR	1 – Company has manipulated the financial statement (Manipulator) 0 – Company has not manipulated the financial statement (Non-Manipulator)

8. How many cases of manipulators versus non-manipulators are there in the dataset? Draw a bar plot to depict.
9. Create a 80:20 partition, and find how many positives are present in the test data.
10. The number of cases of manipulators are very less compared to non-manipulators. Use upsampling technique to create a balance dataset.
11. Build the following models using balanced dataset and compare the accuracies. Use ROC AUC score for accuracy.
 - Random Forest
 - AdaBoost with Logistic Regression as base estimator
 - Gradient Boosting

12. The number of cases of manipulators are very less compared to non-manipulators. Use downsampling technique to create a balance dataset.
13. Compare the results of using both upsampling and downsampling techniques. Report the best model of all the models.
14. Report the features importance of the best model and explain how many features explain 95% of all the variance in the data.
15. Use Youden's index to find most optimal cutoff probability value for the best model chosen in 13. Report the precision and recall for manipulator and non-manipulator cases.

REFERENCES

1. James G, Witten D, Hastie T, and Tibshirani R (2013). *Introduction to Statistical Learning*, Springer, New York. (<http://www-bcf.usc.edu/~gareth/ISL/>)
2. Tibshirani R (1996). "Regression Shrinkage and Selection via the LASSO", *Journal of the Royal Statistical Society, Series B* (methodological).
3. Hoerl A E and Kennard Kennard (1970). "Ridge Regression: Biased Estimation for Nonorthogonal Problems". *Technometrics*, Vol. 12, pp. 55–67.
4. University of California, Irvine machine learning repository available at: <https://archive.ics.uci.edu/ml/datasets/Bank+Marketing>
5. Distance Metrics: <http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.DistanceMetric.html>
6. Schapire R E (1999). "A Brief Introduction to Boosting", *Proceedings of the Sixth International Conference on Artificial Intelligence*, Stockholm, Sweden. Available at <http://rob.schapire.net/papers/Schapire99c.pdf>.
7. U Dinesh Kumar (2017). *Business Analytics: The Science of Data-Driven Decision Making*, Wiley India Pvt. Ltd.
8. Rousseauw J, du Plessis J, Benade A, Jordaan P, Kotze J, and Ferreira J (1983). Coronary Risk Factor Screening in Three Rural Communities, *South African Medical Journal*, Vol. 64, pp. 430–436.



CHAPTER

7

Clustering

LEARNING OBJECTIVES

After completing this chapter, you will be able to

- Understand the role of clusters and their importance in analytics.
- Learn different types of clustering techniques.
- Understand the use of distance measures such as Euclidean distance in clustering.
- Understand how to find an optimal number of clusters in the data.
- Learn to build clusters using *sklearn* library in Python.

7.1 | OVERVIEW

Clustering is one of the most frequently used analytics applications. It helps data scientists to create a homogeneous group of customers/entities for better management of customers/entities. In many analytics projects, once the data preparation is complete, clustering is usually carried out before applying other analytical models. Clustering is a divide-and-conquer strategy which divides the dataset into homogeneous groups which can be further used to prescribe the right strategy for different groups. In clustering, the objective is to ensure that the variation within a cluster is minimized while the variation between clusters is maximized.

Clustering algorithms are unsupervised learning algorithms (classes are not known *a priori*) whereas classification problems are supervised learning algorithms (where classes are known *a priori* in the training data). Another important difference between clustering and classification is that clustering is descriptive analytics whereas classification is usually a predictive analytics algorithm. The main objective of clustering is to create heterogeneous subsets (clusters) from the original dataset such that records within a cluster are homogeneous and identify the characteristics that differentiate the subsets. For example, if a company wants to increase its brand awareness to appeal to all its existing or possible future customers, it must design a campaign. The company can design a single campaign to address all its customers. But what if its customers have different characteristics such as varied income, age, preferences, profession, gender? The same campaign may not appeal to all of them. The company can think of running multiple campaigns targeting different customer segments. But to know how many such campaigns need to be designed, we need an understanding of how many customer segments

exist and which customers fall under which segment. Clustering as a technique can be used to answer these questions.

7.2 | HOW DOES CLUSTERING WORK?

Clustering algorithms use different distance or similarity or dissimilarity measures to derive different clusters. The type of distance/similarity measure used plays a crucial role in the final cluster formation. Larger distance would imply that observations are far away from one another, whereas higher similarity would indicate that the observations are similar.

To understand the existence of different clusters in a dataset, we will use a small dataset of customers containing their age and income information (File Name: *Income Data.csv*). We can analyze and understand the customer segments that might exist and identify the key attributes of each segment.

First, we will plot these attributes using a scatter plot and visualize how the segments look like. This is possible because we are dealing with only two features: age and income. It may not be possible to visualize the clusters if we are dealing with many features. We will explore other techniques later in Section 7.4 to find the number of segments in high-dimensional datasets.

```
import pandas as pd
customers_df = pd.read_csv("Income Data.CSV")
```

Use the following code to print the first few records from the dataset.

```
customers_df.head(5)
```

	Income	Age
0	41100.0	48.75
1	54100.0	28.10
2	47800.0	46.75
3	19100.0	40.25
4	18200.0	35.80

To visualize the relationship between age and income of customers, we will draw a scatter plot (Figure 7.1).

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sn
%matplotlib inline

sn.lmplot("age", "income", data=customers_df, fit_reg = False,
          size = 4);
```

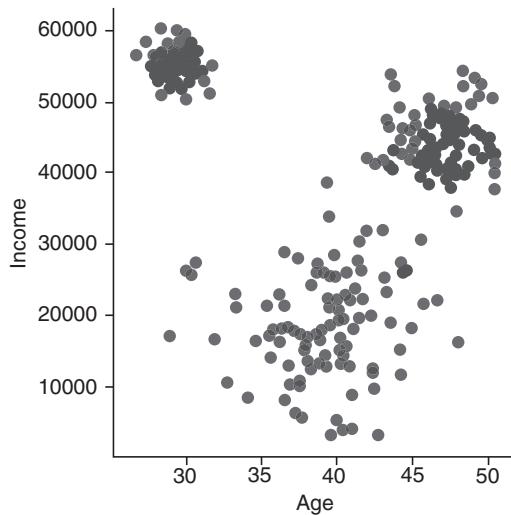


FIGURE 7.1 Income–age scatter plot of customers.

From Figure 7.1, it can be observed that there are three customer segments, which can be described as below:

1. One on the top-left side of the graph, depicting low-age–high-income group.
2. One on the top-right side of the graph, depicting high-age–medium-income group.
3. One on the bottom of the graph, depicting a low-income group, which has an age spread from low to high.

7.2.1 | Finding Similarities Using Distances

Clustering techniques assume that there are subsets in the data that are similar or homogeneous. One approach for measuring similarity is through distances measured using different metrics. Few distance measures used in clustering are discussed in the following sections.

7.2.1.1 Euclidean Distance

Euclidean distance is the radial distance between two observations or records. If there are many attributes (features), then the distance across all attributes is calculated to find out the distance. Euclidean distance between two observations X_1 and X_2 with n features can be calculated as

$$D(X_1, X_2) = \sqrt{\sum(X_{i1} - X_{i2})^2}$$

where X_{i1} is the value of the i^{th} feature for first observation and X_{i2} is the value of i^{th} feature for second observation. For example, the distance between two customers, $customer_1$ and $customer_2$, is calculated as follows:

$$\sqrt{(age_1 - age_2)^2 + (income_1 - income_2)^2}$$

7.2.1.2 Other Distance Metrics

Some of the other widely used distances are as follows:

- Minkowski Distance:** It is the generalized distance measure between two observations.
- Jaccard Similarity Coefficient:** It is a measure used when the data is qualitative, especially when attributes can be represented in binary form.
- Cosine Similarity:** In this, X_1 and X_2 are two n -dimensional vectors. It measures the angle between two vectors (thus called as vector space model).
- Gower's Similarity Coefficient:** The distance and similarity measures that we have discussed so far are valid either for quantitative data or qualitative data. *Gower's similarity coefficient* can be used when both quantitative and qualitative features are present.

7.3 | K-MEANS CLUSTERING

K-means clustering is one of the frequently used clustering algorithms. It is a non-hierarchical clustering method in which the number of clusters (K) is decided *a priori*. The observations in the sample are assigned to one of the clusters (say C_1, C_2, \dots, C_K) based on the distance between the observation and the centroid of the clusters.

The following steps are used in K -means clustering algorithm:

- Decide the value of K (which can be fine-tuned later).
- Choose K observations from the data that are likely to be in different clusters. There are many ways of choosing these initial K values; the easiest approach is to choose observations that are farthest (in one of the parameters of the data).
- The K observations selected in step 2 are the centroids of those clusters.
- For remaining observations, find the cluster closest to the centroid. Add the new observation (say observation j) to the cluster with the closest centroid. Adjust the centroid after adding a new observation to the cluster. The closest centroid is chosen based upon an appropriate distance measure.
- Repeat step 4 until all observations are assigned to a cluster.

For the data described in Section 7.2 (*Income Data.csv*), we can create three clusters, as we know there are three segments from Figure 7.1.

sklearn library has *KMeans* algorithm. Initialize *KMeans* with the number of clusters (k) as an argument and call *fit()* method with the DataFrame, which contains entities and their features that need to be clustered.

```
from sklearn.cluster import KMeans
clusters = KMeans(3)
clusters.fit(customers_df)

KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
       n_clusters=3, n_init=10, n_jobs=1, precompute_distances='auto',
       random_state=None, tol=0.0001, verbose=0)
```

The output variable *clusters.labels_* contains labels that identify the cluster to which an observation belongs. We can concatenate it with the customers' data and verify.

```
customers_df["clusterid"] = clusters.labels_
```

Let us print first five customers with their cluster centers.

```
customers_df[0:5]
```

	Income	Age	Clusterid
0	41100.0	48.75	1
1	54100.0	28.10	2
2	47800.0	46.75	1
3	19100.0	40.25	0
4	18200.0	35.80	0

The three segments are numbered as 0, 1, and 2. The first customer belongs to cluster 1 whereas the second customer belongs to cluster 2 and so on.

7.3.1 | Plotting Customers with Their Segments

We will plot and depict each customer segment with different markings (see Figure 7.2). Plotting the age and income of customers from different segments with different markers. The markers [+, ^, .] will depict the clusters differently.

```
markers = ['+', '^', '.']
sns.lmplot("age", "income",
            data = customers_df,
            hue = "clusterid",
            fit_reg = False,
            markers = markers,
            size = 4);
```

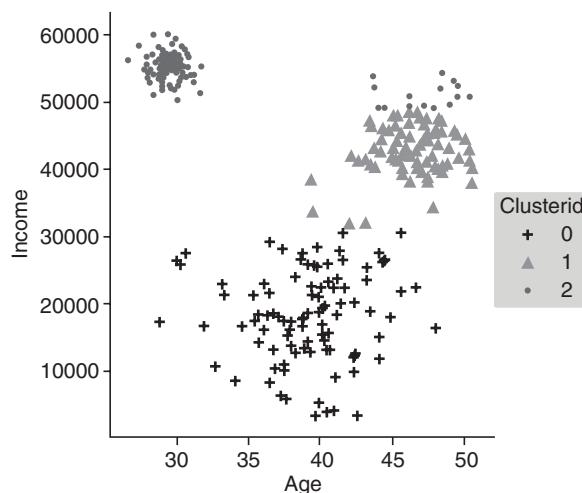


FIGURE 7.2 Three customer segments created by clustering.

The above clusters are mostly segmented based on income. This is because the salary is on a larger scale compared to the age. The scale of age is 0 to 60, while salary ranges from 0 to 50000. For example, difference in age between two customers, with age 20 and 70, is significant, but numerical difference is only 50. Similarly, difference in income between two customers, with income of 10000 and 11000, is not significant; the numerical difference is 1000. So, the distance will always be determined by the difference in salary and not in age. Hence before creating clusters, all features need to be normalized and brought to normalized scale. *StandardScaler* in *sklearn.preprocessing* normalizes all values by subtracting all values from its mean and dividing by standard deviation. Hence

$$X_{\text{normalized}} = \frac{X_i - \bar{X}}{\sigma_X}$$

where \bar{X} is the mean of X and σ_X is the corresponding standard deviation of X feature.

7.3.2 | Normalizing Features

StandardScaler is available under *sklearn.preprocessing* module. Import and initialize the *StandardScaler* and pass DataFrame to *fit()* method to transform the column values.

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaled_customers_df = scaler.fit_transform(customers_df[["age",
                                                       "income"]])
scaled_customers_df[0:5]

array([[ 1.3701637,    0.09718548],
       [-1.3791283,    0.90602749],
       [ 1.10388844,   0.51405021],
       [ 0.23849387,   -1.27162408],
       [-0.35396857,   -1.32762083]])
```

We can create the clusters again using normalized feature set.

```
from sklearn.cluster import KMeans

clusters_new = KMeans(3, random_state=42)
clusters_new.fit(scaled_customers_df)
customers_df["clusterid_new"] = clusters_new.labels_
```

The new segments created can be plotted using the scatter plot and marking each segment differently (Figure 7.3).

```
markers = ['+', '^', '.']

sns.lmplot("age", "income",
            data = customers_df,
            hue = "clusterid_new",
            fit_reg = False,
            markers = markers,
            size = 4);
```

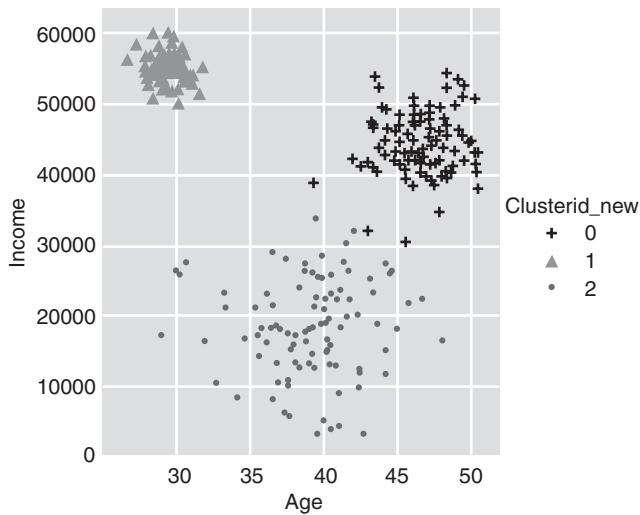


FIGURE 7.3 Three customer segments created by clustering after normalizing features.

Now the clusters seem to have been created properly. The cluster centers are given by `clusters.cluster_centers_`.

```
clusters.cluster_centers_
array([[ 0.5361335,  0.96264295,  0.51632566, -0.61618888],
       [-0.92065895, -0.6352664, -0.86290041, -0.33288365],
       [ 0.7690509, -0.6547531,  0.69314951,  1.89814505]])
```

But these are standardized values. Hence, we can calculate the cluster centers from the original DataFrame using the information “which customers have been allocated to which segment”.

7.3.3 | Cluster Centers and Interpreting the Clusters

Each cluster is defined by its cluster center. The cluster center explains the characteristic of the cluster and helps us to interpret the cluster. For example, in this example of customer segmentation, the cluster center provides the average age and average income of each cluster. Of course, the customer's age and income will vary from the cluster centers and is called the cluster variance.

```
customers_df.groupby('clusterid')[['age',
                                      'income']].agg(['mean',
                                                     'std']).reset_index()
```

Clusterid	Age		Income	
	Mean	Std	Mean	Std
0	31.700435	6.122122	54675.652174	2362.224320
1	39.174479	3.626068	18144.791667	6745.241906
2	46.419101	2.289620	43053.932584	3613.769632

The clusters can be interpreted as follows:

1. $cluster_0$: Customers with mean age of 31 and income of 54k. Low age and high income.
2. $cluster_1$: Customers with mean age of 39 and income of 18K. Mid age and low income.
3. $cluster_2$: Customers with mean age of 46 and income of 43k. High age and medium income.

7.4 | CREATING PRODUCT SEGMENTS USING CLUSTERING

One more example of clustering is product segmentation. We will discuss a case in this section, where a company would like to enter the market with a new beer brand. Before, it decides the kind of beer it will launch, it must understand what kinds of products already exist in the market and what kinds of segments the products address. To understand the segments, the company collects specification of few samples of beer brands. In this section, we will use the dataset *beer.csv* (Feinberg *et al.*, 2012) of beer brands and their corresponding features such as calories, sodium, alcohol, and cost.

Disclaimer: The data here is indicative only and may not represent the actual specification of the products.

7.4.1 | Beer Dataset

The Beer dataset contains about 20 records. First, we will load and print the records.

```
beer_df = pd.read_csv('beer.csv')
beer_df
```

	Name	Calories	Sodium	Alcohol	Cost
0	Budweiser	144	15	4.7	0.43
1	Schlitz	151	19	4.9	0.43
2	Lowenbrau	157	15	0.9	0.48
3	Kronenbourg	170	7	5.2	0.73
4	Heineken	152	11	5.0	0.77
5	Old_Milwaukee	145	23	4.6	0.28
6	Augsberger	175	24	5.5	0.40
7	Srohs_Bohemian_Style	149	27	4.7	0.42
8	Miller_Lite	99	10	4.3	0.43
9	Budweiser_Light	113	8	3.7	0.40
10	Coors	140	18	4.6	0.44
11	Coors_Light	102	15	4.1	0.46
12	Michelob_Light	135	11	4.2	0.50
13	Becks	150	19	4.7	0.76

(Continued)

	Name	Calories	Sodium	Alcohol	Cost
14	Kirin	149	6	5.0	0.79
15	Pabst_Extra_Light	68	15	2.3	0.38
16	Hamms	139	19	4.4	0.43
17	Heilemans_Old_Style	144	24	4.9	0.43
18	Olympia_Goled_Light	72	6	2.9	0.46
19	Schlitz_Light	97	7	4.2	0.47

Each observation belongs to a beer brand and it contains information about the calories, alcohol and sodium content, and the cost.

Before clustering the brands in segments, the features need to be normalized as these are on different scales. So, the first step in creating clusters is to normalize the features in beer dataset.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaled_beer_df = scaler.fit_transform(beer_df[['calories',
                                              'sodium',
                                              'alcohol',
                                              'cost']])
```

7.4.2 | How Many Clusters Exist?

As there are four features, it is not possible to plot and visualize them to understand how many clusters may exist. For high-dimensional data, the following techniques can be used for discovering the possible number of clusters:

1. Dendrogram
2. Elbow method

7.4.2.1 Using Dendrogram

A dendrogram is a cluster tree diagram which groups those entities together that are nearer to each other. A dendrogram can be drawn using the *clustermap()* method in *seaborn*.

```
cmap = sn.cubehelix_palette(as_cmap=True, rot=-.3, light=1)
sn.clustermap(scaled_beer_df, cmap=cmap, linewidths=.2,
               figsize = (8,8));
```

As shown in Figure 7.4, dendrogram reorders the observations based on how close they are to each other using distances (Euclidean). The tree on the left of the dendrogram depicts the relative distance between nodes. For example, the distance between beer brand 10 and 16 is least. They seem to be very similar to each other.

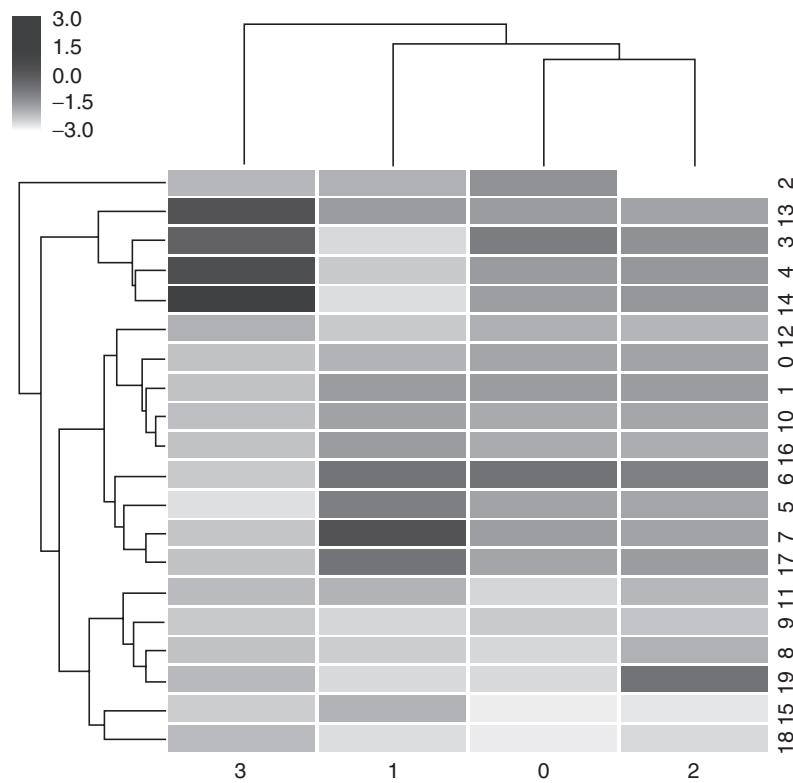


FIGURE 7.4 Dendrogram of beer dataset.

```
beer_df.ix[[10,16]]
```

	Name	Calories	Sodium	Alcohol	Cost
10	Coors	140	18	4.6	0.44
16	Hamm's	139	19	4.4	0.43

It can be observed that both the beer brands *Coors* and *Hamm's* are very similar across all features. Similarly, brands 2 and 18 seem to be most different as the distance is highest. They are represented on two extremes of the dendrogram.

```
beer_df.ix[[2,18]]
```

	Name	Calories	Sodium	Alcohol	Cost
2	Lowenbrau	157	15	0.9	0.48
18	Olympia_Gold_Light	72	6	2.9	0.46

Brand *Lowenbrau* seems to have very low alcohol content. This can be an outlier or maybe a data error. Thus, it can be dropped from the dataset.

The tree structure on the left of the dendrogram indicates that there may be four or five clusters in the dataset. This is only a guideline or indication about the number of clusters, but the actual number of clusters can be determined only after creating the clusters and interpreting them. Creating more number of clusters may give rise to the complexity of defining and managing them. It is always advisable to have a less and reasonable number of clusters that make business sense. We will create four clusters and verify if the clusters explain the product segments clearly and well.

7.4.2.2 Finding Optimal Number of Clusters Using Elbow Curve Method

If we assume all the products belong to only one segment, then the variance of the cluster will be highest. As we increase the number of clusters, the total variance of all clusters will start reducing. But the total variance will be zero if we assume each product is a cluster by itself. So, Elbow curve method considers the percentage of variance explained as a function of the number of clusters. The optimal number of clusters is chosen in such a way that adding another cluster does not change the variance explained significantly.

For a set of records (X_1, X_2, \dots, X_n) , where each observation is a d -dimensional real vector, K-means clustering algorithm segments the observations into k ($\leq n$) sets $S = \{S_1, S_2, \dots, S_k\}$ to minimize the within-cluster sum of squares (WCSS). WCSS is the sum of distances of each point in the cluster to the center of cluster across k clusters. It is given by the following equation:

$$\text{WCSS} = \arg \min_k \sum_{i=1}^k \sum_{X_i \in S_i} \|X_i - \mu_i\|^2$$

where μ_i is the centroid of cluster S_i . If the percentage of variance explained by the clusters is plotted against the number of clusters, the initial increase in the number of clusters will add much information (and explain a lot of variances), but at some point, the *marginal gain in explained variance* will drop, giving an angle to the graph (similar to elbow). The number of clusters indicated at this angle can be chosen to be the most appropriate number of clusters; choosing the number of clusters in this approach is called “elbow criterion”.

Let us create several cluster combinations ranging from one to ten and observe the WCSS in each cluster and how marginal gain in explained variance starts to diminish gradually.

The *inertia_* parameter in *KMeans* cluster algorithms provides the total variance for a particular number of clusters.

The following code iterates and creates clusters ranging from 1 to 10 and captures the total variance in the variable *cluster_errors*.

```
cluster_range = range(1, 10)
cluster_errors = []

for num_clusters in cluster_range:
    clusters = KMeans(num_clusters)
    clusters.fit(scaled_beer_df)
    cluster_errors.append(clusters.inertia_)

plt.figure(figsize=(6, 4))
plt.plot(cluster_range, cluster_errors, marker = "o");
```

Then the *cluster_errors* is plotted against the number of clusters (Figure 7.5).

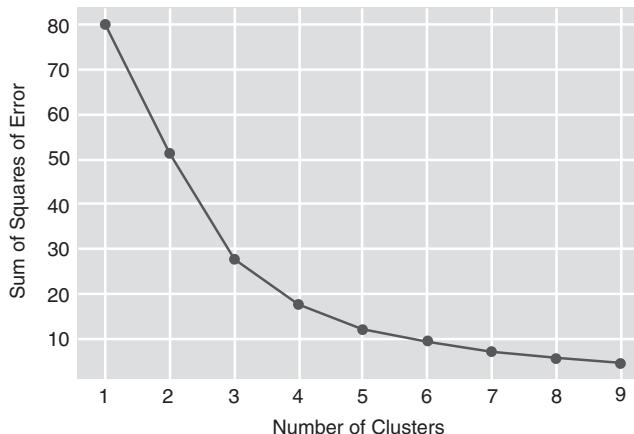


FIGURE 7.5 Elbow diagram.

The plot in Figure 7.5 indicates that the elbow point is at 3, which in turn indicates there might be three clusters existing in the dataset. As mentioned earlier, these techniques provide only a guideline of how many clusters may exist. Here we will discuss three clusters, but the readers can create four clusters and try to interpret them.

7.4.2.3 Normalizing the Features

Before creating the clusters, standardize the features using *StandardScaler*.

```
scaler = StandardScaler()

scaled_beer_df = scaler.fit_transform (beer_df[['calories',
                                              'sodium',
                                              'alcohol',
                                              'cost']])
```

7.4.3 | Creating Clusters

We will set *k* to 3 for running *KMeans* algorithm and create a new column *clusterid* in *beer_df* to capture the cluster number it is assigned to.

```
k = 3

clusters = KMeans(k, random_state = 42)
clusters.fit(scaled_beer_df)
beer_df["clusterid"] = clusters.labels_
```

7.4.4 | Interpreting the Clusters

The three clusters are created and numbered as cluster 0, 1, and 2. We will print each cluster independently and interpret the characteristic of each cluster. We can print each cluster group by filtering the records by *clusterids*.

Cluster 0:

```
beer_df[beer_df.clusterid == 0]
```

	Name	Calories	Sodium	Alcohol	Cost	Clusterid
0	Budweiser	144	15	4.7	0.43	0
1	Schlitz	151	19	4.9	0.43	0
5	Old_Milwaukee	145	23	4.6	0.28	0
6	Augsberger	175	24	5.5	0.40	0
7	Srohs_Bohemian_Style	149	27	4.7	0.42	0
10	Coors	140	18	4.6	0.44	0
16	Hamms	139	19	4.4	0.43	0
17	Heilemans_Old_Style	144	24	4.9	0.43	0

In cluster 0, beers with medium alcohol content and medium cost are grouped together. This is the largest segment and may be targeting the largest segment of customers.

Cluster 1:

```
beer_df[beer_df.clusterid == 1]
```

	Name	Calories	Sodium	Alcohol	Cost	Clusterid
2	Lowenbrau	157	15	0.9	0.48	1
8	Miller_Lite	99	10	4.3	0.43	1
9	Budweiser_Light	113	8	3.7	0.40	1
11	Coors_Light	102	15	4.1	0.46	1
12	Michelob_Light	135	11	4.2	0.50	1
15	Pabst_Extra_Light	68	15	2.3	0.38	1
18	Olympia_Goled_Light	72	6	2.9	0.46	1
19	Schlitz_Light	97	7	4.2	0.47	1

In cluster 1, all the light beers with low calories and sodium content are clustered into one group. This must be addressing the customer segment who wants to drink but are also calorie conscious.

Cluster 2:

```
beer_df[beer_df.clusterid == 2]
```

	Name	Calories	Sodium	Alcohol	Cost	Clusterid
3	Kronenbourg	170	7	5.2	0.73	2
4	Heineken	152	11	5.0	0.77	2
13	Becks	150	19	4.7	0.76	2
14	Kirin	149	6	5.0	0.79	2

These are expensive beers with relatively high alcohol content. Also, the sodium content is low. The costs are high because the target customers could be brand sensitive and the brands are promoted as premium brands.

7.5 | HIERARCHICAL CLUSTERING

Hierarchical clustering is a clustering algorithm which uses the following steps to develop clusters:

1. Start with each data point in a single cluster.
2. Find the data points with the shortest distance (using an appropriate distance measure) and merge them to form a cluster.
3. Repeat step 2 until all data points are merged together to form a single cluster.

The above procedure is called an agglomerative hierarchical cluster. *AgglomerativeClustering* in *sklearn.cluster* provides an algorithm for hierarchical clustering and also takes the number of clusters to be created as an argument.

The agglomerative hierarchical clustering can be represented and understood by using dendrogram, which we had created earlier.

```
from sklearn.cluster import AgglomerativeClustering
```

We will create clusters using *AgglomerativeClustering* and store the new cluster labels in *h_clusterid* variable.

```
h_clusters = AgglomerativeClustering(3)
h_clusters.fit(scaled_beer_df)
beer_df["h_clusterid"] = h_clusters.labels_
```

7.5.1 | Compare the Clusters Created by K-Means and Hierarchical Clustering

We will print each cluster independently and interpret the characteristic of each cluster.

```
beer_df[beer_df.h_clusterid == 0]
```

	Name	Calories	Sodium	Alcohol	Cost	Clusterid	h_clusterid
2	Lowenbrau	157	15	0.9	0.48	1	0
8	Miller_Lite	99	10	4.3	0.43	1	0
9	Budweiser_Light	113	8	3.7	0.40	1	0
11	Coors_Light	102	15	4.1	0.46	1	0
12	Michelob_Light	135	11	4.2	0.50	1	0
15	Pabst_Extra_Light	68	15	2.3	0.38	1	0
18	Olympia_Goled_Light	72	6	2.9	0.46	1	0
19	Schlitz_Light	97	7	4.2	0.47	1	0

```
beer_df[beer_df.h_clusterid == 1]
```

	Name	Calories	Sodium	Alcohol	Cost	Clusterid	h_clusterid
0	Budweiser	144	15	4.7	0.43	0	1
1	Schlitz	151	19	4.9	0.43	0	1
5	Old_Milwaukee	145	23	4.6	0.28	0	1
6	Augsberger	175	24	5.5	0.40	0	1
7	Srohs_Bohemian_Style	149	27	4.7	0.42	0	1
10	Coors	140	18	4.6	0.44	0	1
16	Hamms	139	19	4.4	0.43	0	1
17	Heilemans_Old_Style	144	24	4.9	0.43	0	1

```
beer_df[beer_df.h_clusterid == 2]
```

	Name	Calories	Sodium	Alcohol	Cost	Clusterid	h_clusterid
3	Kronenbourg	170	7	5.2	0.73	2	2
4	Heineken	152	11	5.0	0.77	2	2
13	Becks	150	19	4.7	0.76	2	2
14	Kirin	149	6	5.0	0.79	2	2

Both the clustering algorithms have created similar clusters. Only cluster ids have changed.

CONCLUSION

1. Clustering is an unsupervised learning algorithm that divides the dataset into mutually exclusive and exhaustive subsets (in non-overlapping clusters) that are homogeneous within the group and heterogeneous between the groups.

2. Several distance and similarity measures such as Euclidian distance, Cosine similarity are used in clustering algorithms. Similarity coefficients such as Jaccard coefficient are used for qualitative data, whereas Gower's similarity is used for mixed data i.e. numerical and categorical data.
3. K-means clustering and hierarchical clustering are two popular techniques used for clustering.
4. One of the decisions to be taken during clustering is to decide on the number of clusters. Usually, this is carried out using elbow curve. The cluster number at which the elbow (bend) occurs in the elbow curve is the optimal number of clusters.

EXERCISES

Answer Questions 1 to 5 using the *customerspends.csv* Dataset

An online grocery store has captured amount spent per annum (in Indian rupees) by 20 customers on apparel and beauty and healthcare products and given in the file *customerspends.csv*. It contains the following records.

- Customer – Customer ID
 - Apparel – Amount spent in apparel products
 - Beauty and Healthcare – Amount spent in beauty and healthcare products
1. Create a scatter plot to depict the customer spends on apparel and beauty and healthcare products. Identify number of clusters existing in the dataset.
 2. Normalize the features using *StandardScaler* and plot them in the scatter plot again.
 3. Use dendrogram and elbow method to verify if the number of clusters suggested is same as clusters visible in the scatter plot in Question 1.
 4. Create the number of clusters as suggested by the elbow method using K-means algorithm.
 5. Print the records of customers in each cluster and the cluster center of each cluster separately. Explain the clusters intuitively.

Answer Questions 6 to 10 using the *Online Retail.xlsx* Dataset

The dataset *Online Retail.xlsx* and the description of the data is taken from <https://archive.ics.uci.edu/ml/datasets/online+retail>.

Online Retail.xlsx contains records of transactions occurring between 01/12/2010 and 09/12/2011 for a UK-based and registered non-store online retail. The company mainly sells unique all-occasion gifts. Many customers of the company are wholesalers.

The attributes in the dataset are:

- *InvoiceN* – Invoice number. Nominal, a 6-digit integral number uniquely assigned to each transaction. If this code starts with letter “c”, it indicates a cancellation.
- *StockCode* – Product (item) code. Nominal, a 5-digit integral number uniquely assigned to each distinct product.
- *Description* – Product (item) name. Nominal.
- *Quantity* – The quantities of each product (item) per transaction. Numeric.
- *InvoiceDate* – Invoice Date and time. Numeric, the day and time when each transaction was generated.

- *UnitPrice* – Unit price. Numeric, Product price per unit in sterling.
 - *CustomerID* – Customer number. Nominal, a 5-digit integral number uniquely assigned to each customer.
 - *Country* – Country name. Nominal, the name of the country where each customer resides.
6. Select only the transactions that have occurred in the year 06/01/11 and 11/30/11 and create a new subset of data. For answering questions from 7 to 10, use this subset of data.
 7. Calculate the RFM values for each customer (by customer id). RFM represents
 - R (Recency) – Recency should be calculated as the number of months before he or she has made a purchase from the online store. If he/she made a purchase in the month of November 2011, then the Recency should be 0. If purchase is made in October 2011 then Recency should be 1 and so on and so forth.
 - F (Frequency) – Number of invoices by the customer from 06/01/11 to 11/30/11.
 - M (Monetary Value) – Total spend by the customer from 06/01/11 to 11/30/11.
 8. Use dendrogram and elbow method to identify how many customer segments exist, using the RFM values for each customer created in Question 7.
 9. Create the customer segments with K-means algorithm by using number of clusters is suggested by elbow method.
 10. Print the cluster centers of each customer segment and explain them intuitively.

REFERENCES

1. Daqing Chen, Sai Liang Sain, and Kun Guo (2012). Data mining for the online retail industry: A case study of RFM model-based customer segmentation using data mining, *Journal of Database Marketing and Customer Strategy Management*, Vol. 19, No. 3, pp. 192 to 208, 2012 (Published online before print: 27 August 2012. doi: 10.1057/dbm.2012.17).
2. UC Irvine Machine Learning Repository at <https://archive.ics.uci.edu/ml/index.php>
3. RFM (customer value) [https://en.wikipedia.org/wiki/RFM_\(customer_value\)](https://en.wikipedia.org/wiki/RFM_(customer_value))
4. Dendrogram <https://en.wikipedia.org/wiki/Dendrogram>
5. Robert L. Thorndike (December 1953). "Who Belongs in the Family?". *Psychometrika*. <https://link.springer.com/article/10.1007%2FBF02289263>
6. Sklearn k-means clustering: <https://scikit-learn.org/stable/modules/clustering.html#k-means>
7. Sklearn standard scaler <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html>
8. Feinberg FM, Kinnear TC, and Taylor JR (2012). *Modern Marketing Research: Concepts, Methods and Cases*, Cengage. Data can be downloaded from <http://webuser.bus.umich.edu/feinf/modernmarketingresearch>.



CHAPTER

8

Forecasting

LEARNING OBJECTIVES

After completing this chapter, you will be able to

- Understand the importance of time-series data, forecasting techniques, and their impact on business.
- Learn various components of time-series data such as trend, seasonality, cyclical component, and random component.
- Learn Auto-Regressive (AR), Moving Average (MA), and Auto-Regressive Integrated Moving Average (ARIMA) models
- Learn how to build forecasting models using *statsmodel* API in Python.

8.1 | FORECASTING OVERVIEW

Forecasting is by far the most important and frequently used application of predictive analytics because it has significant impact on both the top line and the bottom line of an organization. Every organization prepares long-range and short-range planning and forecasting demand for product and service is an important input for both long-range and short-range planning. Different capacity planning problems such as manpower planning, machine capacity, warehouse capacity, materials requirements planning (MRP) will depend on the forecasted demand for the product/service. Budget allocation for marketing promotions and advertisements are usually made based on forecasted demand for the product.

Following are few examples of challenging forecasting problems:

1. Demand for products and service is not the only application of forecasting, even manpower planning requires the use of sophisticated models. Indian information technology (IT) companies struggle to manage the right level of manpower for each skill required to manage their business. This would involve forecasting business opportunities, skills required to manage current and future projects, and so on.
2. Boeing 747-400 has more than 6 million parts and several thousand unique parts (Hill, 2011). Forecasting demand for spare parts is important since non-availability of mission-critical parts can result in aircraft on ground (AOG) which can be very expensive for airlines.
3. Walmart sells more than 142,000 products through their supercenters (source: Walmart website; see Reference section). Being a brick-and-mortar retail store, Walmart does not have the

advantages of Amazon.com (being also a market place, Amazon does not have to predict demand for all the products sold through their portal). Walmart must maintain stock for each and every product sold and predict demand for the products as accurately as possible.

8.2 | COMPONENTS OF TIME-SERIES DATA

The time-series data Y_t is a random variable, usually collected at regular time intervals and in chronological order. If the time-series data contains observations of just a single variable (such as demand of a product at time t), then it is termed as *univariate time-series data*. If the data consists of more than one variable, for example, demand for a product at time t , price at time t , amount of money spent by the company on promotion at time t , competitors' price at time t , etc., then it is called *multivariate time-series data*.

A time-series data can be broken into the four following components:

1. **Trend Component (T_t)**: Trend is the consistent long-term upward or downward movement of the data.
2. **Seasonal Component (S_t)**: Seasonal component (measured using seasonality index) is the repetitive upward or downward movement (or fluctuations) from the trend that occurs within a calendar year at fixed intervals (i.e., time between seasons is fixed) such as seasons, quarters, months, days of the week, etc. The upward or downward fluctuation may be caused due to festivals, customs within a society, school holidays, business practices within the market such as "end of season sale", and so on.
3. **Cyclical Component (C_t)**: Cyclical component is fluctuation around the trend line at random interval (i.e., the time between cycles is random) that happens due to macro-economic changes such as recession, unemployment, etc. Cyclical fluctuations have repetitive patterns with time between repetitions of more than a year. Whereas in the case of seasonality, the fluctuations are observed within a calendar year and are driven by factors such as festivals and customs that exist in a society. A major difference between seasonal fluctuation and cyclical fluctuation is that seasonal fluctuation occurs at fixed period within a calendar year, whereas cyclical fluctuations have random time between fluctuations. That is, the periodicity of seasonal fluctuations is constant, whereas the periodicity of cyclical fluctuations is not constant.
4. **Irregular Component (I_t)**: Irregular component is the white noise or random uncorrelated changes that follow a normal distribution with mean value of 0 and constant variance.

There are several forecasting techniques such as moving average, exponential smoothing, and Auto-Regressive Integrated Moving Average (ARIMA) that are used across various industries. Moving average and exponential smoothing predict the future value of a time-series data as a function of past observations. The regression-based models such as auto-regressive (AR), moving average (MA), auto-regressive and moving average (ARMA), auto-regressive integrated moving average (ARIMA) use more sophisticated regression models. It is important to note that using complex mathematical models does not guarantee a more accurate forecast. Simple moving average technique may outperform complex ARIMA models in many cases.

We will discuss all the three techniques, namely, moving average, exponential smoothing, and ARIMA in this chapter.

8.3 | MOVING AVERAGE

To illustrate the moving average technique, we will use the demand dataset for *Kesh*, a shampoo brand which is sold in 100 ml bottles by We Sell Beauty (WSB), a manufacturer and distributor of health and beauty products. The dataset is provided in file *wsb.csv*.

8.3.1 | Loading and Visualizing the Time-Series Dataset

We load the data from file *wsb.csv* onto the DataFrame using *pd.read_csv()*.

```
import pandas as pd
wsb_df = pd.read_csv('wsb.csv')
wsb_df.head(10)
```

Now we print the first 10 records.

	Month	Sale Quantity	Promotion Expenses	Competition Promotion
0	1	3002666	105	1
1	2	4401553	145	0
2	3	3205279	118	1
3	4	4245349	130	0
4	5	3001940	98	1
5	6	4377766	156	0
6	7	2798343	98	1
7	8	4303668	144	0
8	9	2958185	112	1
9	10	3623386	120	0

To visualize *Sale Quantity* over *Month* using *plot()* method from Matplotlib use the following code:

```
import matplotlib.pyplot as plt
import seaborn as sn
%matplotlib inline

plt.figure(figsize=(10, 4))
plt.xlabel("Months")
plt.ylabel("Quantity")
plt.plot(wsb_df['Sale Quantity']);
```

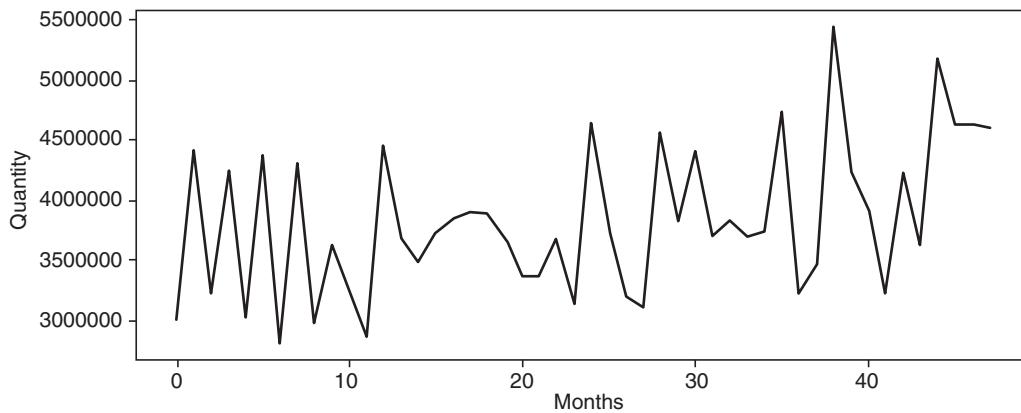


FIGURE 8.1 Sales quantity over 48 months.

In Figure 8.1, the sales quantity (vertical axis) shows a lot of fluctuations over the months. But there seems to be an increasing trend as well.

We now print the summary of the dataset using the *info()* method of the DataFrame.

```
wsb_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 48 entries, 0 to 47
Data columns (total 4 columns):
Month           48 non-null int64
Sale Quantity   48 non-null int64
Promotion Expenses 48 non-null int64
Competition Promotion 48 non-null int64
dtypes: int64(4)
memory usage: 1.6 KB
```

The dataset contains quantity of 100 ml bottles sold during each month for a period of 48 months, promotion expenses incurred by the company, and whether the competition was on promotion (value of 1 implies that the competition was on promotion and 0 otherwise).

8.3.2 | Forecasting Using Moving Average

Moving average is one of the simplest forecasting techniques which forecasts the future value of a time-series data using average (or weighted average) of the past N observations. Forecasted value for time $t+1$ using the simple moving average is given by

$$F_{t+1} = \frac{1}{N} \sum_{k=t+1-N}^t Y_k \quad (8.1)$$

Pandas has a function `rolling()` which can be used with an aggregate function like `mean()` for calculating moving average for a time window. For example, to calculate 12 month's moving average using last 12 months' data starting from last month (previous period), `rolling()` will take a parameter `window`, which is set to 12 to indicate moving average of 12-months data, and then use Pandas' `shift()` function, which takes parameter 1 to specify that the 12-months data should start from last month. `shift(1)` means calculating moving average for the specified window period starting from previous observation (in this case last month).

```
wsb_df['mavg_12'] = wsb_df['Sale Quantity'].rolling(window = 12).mean().shift(1)
```

To display values up to 2 decimal points, we can use `pd.set_option` and floating format `% .2f`.

```
pd.set_option('display.float_format', lambda x: '%.2f' % x)
wsb_df[['Sale Quantity', 'mavg_12']][36:]
```

Now print the forecasted value from the month 37 onwards.

	Sale Quantity	mavg_12
36	3216483	3928410.33
37	3453239	3810280.00
38	5431651	3783643.33
39	4241851	3970688.42
40	3909887	4066369.08
41	3216438	4012412.75
42	4222005	3962369.58
43	3621034	3946629.42
44	5162201	3940489.50
45	4627177	4052117.17
46	4623945	4130274.75
47	4599368	4204882.33

We use the following code to plot actual versus the predicted values from moving average forecasting:

```
plt.figure(figsize=(10, 4))
plt.xlabel("Months")
plt.ylabel("Quantity")
plt.plot(wsb_df['Sale Quantity'][12:]);
plt.plot(wsb_df['mavg_12'][12:], '.');
plt.legend();
```

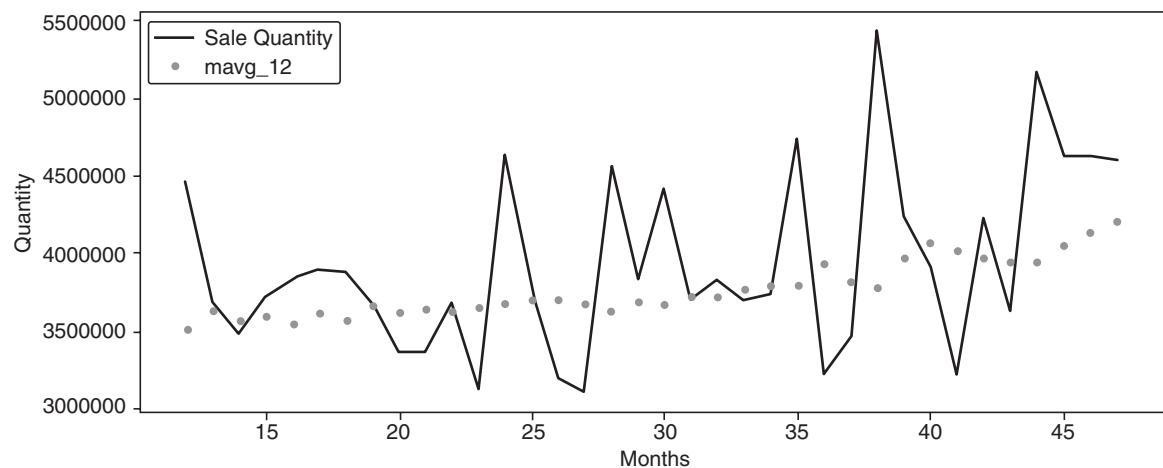


FIGURE 8.2 Actual versus forecasted sales quantity using moving average.

As observed in Figure 8.2, there is an increasing trend in sales quantity over the months.

8.3.3 | Calculating Forecast Accuracy

Root mean square error (RMSE) and mean absolute percentage error (MAPE) are the two most popular accuracy measures of forecasting. We will be discussing these measures in this section.

8.3.3.1 Mean Absolute Percentage Error

Mean absolute percentage error (MAPE) is the average of absolute percentage error. Assume that the validation data has n observations and forecasting is carried out on these n observations. The mean absolute percentage error is given by

$$\text{MAPE} = \frac{1}{n} \sum_{t=1}^n \frac{|Y_t - F_t|}{Y_t} \times 100\% \quad (8.2)$$

MAPE is one of the popular forecasting accuracy measures used by practitioners since it expresses the average error in percentage terms and is easy to interpret. Since MAPE is dimensionless, it can be used for comparing different models with varying scales.

The following custom method `get_mape()` takes the series of actual values and forecasted values, and returns the MAPE.

```
import numpy as np

def get_mape(actual, predicted):
    y_true, y_pred = np.array(actual), np.array(predicted)
    return np.round( np.mean(np.abs((actual - predicted) / actual)) * 100, 2 )
```

We invoke the above method using column values of `wbd_df` DataFrame. “Sale Quantity” is passed as actual and `mavg_12` is passed as predicted values. Records from the 37th month are used to calculate MAPE.

```
get_mape( wsb_df['Sale Quantity'][36:].values,
          wsb_df['mavg_12'][36:].values)
```

The MAPE in this case is 14.04. So, forecasting using moving average gives us a MAPE of 14.04%.

8.3.3.2 Root Mean Square Error

Root mean square error (RMSE) is the square root of average of squared error calculated over the validation dataset, and is the standard deviation of the errors for unbiased estimator. RMSE is given by

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{t=1}^n (Y_t - F_t)^2} \quad (8.3)$$

Lower RMSE implies better prediction. However, it depends on the scale of the time-series data. MSE (*Mean Squared Error*) can be calculated using `mean_squared_error()` method in `sklearn.metrics`. We can pass MSE value to `np.sqrt()` to calculate RMSE.

```
from sklearn.metrics import mean_squared_error
np.sqrt(mean_squared_error(wsb_df['Sale Quantity'][36:].values,
                           wsb_df['mavg_12'][36:].values))
```

The RMSE in this case is 734725.83. So, the RMSE of the moving average model indicates that the prediction by the models has a standard deviation of 734725.83.

8.3.4 | Exponential Smoothing

One of the drawbacks of the simple moving average technique is that it gives equal weight to all the previous observations used in forecasting the future value. Exponential smoothing technique (also known as simple exponential smoothing; SES) assigns differential weights to past observations.

$$F_{t+1} = \alpha Y_t + (1-\alpha)F_t \quad (8.4)$$

where α is called the smoothing constant, and its value lies between 0 and 1. F_{t+1} is the forecasted value at time $t+1$ using actual value Y_t at time t and forecasted values F_t of time t . But the model applies differential weights to both the inputs using smoothing constant α .

The `ewm()` method in Pandas provides the features for computing the exponential moving average taking `alpha` as a parameter.

```
wsb_df['ewm'] = wsb_df['Sale Quantity'].ewm( alpha = 0.2 ).mean()
```

Set the floating point formatting up to 2 decimal points.

```
pd.options.display.float_format = '{:.2f}'.format
```

Use the following code to display the records from the 37th month.

```
wsb_df[36:]
```

Month	Sale Quantity	Promotion Expenses	Competition Promotion	mavg_12	Ewm
36	37	3216483	121	1	3928410.33
37	38	3453239	128	0	3810280.00
38	39	5431651	170	0	3783643.33
39	40	4241851	160	0	3970688.42
40	41	3909887	151	1	4066369.08
41	42	3216438	120	1	4012412.75
42	43	4222005	152	0	3962369.58
43	44	3621034	125	0	3946629.42
44	45	5162201	170	0	3940489.50
45	46	4627177	160	0	4052117.17
46	47	4623945	168	0	4130274.75
47	48	4599368	166	0	4204882.33

Now calculate MAPE of the model using records from 37th month.

```
get_mape(wsb_df[['Sale Quantity']][36:].values,
          wsb_df[['ewm']][36:].values)
```

The MAPE of the model is 11.15.

So, forecasting using exponential smoothing has about 11.15% error (MAPE) from the actual values. It is an improvement compared to the simple moving average model. Let us plot the output to view the difference between the forecasted and actual sales quantity using exponential moving average.

```
plt.figure( figsize=(10, 4) )
plt.xlabel( "Months" )
plt.ylabel( "Quantity" )
plt.plot( wsb_df['Sale Quantity'][12:] );
plt.plot( wsb_df['ewm'][12:], '.' );
plt.legend();
```

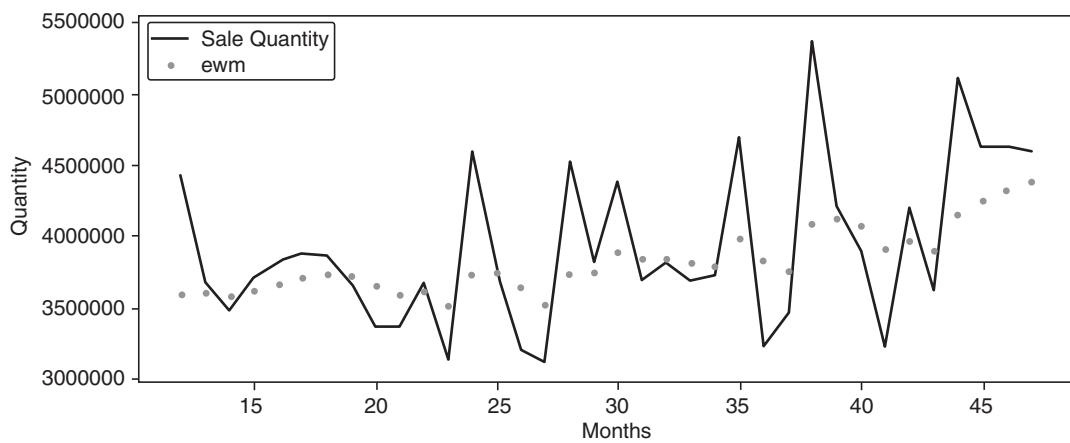


FIGURE 8.3 Forecasted versus actual sales quantity using exponential moving average.

In Figure 8.3, prediction by exponential moving average is shown by the dotted line and shows an increasing trend over the months. Moving average and simple exponential smoothing (SES) assume a fairly steady time-series data with no significant trend, seasonal or cyclical components, that is, the data is stationary. However, many dataset will have trend and seasonality.

8.4 | DECOMPOSING TIME SERIES

The time-series data can be modelled as addition or product of trend, seasonality, cyclical, and irregular components. The additive time-series model is given by

$$Y_t = T_t + S_t + C_t + I_t$$

The additive models assume that the seasonal and cyclical components are independent of the trend component. Additive models are not very common, since in many cases the seasonal component may not be independent of the trend.

The multiplicative time-series model is given by

$$Y_t = T_t \times S_t \times C_t \times I_t$$

Multiplicative models are more common and are a better fit for many datasets. For example, the seasonality effect on sales during festival times like Diwali does not result in constant increase in sales over the years. For example, the increase in sales of cars during festival season is not just 100 units every year. The seasonality effect has a multiplicative effect on sales based on the trend over the years like 10% additional units based on the trend in the current year. So, in many cases the seasonality effect is multiplied with the trend and not just added as in additive model.

In many cases, while building a forecasting model, only trend and seasonality components are used. To estimate the cyclical component, we will need a large dataset. For example, typical period of business cycles is about 58 months as per Investopedia (Anon, 2018). So, to understand the effect of cyclic component we will need observations spanning more than 10 years. Most of the times, data for such a long duration is not available.

For decomposing a time-series data, we can leverage the following libraries:

1. `statsmodel.tsa` provides various features for time-series analysis.
2. `seasonal_decompose()` in `statsmodel.tsa.seasonal` decomposes a time series into trend, seasonal, and residuals. It takes *frequency* parameters; for example, the frequency is 12 for monthly data.

The `plot()` function will render the trend, seasonal, and residuals, as shown in the following code:

```
from statsmodels.tsa.seasonal import seasonal_decompose

ts_decompose = seasonal_decompose(np.array(wsb_df['Sale Quantity']),
                                   model='multiplicative',
                                   freq = 12)

## Plotting the deocomposed time series components
ts_plot = ts_decompose.plot()
```

To capture the seasonal and trend components after time-series decomposition we use the following code. The information can be read from two variables `seasonal` and `trend` in `ts_decompose`.

```
wsb_df['seasonal'] = ts_decompose.seasonal
wsb_df['trend'] = ts_decompose.trend
```

Sales quantity in `wsb-df` has trend, which is increasing over a period. Also it has seasonality and seasonality index varies from 0.9 to 1.1 over the trend (Figure 8.4).

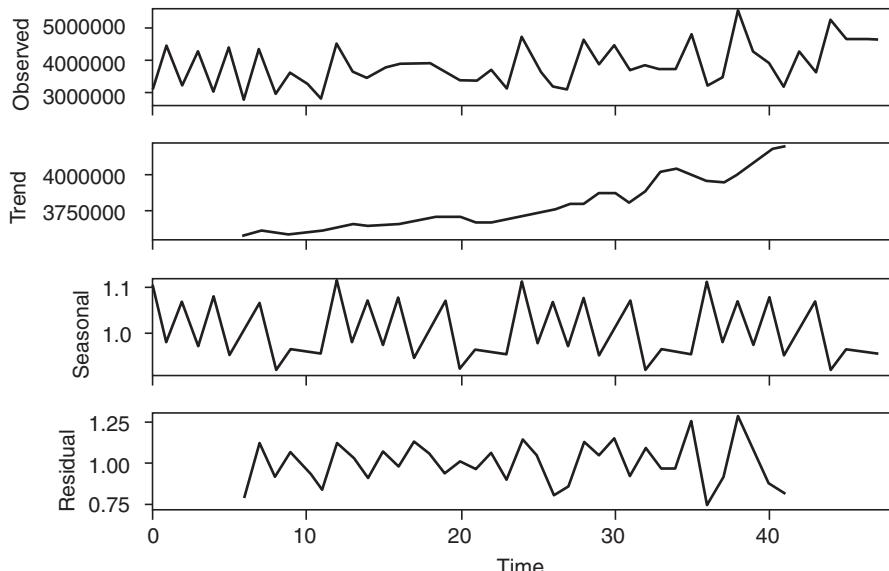


FIGURE 8.4 Plots of decomposed time-series components

8.5 | AUTO-REGRESSIVE INTEGRATED MOVING AVERAGE MODELS

Auto-regressive (AR) and moving average (MA) models are popular models that are frequently used for forecasting. AR and MA models are combined to create models such as auto-regressive moving average (ARMA) and auto-regressive integrated moving average (ARIMA) models. The initial ARMA and ARIMA models were developed by Box and Jenkins in 1970 (Box and Jenkins, 1970). ARMA models are basically regression models; auto-regression simply means regression of a variable on itself measured at different time periods. We will discuss each component in the subsequent sections.

8.5.1 | Auto-Regressive (AR) Models

Auto-regression is a regression of a variable on itself measured at different time points. Auto-regressive model with lag 1, AR (1), is given by

$$Y_{t+1} = \mu + \beta Y_t + \varepsilon_{t+1} \quad (8.5)$$

Equation (8.5) can be generalized to include p lags on the right-hand side and is called a AR (p) model. Equation (8.5) can be re-written as

$$Y_{t+1} - \mu = \beta(Y_t - \mu) + \varepsilon_{t+1} \quad (8.6)$$

where ε_{t+1} is a sequence of uncorrelated residuals assumed to follow the normal distribution with zero mean and constant standard deviation. $(Y_t - \mu)$ can be interpreted as a deviation from mean value μ ; it is known as *mean centered series*.

One of the important tasks in using the AR model in forecasting is model identification, which is, identifying the value of p (the number of lags). One of the standard approaches used for model identification is using auto-correlation function (ACF) and partial auto-correlation function (PACF).

8.5.1.1 ACF

Auto-correlation of lag k is the correlation between Y_t and Y_{t-k} measured at different k values (e.g., Y_t and Y_{t-1} or Y_t and Y_{t-2}). A plot of auto-correlation for different values of k is called an auto-correlation function (ACF) or *correlogram*.

`statsmodels.graphics.tsaplots.plot_acf` plots the autocorrelation plot.

8.5.1.2 PACF

Partial auto-correlation of lag k is the correlation between Y_t and Y_{t-k} when the influence of all intermediate values (Y_{t-1} , Y_{t-2} , ..., Y_{t-k+1}) is removed (partial out) from both Y_t and Y_{t-k} . A plot of partial auto-correlation for different values of k is called partial auto-correlation function (PACF).

`statsmodels.graphics.tsaplots.plot_pacf` plots the partial auto-correlation plot.

For applying the AR model, we will use another dataset described in the following example.

8.1 EXAMPLE**Aircraft Spare Parts**

Monthly demand for avionic system spares used in Vimana 007 aircraft is provided in *vimana.csv*. Build an ARMA model based on the first 30 months of data and forecast the demand for spares for months 31 to 37. Comment on the accuracy of the forecast.

Solution:

We first read the dataset from *vimana.csv* onto a DataFrame and print the first 5 records.

```
vimana_df = pd.read_csv('vimana.csv')
vimana_df.head(5)
```

	Month	Demand
0	1	457
1	2	439
2	3	404
3	4	392
4	5	403

Now print the metadata from *vimana.csv*.

```
vimana_df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 37 entries, 0 to 36
Data columns (total 2 columns):
Month      37 non-null int64
demand     37 non-null int64
dtypes: int64(2)
memory usage: 672.0 bytes
```

Draw the ACF plot to show auto-correlation upto lag of 20 using the following code:

```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

# Show autocorrelation upto lag 20
acf_plot = plot_acf(vimana_df.demand,
                     lags=20)
```

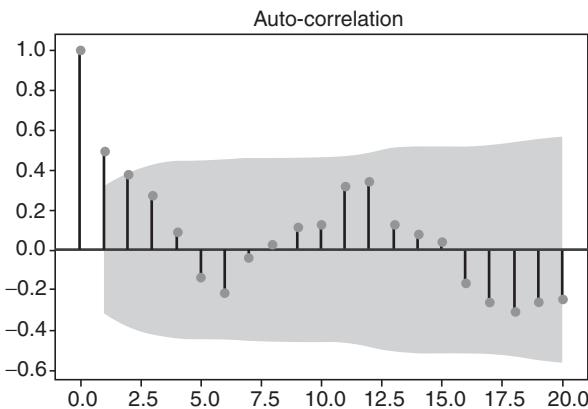


FIGURE 8.5 ACF plot for aircraft spare parts.

In Figure 8.5, the shaded area represents the upper and lower bounds for critical values, where the null hypothesis cannot be rejected (auto-correlation value is 0). So, as can be seen from Figure 8.5, null hypothesis is rejected only for lag = 1 (i.e., auto-correlation is statistically significant for lag 1).

We draw the PACF plot with lag up to 20.

```
pacf_plot = plot_pacf( vimana_df.demand,
                      lags=20 )
```

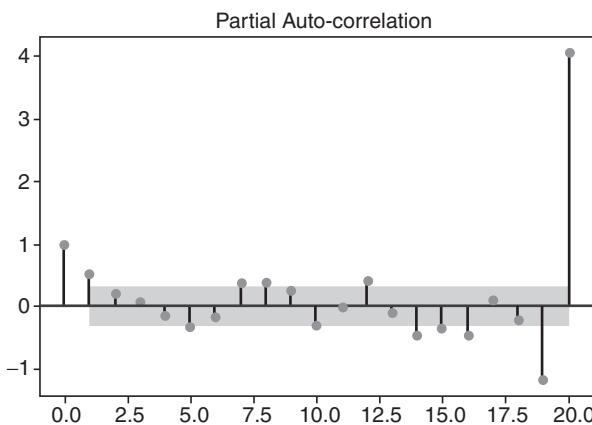


FIGURE 8.6 PACF plot for aircraft spare parts.

In Figure 8.6, the shaded area represents the upper and lower bounds for critical values, where the null hypothesis cannot be rejected. So, for lag = 1 the partial auto-correlation is significant.

To select the appropriate p in the AR model, the following thumb rule can be used:

1. The partial auto-correlation is significant for first p -values (first p lags) and cuts off to zero.
2. The ACF decreases exponentially. (Signs of stationarity)

In the above case, based on ACF and PACF plots, AR with lag 1, AR(1), can be used.

8.5.1.3 Building AR Model

The `statsmodels.tsa.arima_model.ARIMA` can be used to build AR model. It takes the following two parameters:

1. `endog`: list of values – It is the endogenous variable of the time series.
2. `order`: The (p, d, q) – ARIMA model parameters. Order of AR is given by the value p , the order of integration is d , and the order of MA is given by q .

We will set d and q to 0 and use $p = 1$ for AR(1) model, and we will use only 30 months of data for building the model and forecast the next six months for measuring accuracy.

Building the model and printing the accuracy.

```
from statsmodels.tsa.arima_model import ARIMA
```

```
arima = ARIMA(vimana_df.demand[0:30].astype(np.float64).as_matrix(),
              order = (1,0,0))
ar_model = arima.fit()
```

Printing the model summary.

```
ar_model.summary2()
```

Model:	ARMA	BIC:	375.7336
Dependent Variable:	y	Log-Likelihood:	-182.77
Date:	2019-03-01 13:40	Scale:	1.0000
No. Observations:	30	Method:	css-mle
Df Model:	2	Sample:	0
Df Residuals:	28		0
Converged:	1.0000	S.D. of innovations:	106.593
No. Iterations:	14.0000	HQIC:	372.875
AIC:	371.5300		

	Coef.	Std.Err.	t	P > t	[0.025	0.975]
const	513.4433	35.9147	14.2962	0.0000	443.0519	583.8348
ar.L1.y	0.4726	0.1576	2.9995	0.0056	0.1638	0.7814

	Real	Imaginary	Modulus	Frequency
AR.1	2.1161	0.0000	2.1161	0.0000

The model summary indicates the AR with lag 1 is significant variables in the model. The corresponding *p-value* is less than 0.05 (0.0056).

8.5.1.4 Forecast and Measure Accuracy

Forecast the demand for the months 31 to 37. The index will be (month index – 1).

```
forecast_31_37 = ar_model.predict(30, 36)

forecast_31_37

array([480.15343682, 497.71129378, 506.00873185, 509.92990963,
      511.78296777, 512.65868028, 513.07252181])
```

```
get_mape( vimana_df.demand[30:],
          forecast_31_37 )
```

The MAPE of the AR model with lag 1 is 19.12.

8.5.2 | Moving Average (MA) Processes

MA processes are regression models in which the past residuals are used for forecasting future values of the time-series data.

A moving average process of lag 1 can be written as

$$Y_{t+1} = \alpha_1 \varepsilon_t + \varepsilon_{t+1} \quad (8.7)$$

The model in Eq. (8.7) can be generalized to q lags. The value of q (number of lags) in a moving average process can be identified using the following rules (Yaffee and McGee, 2000):

1. Auto-correlation value is significant for first q lags and cuts off to zero.
2. The PACF decreases exponentially.

We can build an MA model with q value as 1 and print the model summary.

```
arima = ARIMA(vimana_df.demand[0:30].astype(np.float64).as_matrix(),
               order = (0,0,1))
ma_model = arima.fit()
ma_model.summary2()
```

Model:	ARMA	BIC:	378.7982
Dependent Variable:	y	Log-Likelihood:	-184.30
Date:	2019-03-01 13:40	Scale:	1.0000

(Continued)

No. Observations:	30	Method:	css-mle
Df Model:	2	Sample:	0
Df Residuals:	28		0
Converged:	1.0000	S.D. of innovations:	112.453
No. Iterations:	15.0000	HQIC:	375.939
AIC:	374.5946		

	Coef.	Std.Err.	t	P > t	[0.025	0.975]
const	516.5440	26.8307	19.2520	0.0000	463.9569	569.1312
ma.L1.y	0.3173	0.1421	2.2327	0.0337	0.0388	0.5958

	Real	Imaginary	Modulus	Frequency
MA.1	-3.1518	0.0000	3.1518	0.5000

As per the model summary, moving average with lag 1 is statistically significant as the corresponding *p-value* is less than 0.05. Use the following code to measure the accuracy with the forecasted values of six periods.

```
forecast_31_37 = ma_model.predict(30, 36)
get_mape(vimana_df.demand[30:],
          forecast_31_37 )
```

17.8

The MAPE of the MA model with lag 1 is 17.8.

8.5.3 | ARMA Model

Auto-regressive moving average (ARMA) is a combination auto-regressive and moving average process. ARMA(p, q) process combines AR(p) and MA(q) processes.

The values of p and q in an ARMA process can be identified using the following thumb rules:

1. Auto-correlation values are significant for first q values (first q lags) and cuts off to zero.
2. Partial auto-correlation values are significant for first p values and cuts off to zero.

Based on the ACF and PACF plots in the previous section, we will develop ARMA(1, 1) model using the following codes:

```
arima = ARIMA( vimana_df.demand[0:30].astype(np.float64).as_
               matrix(), order = (1, 0, 1))
arma_model = arima.fit()
arma_model.summary2()
```

Model:	ARMA	BIC:	377.2964
Dependent Variable:	y	Log-Likelihood:	-181.85
Date:	2019-03-01 13:41	Scale:	1.0000
No. Observations:	30	Method:	css-mle
Df Model:	3	Sample:	0
Df Residuals:	27		0
Converged:	1.0000	S.D. of innovations:	103.223
No. Iterations:	21.0000	HQIC:	373.485
AIC:	371.6916		

	Coef.	Std.Err.	t	P > t	[0.025	0.975]
const	508.3995	45.3279	11.2160	0.0000	419.5585	597.2405
ar.L1.y	0.7421	0.1681	4.4158	0.0001	0.4127	1.0715
MA.L1.y	-0.3394	0.2070	-1.6401	0.1126	-0.7451	0.0662

	Real	Imaginary	Modulus	Frequency
AR.1	1.3475	0.0000	1.3475	0.0000
MA.1	2.9461	0.0000	2.9461	0.0000

The model summary indicates that moving average with lag 1 is not significant (p-value is more than 0.05), when both auto regressive with lag1 is also used in the model.

```
forecast_31_37 = arma_model.predict(30, 36)
get_mape(vimana_df.demand[30:], 
          forecast_31_37 )
```

20.27

The MAPE of the MA model with lag 1 is 20.27. Since MA lag is not significant, we will use AR(1) model [or MA(1) model].

8.5.4 | ARIMA Model

ARMA models can be used only when the time-series data is stationary. ARIMA models are used when the time-series data is non-stationary. Time-series data is called stationary if the mean, variance, and covariance are constant over time. ARIMA model was proposed by Box and Jenkins (1970) and thus is also known as Box–Jenkins methodology. ARIMA has the following three components and is represented as ARIMA (p, d, q):

1. AR component with p lags AR(p).
2. Integration component (d).
3. MA with q lags, MA(q).

The main objective of the integration component is to convert a non-stationary time-series process to a stationary process so that the AR and MA processes can be used for forecasting. When the data is non-stationary, the ACF will not cut-off to zero quickly; rather ACF may show a very slow decline.

8.5.4.1 What is Stationary Data?

Time-series data should satisfy the following conditions to be stationary:

1. The mean values of Y_t at different values of t are constant.
2. The variances of Y_t at different time periods are constant (Homoscedasticity).
3. The covariance of Y_t and Y_{t-k} for different lags depend only on k and not on time t .

We will use dataset provided in *store.xls* (described in Example 8.2) for performing the stationary test and developing ARIMA model.

8.2 EXAMPLE

Daily demand for a product in a store for the past 115 days is provided in *store.xls*. Develop an appropriate ARIMA model that can be used for forecasting demand for Omelette.

Solution: The data is available in excel file and read into DataFrame using *read_excel* api.

```
store_df = pd.read_excel('store.xls')
```

```
store_df.head(5)
```

	Date	Demand
0	2014-10-01	15
1	2014-10-02	7
2	2014-10-03	8
3	2014-10-04	10
4	2014-10-05	13

Now print the metadata of the DataFrame.

```
store_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 115 entries, 0 to 114
Data columns (total 2 columns):
Date      115 non-null datetime64[ns]
demand    115 non-null int64
```

8.2 EXAMPLE (Continued)

```
dtypes: datetime64[ns] (1), int64 (1)
memory usage: 1.9 KB
```

The dataset contains 115 days of demand per day data. We can convert the column into *DateTime* index, which is a default input to time-series models. Creating a *DateTime* index will make sure that the data is ordered by date or time.

```
store_df.set_index(pd.to_datetime(store_df.Date), inplace=True)
store_df.drop('Date', axis = 1, inplace = True)
store_df[-5:]
```

Now we print the last 5 records to make sure that index is created correctly and is sorted by date.

Date	Demand
2015-01-19	18
2015-01-20	22
2015-01-21	22
2015-01-22	21
2015-01-23	17

Finally, plot the demand against the date using *plot()* method in matplotlib.

```
plt.figure( figsize=(10, 4) )
plt.xlabel( "Date" )
plt.ylabel( "Demand" )
plt.plot( store_df.demand );
```

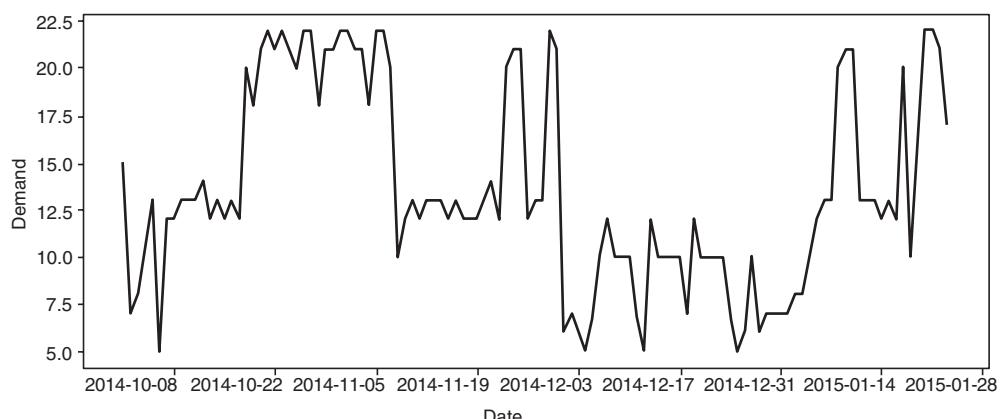


FIGURE 8.7 Daily store demand.

8.2 EXAMPLE (Continued)

Figure 8.7 indicates the trend of demand for omelette over dates. It is not very apparent from the trend if the series is stationary or not.

Now we will draw the ACF plot to verify stationarity of the time series.

```
acf_plot = plot_acf( store_df.demand,
                     lags=20 )
```

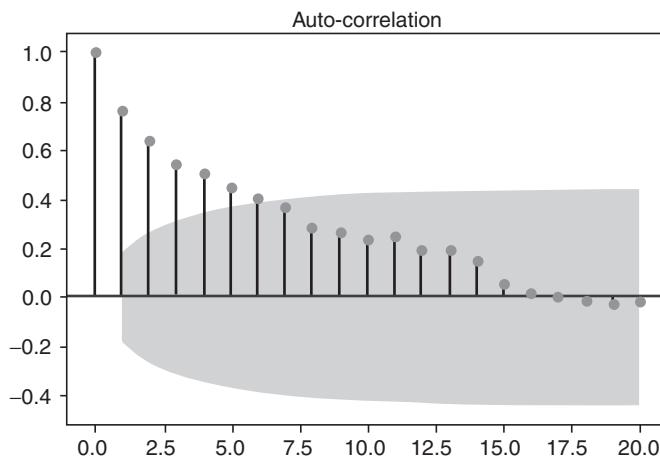


FIGURE 8.8 ACF plot for store demand.

In Figure 8.8, the slow decline of auto-correlations for different lags indicates that the series is not stationary.

8.5.4.2 Dicky-Fuller Test

To find out if a time series is stationary, Dickey–Fuller (Dickey and Fuller, 1979) test can also be conducted. Dickey–Fuller test checks whether the β in Eq. (8.5) is equal to 1 or less than equal to 1. It is a hypothesis test in which the null hypothesis and alternative hypothesis are given by

$$H_0: \beta = 1 \text{ (the time series is non-stationary)}$$

$$H_A: \beta < 1 \text{ (the time series is stationary)}$$

`statsmodels.tsa.stattools.adfuller` is a Dicky–Fuller test and returns test statistics and p -value for the test of the null hypothesis. If the p -value is less than 0.05, the time series is stationary.

```
from statsmodels.tsa.stattools import adfuller
```

```
def adfuller_test(ts):
    adfuller_result = adfuller(ts, autolag=None)
    adfuller_out = pd.Series(adfuller_result[0:4],
                            index=['Test Statistic',
                                   'p-value',
                                   'Lags Used',
                                   'Number of Observations Used'])
    print(adfuller_out)
    adfuller_text(stone_df.demand)
```

```
Test Statistic           -1.65
p-value                 0.46
Lags Used              13.00
Number of Observations Used 101.00
dtype: float64
```

The *p*-value (>0.05) indicates that we cannot reject the null hypothesis and hence, the series is not stationary.

Differencing the original time series is an usual approach for converting the non-stationary process into a stationary process (called *difference stationarity*). For example, the first difference ($d = 1$) is the difference between consecutive values of the time series (Y_t). That is, the first difference is given by

$$\Delta Y_t = Y_t - Y_{t-1} \quad (8.8)$$

8.5.4.3 Differencing

First difference between consecutive Y_t values can be computed by subtracting the previous day's demand from that day's demand. We can use *shift()* function in Pandas to shift the values before subtracting.

```
store_df['demand_diff'] = store_df.demand - store_df.demand.shift(1)
```

```
store_df.head(5)
```

	demand	demand_diff
Date		
2014-10-01	15	nan
2014-10-02	7	-8.00
2014-10-03	8	1.00
2014-10-04	10	2.00
2014-10-05	13	3.00

The differencing for the first day is *nan* as we do not have previous days' demand. We can drop this value from the observation before verifying stationarity.

```
store_diff_df = store_df.dropna()
```

Let us plot the first-order difference values.

```
plt.figure(figsize=(10, 4))
plt.xlabel("Date")
plt.ylabel("First Order Differences")
plt.plot(store_diff_df.demand_diff);
```

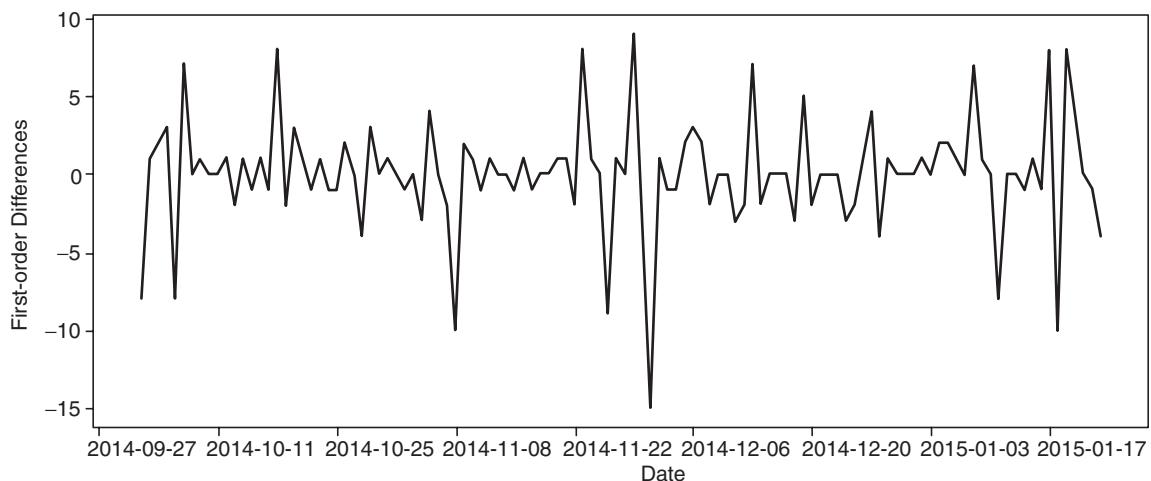


FIGURE 8.9 First-order differences for daily store demand over date.

Figure 8.9 indicates the trend of first-order of differences for demand over dates. Again, it is not very apparent from the trend if the series is stationary or not. We will need to plot the ACF plot to verify (Figure 8.10).

```
pacf_plot = plot_acf(store_df.demand_diff.dropna(),
                      lags=10)
```

The ACF plot in Figure 8.10 shows no sign of slow reduction in autocorrelation over lags. It immediately cuts off to zero. We can build the model with first 100 observations as training set and subsequent observations as test set.

```
store_train = store_df[0:100]
store_test = store_df[100:]
```

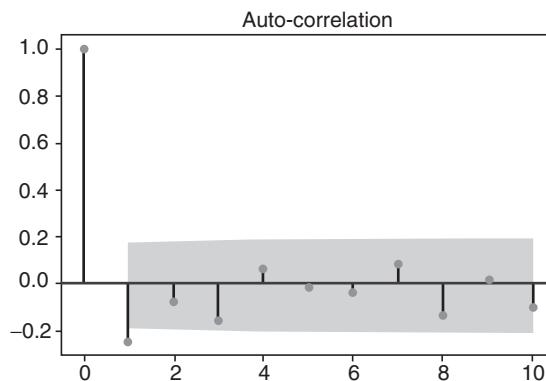


FIGURE 8.10 ACF plot after first-order differencing.

```
arima = ARIMA(store_train.demand.astype(np.float64).as_matrix(),
              order = (1,1,1))
arima_model = arima.fit()
arima_model.summary2()
```

Model:	ARIMA	BIC:	532.1510
Dependent Variable:	D.y	Log-Likelihood:	-256.89
Date:	2019-03-01 13:42	Scale:	1.0000
No. Observations:	99	Method:	css-mle
Df Model:	3	Sample:	1
Df Residuals:	96		0
Converged:	1.0000	S.D. of innovations:	3.237
No. Iterations:	10.0000	HQIC:	525.971
AIC:	521.7706		

	Coef.	Std.Err.	t	P > t	[0.025	0.975]
const	0.0357	0.1599	0.2232	0.8238	-0.2776	0.3490
ar.L1.D.y	0.4058	0.2294	1.7695	0.0800	-0.0437	0.8554
ma.L1.D.y	-0.7155	0.1790	-3.9972	0.0001	-1.0663	-0.3647

	Real	Imaginary	Modulus	Frequency
AR.1	2.4641	0.0000	2.4641	0.0000
MA.1	1.3977	0.0000	1.3977	0.0000

ARIMA model is a regression model and thus has to satisfy all the assumptions of regression. The residuals should be white noise and not correlated. This can be observed by using ACF and PACF plots of the residuals. The model residuals are given by *arima_model.resid* variable.

```
acf_plot = plot_acf(arima_model.resid,  
                     lags = 20)
```

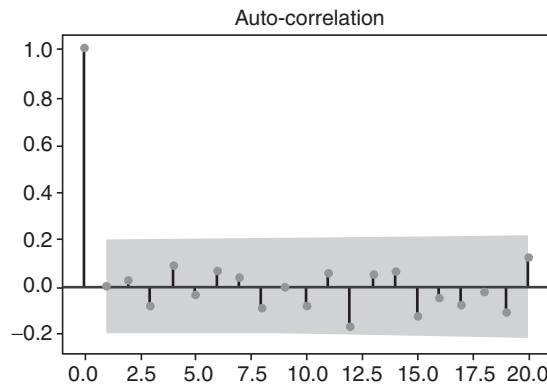


FIGURE 8.11 ACF plot of the residuals.

```
pacf_plot = plot_pacf(arima_model.resid,  
                      lags = 20)
```

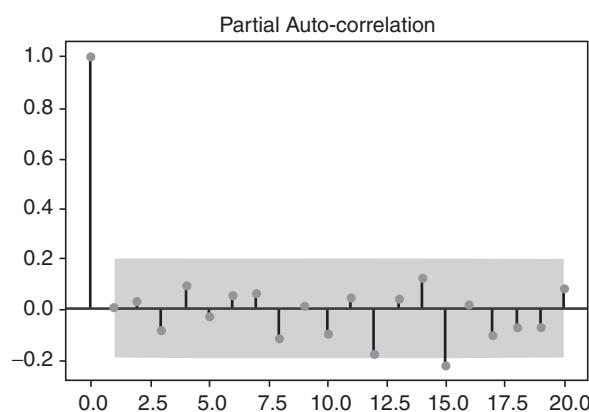


FIGURE 8.12 PACF plot of the residuals.

The plots in Figures 8.11 and 8.12 do not show any auto-correlation of residuals. So, the model can be assumed to be valid.

8.5.4.4 Forecast and Measure Accuracy

The `forecast()` method takes the number of future periods (steps) for predicting values and returns the predicted values along with standard error and confidence interval of the predictions.

```
store_predict, stderr, ci = arima_model.forecast(steps = 15)
```

```
store_predict
```

```
array([[ 17.32364962,    16.2586981,    15.84770837,    15.70211912,
       15.66423863,    15.67007012,    15.69364144,    15.7244122,
       15.75810475,    15.79298307,    15.8283426,     15.86389744,
       15.89953153,    15.93519779,    15.9708771 ]])
```

```
get_mape(store_df.demand[100:],
          store_predict)
```

24.17

The ARIMA model with first-order differencing gives forecast accuracy of 24.17%.

CONCLUSION

1. Forecasting is carried out on a time-series data in which the dependent variable Y_t is observed at different time periods t .
2. Several techniques such as moving average, exponential smoothing, and auto-regressive models are used for forecasting future value of Y_t .
3. The forecasting models are validated using accuracy measures such as RMSE and MAPE.
4. Auto-regressive (AR) models are regression-based models in which dependent variable is Y_t and the independent variables are Y_{t-1} , Y_{t-2} , etc.
5. AR models can be used only when the data is stationary.
6. Moving average (MA) models are regression models in which the independent variables are past error values.
7. Auto-regressive integrated moving average (ARIMA) has three components:
 - a. Auto-regressive component with p lags – AR(p)
 - b. Moving average component with q lags – MA(q)
 - c. Integration which is differencing the original data to make it stationary (denoted by d).
8. One of the necessary conditions of acceptance of ARIMA model is that the residuals should follow white noise.
9. In ARIMA, the model identification, that is, identifying the value of p in AR and q in MA, is achieved through auto-correlation function (ACF) and partial auto-correlation function (PACF).
10. The stationarity of time-series data can be checked using the Dickey–Fuller test.

EXERCISES

Dataset for All the Exercises is given in the File *forecast.xls*.

Answer Questions 1 to 4 using the Data Available in the Tab “Quarterly Demand”.

The dataset contains quarterly demand for a product of a company. The fields are:

year: 2012 to 2014

quarter number: Q1 to Q4

demand: Number of products

1. Plot the trend of demand against the year and quarter.
2. Build a simple moving average model to forecast demand for next quarter.
 - a. Plot the moving average forecast against the actual demand.
 - b. Use forecasted values of all four quarters of 2015 to calculate RMSE and MAPE.
3. Build an exponential moving average model to forecast demand for next quarter.
 - a. Plot the forecasted values against the actual demand.
 - b. Use forecasted values of all four quarters of 2015 to calculate RMSE and MAPE.
4. Build an ARMA model to forecast demand for next quarter.
 - a. Draw ACF and PACF plots to find out p and q values.
 - b. Build an ARMA model using p and q values obtained from ACF and PACF plots.
 - c. Plot the forecasted demand against the actual demand.
 - d. Use forecasted values of all four quarters of 2015 to calculate RMSE and MAPE.

Answer Questions 5 to 8 using the Data Available in the Tab “TRP”.

The dataset contains TRP (Television Rating Points) of a television program over 30 episodes. And the fields are:

episode number: 1 to 30

TRP: Actual TRP number

5. Plot the trend of demand against the year and quarter.
6. Build an exponential moving average model to forecast demand for next quarter.
 - a. Plot the forecasted values against the actual TRP of all episodes.
 - b. Use forecasted values of last six episodes to calculate RMSE and MAPE.
 - c. Find the best smoothing constant, which gives the best MAPE.
7. Decompose the time series data (TRP) and interpret if there is trend and seasonality in the data.
8. Build an ARMA model to forecast TRP.
 - a. Draw ACF and PACF plots to find out p and q values.
 - b. Build an ARMA model using p and q values obtained from ACF and PACF plots.
 - c. Plot the forecasted values against the actual TRP of all episodes.
 - d. Use forecasted values of last six episodes to calculate RMSE and MAPE.
 - e. Find if the data is stationary using Dicky–Fuller Test.
 - (i) If not stationary, use first order differencing to build an ARIMA model.
 - (ii) Use forecasted values of last six episodes to calculate RMSE and MAPE.
 - (iii) Compare the results of ARMA and ARIMA models.

REFERENCES

1. Hill K (2011). *Extreme Engineering: The Boeing 747, Science Based Life – Add Little Reason to Your Day*, 25 July 2011. Available at <https://sciencebasedlife.wordpress.com/2011/07/25/extreme-engineering-the-boeing-747/> (accessed on 10 May 2017).
2. http://corporate.walmart.com/_news/_news-archive/2005/01/07/our-retail-divisions
3. Anon (2018). What's an Average Length of Boom and Bust Cycle, INVESTOPEDIA. Available at <https://www.investopedia.com/ask/answers/071315/what-average-length-boom-and-bust-cycle-us-economy.asp>
4. Box G E P and Jenkins G M (1970). *Time Series Analysis, Forecasting and Control*, Holden Day, San Francisco.
5. Yaffee R A and McGee M (2000). *An Introduction to Time Series Analysis and Forecasting: With Applications of SAS and SPSS*, Academic Press, New York.
6. Dickey D A and Fuller W A (1979). "Distribution of Estimation of Auto-Regressive Time Series with a Unit Root", *Journal of the American Statistical Association*, 74, 427–431.
7. U Dinesh Kumar (2017). *Business Analytics: The Science of Data-Driven Decision Making*, Wiley India, India.



CHAPTER

9

Recommender Systems

LEARNING OBJECTIVES

After completing this chapter, you will be able to

- Understand recommender systems and their business applications.
- Learn about the datasets and algorithm required for building recommendation systems.
- Learn recommender system development techniques such as association rules and collaborative filtering.
- Learn how to build and evaluate recommendation systems using Python libraries.

9.1 | OVERVIEW

Marketing is about connecting the best products or services to the right customers. In today's digital world, personalization is essential for meeting customer's needs more effectively, thereby increasing customer satisfaction and the likelihood of repeat purchases. Recommendation systems are a set of algorithms which recommend most relevant items to users based on their preferences predicted using the algorithms. It acts on behavioral data, such as customer's previous purchase, ratings or reviews to predict their likelihood of buying a new product or service.

Amazon's "Customers who buy this item also bought", Netflix's "shows and movies you may want to watch" are examples of recommendation systems. Recommender systems are very popular for recommending products such as movies, music, news, books, articles, groceries and act as a backbone for cross-selling across industries.

In this chapter, we will be discussing the following three algorithms that are widely used for building recommendation systems:

1. Association Rules
2. Collaborative Filtering
3. Matrix Factorization

9.1.1 | Datasets

For exploring the algorithms specified in the previous section, we will be using the following two publicly available datasets and build recommendations.

- groceries.csv:** This dataset contains transactions of a grocery store and can be downloaded from http://www.sci.csueastbay.edu/~esuess/classes/Statistics_6620/Presentations/ml13/groceries.csv.
- Movie Lens:** This dataset contains 20000263 ratings and 465564 tag applications across 27278 movies. As per the source of data, these data were created by 138493 users between January 09, 1995 and March 31, 2015. This dataset was generated on October 17, 2016. Users were selected and included randomly. All selected users had rated at least 20 movies. The dataset can be downloaded from the link <https://grouplens.org/datasets/movielens/>.

9.2 | ASSOCIATION RULES (ASSOCIATION RULE MINING)

Association rule finds combinations of items that frequently occur together in orders or baskets (in a retail context). The items that frequently occur together are called *itemsets*. Itemsets help to discover relationships between items that people buy together and use that as a basis for creating strategies like combining products as combo offer or place products next to each other in retail shelves to attract customer attention. An application of association rule mining is in Market Basket Analysis (MBA). MBA is a technique used mostly by retailers to find associations between items purchased by customers.

To illustrate the association rule mining concept, let us consider a set of baskets and the items in those baskets purchased by customers as depicted in Figure 9.1.

Items purchased in different baskets are:

- Basket 1: egg, beer, sugar, bread, diaper
- Basket 2: egg, beer, cereal, bread, diaper
- Basket 3: milk, beer, bread
- Basket 4: cereal, diaper, bread

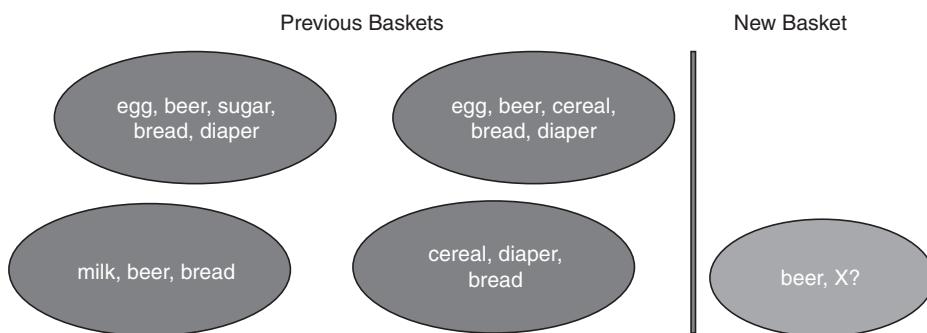


FIGURE 9.1 Baskets and the items in the baskets.

The primary objective of a recommender system is to predict items that a customer may purchase in the future based on his/her purchases so far. In future, if a customer buys beer, can we predict what he/she is most likely to buy along with *beer*? To predict this, we need to find out which items have shown a strong association with *beer* in previously purchased baskets. We can use association rule mining technique to find this out.

Association rule considers all possible combination of items in the previous baskets and computes various measures such as *support*, *confidence*, and *lift* to identify rules with stronger associations. One of the challenges in association rule mining is the number of combination of items that need to be considered; as the number of unique items sold by the seller increases, the number of associations can increase exponentially. And in today's world, retailers sell millions of items. Thus, association rule mining may require huge computational power to go through all possible combinations.

One solution to this problem is to eliminate items that possibly cannot be part of any itemsets. One such algorithm the association rules use *apriori algorithm*. The *apriori* algorithm was proposed by Agrawal and Srikant (1994). The rules generated are represented as

$$\{\text{diapers}\} \rightarrow \{\text{beer}\}$$

which means that customers who purchased diapers also purchased beer in the same basket. $\{\text{diaper}, \text{beer}\}$ together is called itemset. $\{\text{diaper}\}$ is called the antecedent and the $\{\text{beer}\}$ is called the consequent. Both antecedents and consequents can have multiple items, e.g. $\{\text{diaper}, \text{milk}\} \rightarrow \{\text{beer}, \text{bread}\}$ is also a valid rule. Each rule is measured with a set of metrics, which are explained in the next section.

9.2.1 | Metrics

Concepts such as support, confidence, and lift are used to generate association rules. These concepts are explained below.

9.2.1.1 Support

Support indicates the frequencies of items appearing together in baskets with respect to all possible baskets being considered (or in a sample). For example, the support for (beer, diaper) will be 2/4 (based on the data shown in Figure 9.1), that is, 50% as it appears together in 2 baskets out of 4 baskets.

Assume that X and Y are items being considered. Let

1. N be the total number of baskets.
2. N_{XY} represent the number of baskets in which X and Y appear together.
3. N_X represent the number of baskets in which X appears.
4. N_Y represent the number of baskets in which Y appears.

Then the support between X and Y , $\text{Support}(X, Y)$, is given by

$$\text{Support}(X, Y) = \frac{N_{XY}}{N} \quad (9.1)$$

To filter out stronger associations, we can set a minimum support (for example, minimum support of 0.01). This means the itemset must be present in at least 1% of baskets. Apriori algorithm uses minimum support criteria to reduce the number of possible itemset combinations, which in turn reduces computational requirements.

If minimum support is set at 0.01, an association between X and Y will be considered if and only if both X and Y have minimum support of 0.01. Hence, *apriori* algorithm computes support for each item independently and eliminates items with support less than minimum support. The support of each individual item can be calculated using Eq. (9.1).

9.2.1.2 Confidence

Confidence measures the proportion of the transactions that contain X , which also contain Y . X is called antecedent and Y is called consequent. Confidence can be calculated using the following formula:

$$\text{Confidence}(X \rightarrow Y) = P(Y | X) = \frac{N_{XY}}{N_X} \quad (9.2)$$

where $P(Y|X)$ is the conditional probability of Y given X .

9.2.1.3 Lift

Lift is calculated using the following formula:

$$\text{Lift} = \frac{\text{Support}(X, Y)}{\text{Support}(X) \times \text{Support}(Y)} = \frac{N_{XY}}{N_X N_Y} \quad (9.3)$$

Lift can be interpreted as the degree of association between two items. Lift value 1 indicates that the items are independent (no association), lift value of less than 1 implies that the products are substitution (purchase one product will decrease the probability of purchase of the other product) and lift value of greater than 1 indicates purchase of Product X will increase the probability of purchase of Product Y . Lift value of greater than 1 is a necessary condition of generating association rules.

9.2.2 | Applying Association Rules

We will create association rules using the transactions data available in the *groceries.csv* dataset. Each line in the dataset is an order and contains a variable number of items. Each item in each order is separated by a comma in the dataset.

9.2.2.1 Loading the Dataset

Python's *open()* method can be used to open the file and *readlines()* to read each line. The following code block can be used for loading and reading the data:

```
all_txns = []

#open the file
with open('groceries.csv') as f:
    #read each line
    content = f.readlines()
    #Remove white space from the beginning and end of the line
    txns = [x.strip() for x in content]
    # Iterate through each line and create a list of transactions
    for each_txn in txns:
        #Each transaction will contain a list of item in the
        #transaction
        all_txns.append( each_txn.split(',') )
```

The steps in this code block are explained as follows:

1. The code opens the file *groceries.csv*.
2. Reads all the lines from the file.
3. Removes leading or trailing white spaces from each line.
4. Splits each line by a comma to extract items.
5. Stores the items in each line in a list.

In the end, the variable *all_txns* will contain a list of orders and list of items in each order. An order is also called a transaction.

To print the first five transactions, we can use the following code:

```
all_txns[0:5]
```

The output is shown below:

```
[['citrus fruit', 'semi-finished bread', 'margarine', 'ready soups'],
 ['tropical fruit', 'yogurt', 'coffee'],
 ['whole milk'],
 ['pip fruit', 'yogurt', 'cream cheese', 'meat spreads'],
 ['other vegetables',
 'whole milk',
 'condensed milk',
 'long life bakery product']]
```

9.2.2.2 Encoding the Transactions

Python library *mlxtend* provides methods to generate association rules from a list of transactions. But these methods require the data to be fed in specific format. The transactions and items need to be converted into a tabular or matrix format. Each row represents a transaction and each column represents an item. So, the matrix size will be of $M \times N$, where M represents the total number of transactions and N represents all unique items available across all transactions (or the number of items sold by the seller). The items available in each transaction will be represented in one-hot-encoded format, that is, the item is encoded 1 if it exists in the transaction or 0 otherwise. The *mlxtend* library has a feature pre-processing implementation class called *OnehotTransactions* that will take *all_txns* as an input and convert the transactions and items into one-hot-encoded format. The code for converting the transactional data using one-hot encoding is as follows:

```
# Import all required libraries
import pandas as pd
import numpy as np
from mlxtend.preprocessing import OnehotTransactions
from mlxtend.frequent_patterns import apriori, association_rules
```

The print method can be used for printing the first five transactions and items, indexed from 10 to 20. The results are shown in Table 9.1.

```
# Initialize OnehotTransactions
one_hot_encoding = OnehotTransactions()
# Transform the data into one-hot-encoding format
one_hot_txns = one_hot_encoding.fit(all_txns).transform(all_txns)
# Convert the matrix into the dataframe.
one_hot_txns_df = pd.DataFrame(one_hot_txns,
                                columns=one_hot_encoding.columns_)
```

```
one_hot_txns_df.iloc[5:10, 10:20]
```

TABLE 9.1 One-hot-encoded transactions data

Berries	Beverages	Bottled beer	Bottled water	Brandy	Brown bread	Butter	Buttermilk	Cake bar	Candles
5	0	0	0	0	0	1	0	0	0
6	0	0	0	0	0	0	0	0	0
7	0	0	1	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0

It can be noticed from Table 9.1 that transaction with index 5 contains an item called *butter* (item purchased by the customer) and transaction with index 7 contains an item *bottled beer*. All other entries in Table 9.1 are 0, which implies that these items were not purchased. The transactional matrices are likely to be sparse since each customer is likely to buy very few items in comparison to the total number of items sold by the seller. The following code can be used for finding the size (shape or dimension) of the matrix.

```
one_hot_txns_df.shape
```

```
(9835, 171)
```

The sparse matrix has a dimension of 9835×171 . So, a total of 9835 transactions and 171 items are available. This matrix can be fed to generate rules, which will be discussed in the next sub-section.

9.2.2.3 Generating Association Rules

We will use *apriori* algorithms to generate itemset. The total number of itemset will depend on the number of items that exist across all transactions. The number of items in the data can be obtained using the following code:

```
len(one_hot_txns_df.columns)
```

```
171
```

The code gives us an output of 171, that is, as mentioned in the previous section, there are 171 items. For itemset containing 2 items in each set, the total number of itemsets will be $^{171}C_2$, that is, the number of itemset will be 14535. It is a very large number and computationally intensive. To limit the number of

generated rules, we will apply minimum support value. All items that do not have the minimum support will be removed from the possible itemset combinations.

Apriori algorithm takes the following parameters:

1. **df: pandas** – DataFrame in a one-hot-encoded format.
2. **min_support: float** – A float between 0 and 1 for minimum support of the itemsets returned. Default is 0.5.
3. **use_colnames: boolean** – If true, uses the DataFrames' column names in the returned DataFrame instead of column indices.

We will be using a minimum support of 0.02, that is, the itemset is available in at least 2% of all transactions. The following commands can be used for setting minimum support.

```
frequent_itemsets = apriori(one_hot_txns_df,
                             min_support=0.02,
                             use_colnames=True)
```

The following command can be used for printing 10 randomly sampled itemsets and their corresponding support. The itemsets are shown in Table 9.2.

```
frequent_itemsets.sample(10, random_state = 90)
```

TABLE 9.2 Itemsets with a minimum support value of 0.02

	Support	Itemsets
60	0.020437	[bottled beer, whole milk]
52	0.033859	[sugar]
89	0.035892	[other vegetables, tropical fruit]
105	0.021047	[root vegetables, tropical fruit]
88	0.032740	[other vegetables, soda]
16	0.058058	[coffee]
111	0.024504	[shopping bags, whole milk]
36	0.079817	[newspapers]
119	0.056024	[whole milk, yogurt]
55	0.071683	[whipped/sour cream]

The *apriori* algorithm filters out frequent itemsets which have minimum support of greater than 2%. From Table 9.2, we can infer that *whole milk* and *yogurt* appear together in about 5.6% of the baskets. These itemsets can be passed to *association_rules* for generating rules and corresponding metrics. The following commands are used. The corresponding association rules are shown in Table 9.3

1. **df : pandas** – DataFrame of frequent itemsets with columns ['support', 'itemsets'].
2. **metric** – In this use 'confidence' and 'lift' to evaluate if a rule is of interest. Default is 'confidence'.
3. **min_threshold** – Minimal threshold for the evaluation metric to decide whether a candidate rule is of interest.

```
rules = association_rules(frequent_itemsets, # itemsets
                           metric="lift", # lift
                           min_threshold=1)
```

```
rules.sample(5)
```

TABLE 9.3 Association rules

	Antecedants	Consequents	Support	Confidence	Lift
7	(soda)	(rolls/buns)	0.174377	0.219825	1.195124
55	(yogurt)	(bottled water)	0.139502	0.164723	1.490387
74	(soda)	(yogurt)	0.174377	0.156851	1.124368
89	(root vegetables)	(whole milk)	0.108998	0.448694	1.756031
59	(citrus fruit)	(yogurt)	0.082766	0.261671	1.875752

9.2.2.4 Top Ten Rules

Let us look at the top 10 association rules sorted by *confidence*. The rules stored in the variable *rules* are sorted by confidence in descending order. They are printed as seen in Table 9.4.

```
rules.sort_values('confidence',
                  ascending = False)[0:10]
```

TABLE 9.4 Top 10 association rules based on confidence sorted from highest to lowest

	Antecedants	Consequents	Support	Confidence	Lift
42	(yogurt, other vegetables)	(whole milk)	0.043416	0.512881	2.007235
48	(butter)	(whole milk)	0.055414	0.497248	1.946053
120	(curd)	(whole milk)	0.053279	0.490458	1.919481
80	(other vegetables, root vegetables)	(whole milk)	0.047382	0.489270	1.914833
78	(root vegetables, whole milk)	(other vegetables)	0.048907	0.474012	2.449770
27	(domestic eggs)	(whole milk)	0.063447	0.472756	1.850203
0	(whipped/sour cream)	(whole milk)	0.071683	0.449645	1.759754
89	(root vegetables)	(whole milk)	0.108998	0.448694	1.756031
92	(root vegetables)	(other vegetables)	0.108998	0.434701	2.246605
24	(frozen vegetables)	(whole milk)	0.048094	0.424947	1.663094

From Table 9.4, we can infer that the probability that a customer buys (whole milk), given he/she has bought (yogurt, other vegetables), is 0.51. Now, these rules can be used to create strategies like keeping the items together in store shelves or cross-selling.

9.2.2.5 Pros and Cons of Association Rule Mining

The following are advantages of using association rules:

1. Transactions data, which is used for generating rules, is always available and mostly clean.
2. The rules generated are simple and can be interpreted.

However, association rules do not take the preference or ratings given by customers into account, which is an important information pertaining for generating rules. If customers have bought two items but disliked one of them, then the association should not be considered. Collaborative filtering takes both, what customers bought and how they liked (rating) the items, into consideration before recommending.

Association rules mining is used across several use cases including product recommendations, fraud detection from transaction sequences, medical diagnosis, weather prediction, etc.

9.3 | COLLABORATIVE FILTERING

Collaborative filtering is based on the notion of similarity (or distance). For example, if two users A and B have purchased the same products and have rated them similarly on a common rating scale, then A and B can be considered similar in their buying and preference behavior. Hence, if A buys a new product and rates high, then that product can be recommended to B. Alternatively, the products that A has already bought and rated high can be recommended to B, if not already bought by B.

9.3.1 | How to Find Similarity between Users?

Similarity or the distance between users can be computed using the rating the users have given to the common items purchased. If the users are similar, then the similarity measures such as Jaccard coefficient and cosine similarity will have a value closer to 1 and distance measures such as Euclidian distance will have low value. Calculating similarity and distance have already been discussed in Chapter 7. Most widely used distances or similarities are Euclidean distance, Jaccard coefficient, cosine similarity, and Pearson correlation.

We will be discussing collaborative filtering technique using the example described below. The picture in Figure 9.2 depicts three users *Rahul*, *Purvi*, and *Gaurav* and the books they have bought and rated.

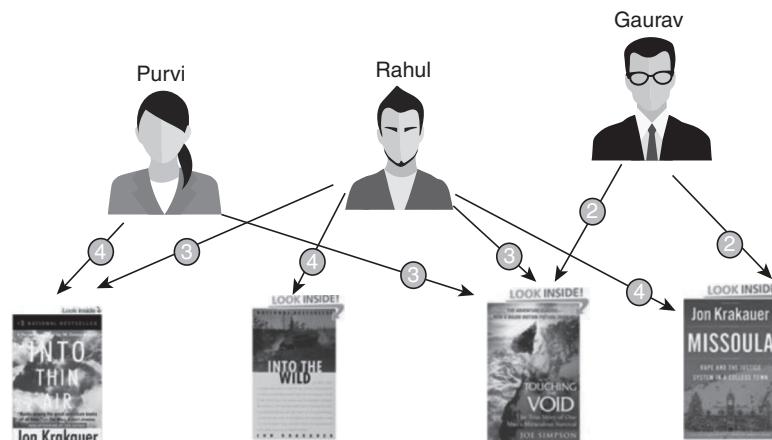


FIGURE 9.2 Users, their purchases and ratings.

The users are represented using their rating on the Euclidean space in Figure 9.3. Here the dimensions are represented by the two books *Into Thin Air* and *Missoula*, which are the two books commonly bought by *Rahul*, *Purvi*, and *Gaurav*.

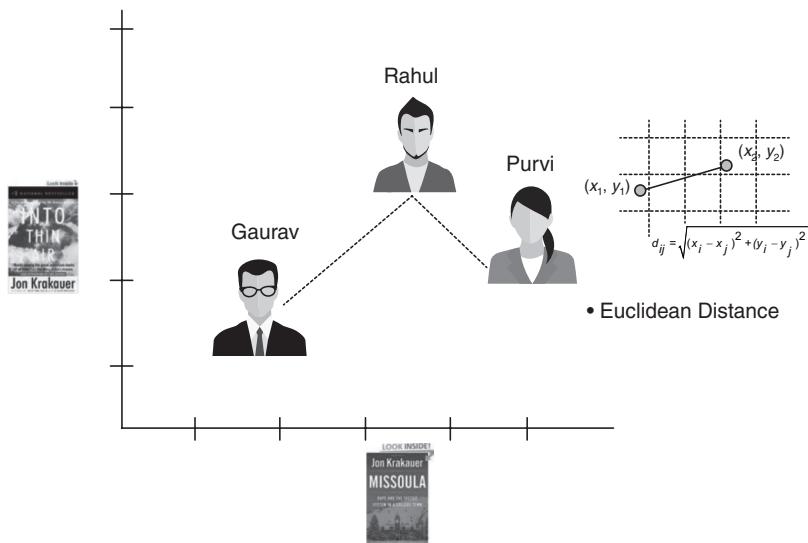


FIGURE 9.3 Euclidean distance based on user's ratings.

Figure 9.3 shows that *Rahul*'s preferences are similar to *Purvi*'s rather than to *Gaurav*'s. So, the other book, *Into the Wild*, which *Rahul* has bought and rated high, can now be recommended to *Purvi*.

Collaborative filtering comes in two variations:

1. **User-Based Similarity:** Finds K similar users based on common items they have bought.
2. **Item-Based Similarity:** Finds K similar items based on common users who have bought those items.

Both algorithms are similar to K -Nearest Neighbors (KNN), which was discussed in Chapter 6.

9.3.2 | User-Based Similarity

We will use *MovieLens* dataset (see <https://grouplens.org/datasets/movielens/>) for finding similar users based on common movies the users have watched and how they have rated those movies. The file *ratings.csv* in the dataset contains ratings given by users. Each line in this file represents a rating given by a user to a movie. The ratings are on the scale of 1 to 5. The dataset has the following features:

1. userId
2. movieId
3. rating
4. timestamp

9.3.2.1 Loading the Dataset

The following loads the file onto a DataFrame using *pandas' read_csv()* method.

```
rating_df = pd.read_csv( "ml-latest-small/ratings.csv" )
```

Let us print the first five records.

```
rating_df.head(5)
```

The output is shown in Table 9.5.

TABLE 9.5 Movie ID and rating

	userId	movieId	rating	timestamp
0	1	31	2.5	1260759144
1	1	1029	3.0	1260759179
2	1	1061	3.0	1260759182
3	1	1129	2.0	1260759185
4	1	1172	4.0	1260759205

The *timestamp* column will not be used in this example, so it can be dropped from the dataframe.

```
rating_df.drop( 'timestamp', axis = 1, inplace = True )
```

The number of unique users in the dataset can be found using method *unique()* on *userId* column.

```
len( rating_df.userId.unique() )
```

671

Similarly, the number of unique movies in the dataset is

```
len( rating_df.movieId.unique() )
```

9066

Before proceeding further, we need to create a pivot table or matrix and represent users as rows and movies as columns. The values of the matrix will be the ratings the users have given to those movies. As there are 671 users and 9066 movies, we will have a matrix of size 671×9066 . The matrix will be very sparse as very few cells will be filled with the ratings using only those movies that users have watched.

Those movies that the users have not watched and rated yet, will be represented as NaN. Pandas DataFrame has pivot method which takes the following three parameters:

- index:** Column value to be used as DataFrame's index. So, it will be *userId* column of *rating_df*.
- columns:** Column values to be used as DataFrame's columns. So, it will be *movieId* column of *rating_df*.
- values:** Column to use for populating DataFrame's values. So, it will be *rating* column of *rating_df*.

```
user_movies_df = rating_df.pivot( index='userId',
                                  columns='movieId',
                                  values = "rating"
                                ).reset_index(drop=True)
user_movies_df.index=rating_df.userId.unique()
```

Let us print the first 5 rows and first 15 columns. The result is shown in Table 9.6.

```
user_movies_df.iloc[0:5, 0:15]
```

TABLE 9.6 First few records

movieId	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	NaN														
2	NaN	4.0	NaN	NaN	NaN	NaN	NaN								
3	NaN														
4	NaN	4.0	NaN	NaN	NaN	NaN	NaN								
5	NaN	NaN	4.0	NaN											

The DataFrame contains NaN for those entries where users have seen a movie and not rated. We can impute those NaNs with 0 values using the following codes. The results are shown in Table 9.7.

```
user_movies_df.fillna( 0, inplace = True
) user_movies_df.iloc[0:5, 0:10]
```

TABLE 9.7 NaN entries replaced with 0.0

movieId	1	2	3	4	5	6	7	8	9	10
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	4.0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	4.0
5	0.0	0.0	4.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

9.3.2.2 Calculating Cosine Similarity between Users

Each row in `user_movies_df` represents a user. If we compute the similarity between rows, it will represent the similarity between those users. `sklearn.metrics.pairwise_distances` can be used to compute distance between all pairs of users. `pairwise_distances()` takes a `metric` parameter for what distance measure to use. We will be using cosine similarity for finding similarity.

Cosine similarity closer to 1 means users are very similar and closer to 0 means users are very dissimilar. The following code can be used for calculating the similarity.

```
from sklearn.metrics import pairwise_distances
from scipy.spatial.distance import cosine, correlation

user_sim = 1 - pairwise_distances( user_movies_df.values,
                                   metric="cosine" )

#Store the results in a dataframe
user_sim_df = pd.DataFrame( user_sim )
#Set the index and column names to user ids (0 to 671)
user_sim_df.index = rating_df.userId.unique()
user_sim_df.columns = rating_df.userId.unique()
```

We can print the similarity between first 5 users by using the following code. The result is shown in Table 9.8.

```
user_sim_df.iloc[0:5, 0:5]
```

TABLE 9.8 Cosine similarity values

	1	2	3	4	5
1	1.000000	0.000000	0.000000	0.074482	0.016818
2	0.000000	1.000000	0.124295	0.118821	0.103646
3	0.000000	0.124295	1.000000	0.081640	0.151531
4	0.074482	0.118821	0.081640	1.000000	0.130649
5	0.016818	0.103646	0.151531	0.130649	1.000000

The total dimension of the matrix is available in the `shape` variable of `user_sim_df` matrix.

```
user_sim_df.shape  
(671, 671)
```

`user_sim_df` matrix shape shows that it contains the cosine similarity between all possible pairs of users. And each cell represents the cosine similarity between two specific users. For example, the similarity between userid 1 and userid 5 is 0.016818.

The diagonal of the matrix shows the similarity of an user with itself (i.e., 1.0). This is true as each user is most similar to himself or herself. But we need the algorithm to find other users who are similar to a specific user. So, we will set the diagonal values as 0.0 . The result is shown in Table 9.9.

```
np.fill_diagonal( user_sim, 0 )
user_sim_df.iloc[0:5, 0:5]
```

TABLE 9.9 Diagonal similarity value set to 0

	1	2	3	4	5
1	0.000000	0.000000	0.000000	0.074482	0.016818
2	0.000000	0.000000	0.124295	0.118821	0.103646
3	0.000000	0.124295	0.000000	0.081640	0.151531
4	0.074482	0.118821	0.081640	0.000000	0.130649
5	0.016818	0.103646	0.151531	0.130649	0.000000

All diagonal values are set to 0, which helps to avoid selecting self as the most similar user.

9.3.2.3 Filtering Similar Users

To find most similar users, the maximum values of each column can be filtered. For example, the most similar user to first 5 users with *userid* 1 to 5 can be obtained using the following code:

```
user_sim_df.idxmax(axis=1) [0:5]
```

```
1    325
2    338
3    379
4    518
5    313
dtype: int64
```

The above result shows user 325 is most similar to user 1, user 338 is most similar to user 2, and so on.

To dive a little deeper to understand the similarity, let us print the similarity values between user 2 and users ranging from 331 to 340.

```
user_sim_df.iloc[1:2, 330:340]
```

Output

331	332	333	334	335	336	337	338	339	340	
2	0.030344	0.002368	0.052731	0.047094	0.0	0.053044	0.05287	0.581528	0.093863	0.081814

The output shows that the cosine similarity between *userid* 2 and *userid* 338 is 0.581528 and highest. But why is user 338 most similar to user 2? This can be explained intuitively if we can verify that the two users have watched several movies in common and rated very similarly. For this, we need to read *movies* dataset, which contains the movie id along with the movie name.

9.3.2.4 Loading the Movies Dataset

Movie information is contained in the file *movies.csv*. Each line of this file contains the *movieid*, the movie name, and the movie genre.

Movie titles are entered manually or imported from <https://www.themoviedb.org/> and include the year of release in parentheses. Errors and inconsistencies may exist in these titles. The movie can be loaded using the following codes:

```
movies_df = pd.read_csv( "ml-latest-small/movies.csv" )
```

We will print the first 5 movie details using the following code. The result is shown in Table 9.10.

```
movies_df[0:5]
```

TABLE 9.10 Movie details

	movieid	title	genres
0	1	Toy Story (1995)	Adventure Animation Children Comedy Fantasy
1	2	Jumanji (1995)	Adventure Children Fantasy
2	3	Grumpier Old Men (1995)	Comedy Romance
3	4	Waiting to Exhale (1995)	Comedy Drama Romance
4	5	Father of the Bride Part II (1995)	Comedy

The *genres* column is dropped from the DataFrame, as it is not going to be used in this analysis.

```
movies_df.drop( 'genres', axis = 1, inplace = True )
```

9.3.2.5 Finding Common Movies of Similar Users

The following method takes *userids* of two users and returns the common movies they have watched and their ratings.

```
def get_user_similar_movies( user1, user2 ):
    # Inner join between movies watched between two users will give
    # the common movies watched.
    common_movies = rating_df[rating_df.userId == user1].merge(
        rating_df[rating_df.userId == user2],
        on = "movieId",
        how = "inner" )
```

```
# join the above result set with movies details
return common_movies.merge( movies_df, on = 'movieId' )
```

To find out the movies, user 2 and user 338 have watched in common and how they have rated each one of them, we will filter out movies that both have rated at least 4 to limit the number of movies to print. The movies are shown in Table 9.11.

```
common_movies = get_user_similar_movies( 2, 338 )
```

```
common_movies[ (common_movies.rating_x >= 4.0) &
              ((common_movies.rating_y >= 4.0)) ]
```

TABLE 9.11 Common movies between user 2 and user 338 with least rating of 4.0

	userId_x	movieId	rating_x	userId_y	rating_y	title
0	2	17	5.0	338	4.0	Sense and Sensibility (1995)
2	2	47	4.0	338	4.0	Seven (a.k.a. Se7en) (1995)
5	2	150	5.0	338	4.0	Apollo 13 (1995)
28	2	508	4.0	338	4.0	Philadelphia (1993)
29	2	509	4.0	338	4.0	Piano, The (1993)
31	2	527	4.0	338	5.0	Schindler's List (1993)
34	2	589	5.0	338	5.0	Terminator 2: Judgment Day (1991)

From the table we can see that users 2 and 338 have watched 7 movies in common and have rated almost on the same scale. Their preferences seem to be very similar.

How about users with dissimilar behavior? Let us check users 2 and 332, whose cosine similarity is 0.002368.

```
common_movies = get_user_similar_movies( 2, 332 )
common_movies
```

	userId_x	movieId	rating_x	userId_y	rating_y	title
0	2	552	3.0	332	0.5	Three Musketeers, The (1993)

Users 2 and 332 have only one movie in common and have rated very differently. They indeed are very dissimilar.

9.3.2.6 Challenges with User-Based Similarity

Finding user similarity does not work for new users. We need to wait until the new user buys a few items and rates them. Only then users with similar preferences can be found and recommendations can be made based on that. This is called *cold start* problem in recommender systems. This can be overcome by using item-based similarity. *Item-based similarity* is based on the notion that if two items have been bought by

many users and rated similarly, then there must be some inherent relationship between these two items. In other terms, in future, if a user buys one of those two items, he or she will most likely buy the other one.

9.3.3 | Item-Based Similarity

If two movies, movie A and movie B, have been watched by several users and rated very similarly, then movie A and movie B can be similar in taste. In other words, if a user watches movie A, then he or she is very likely to watch B and vice versa.

9.3.3.1 Calculating Cosine Similarity between Movies

In this approach, we need to create a pivot table, where the rows represent movies, columns represent users, and the cells in the matrix represent ratings the users have given to the movies. So, the *pivot()* method will be called with *movieId* as *index* and *userId* as *columns* as described below:

```
rating_mat = rating_df.pivot(index='movieId',
                               columns='userId',
                               values = "rating").reset_index(drop = True)
# Fill all NaNs with 0
rating_mat.fillna(0, inplace = True)
# Find the correlation between movies
movie_sim = 1 - pairwise_distances(rating_mat.values,
                                    metric="correlation")
# Fill the diagonal with 0, as it represents the auto-correlation
# of movies
movie_sim_df = pd.DataFrame( movie_sim )
```

Now, the following code is used to print similarity between the first 5 movies. The results are shown in Table 9.12.

```
movie_sim_df.iloc[0:5, 0:5]
```

TABLE 9.12 Movie similarity

	0	1	2	3	4
0	1.000000	0.223742	0.183266	0.071055	0.105076
1	0.223742	1.000000	0.123790	0.125014	0.193144
2	0.183266	0.123790	1.000000	0.147771	0.317911
3	0.071055	0.125014	0.147771	1.000000	0.150562
4	0.105076	0.193144	0.317911	0.150562	1.000000

The shape of the above similarity matrix is

```
movie_sim_df.shape
```

```
(9066, 9066)
```

There are 9066 movies and the dimension of the matrix (9066, 9066) shows that the similarity is calculated for all pairs of 9066 movies.

9.3.3.2 Finding Most Similar Movies

In the following code, we write a method `get_similar_movies()` which takes a `movieid` as a parameter and returns the similar movies based on cosine similarity. Note that movieid and index of the movie record in the `movies_df` are not same. We need to find the index of the movie record from the movieid and use that to find similarities in the `movie_sim_df`. It takes another parameter `topN` to specify how many similar movies will be returned.

```
def get_similar_movies( movieid, topN = 5 ):
    # Get the index of the movie record in movies_df
    movieidx = movies_df[movies_df.movieId == movieid].index[0]
    movies_df['similarity'] = movie_sim_df.iloc[movieidx]
    top_n = movies_df.sort_values( ["similarity"], ascending =
                                    False )[0:topN]
    return top_n
```

The above method `get_similar_movies()` takes movie id as an argument and returns other movies which are similar to it. Let us find out how the similarities play out by finding out movies which are similar to the movie *Godfather*. And if it makes sense at all! The movie id for the movie *Godfather* is 858.

```
movies_df[movies_df.movieId == 858]
```

movieid	title	similarity
695	858 Godfather, The (1972)	1.0

```
get_similar_movies(858)
```

Table 9.13 shows the movies that are similar to *Godfather*.

TABLE 9.13 Movies similar to *Godfather*

movieid	title	similarity
695	858 Godfather, The (1972)	1.000000
977	1221 Godfather: Part II, The (1974)	0.709246
969	1213 Goodfellas (1990)	0.509372
951	1193 One Flew Over the Cuckoo's Nest (1975)	0.430101
1744	2194 Untouchables, The (1987)	0.418966

It can be observed from Table 9.13 that users who watched '*Godfather, The*', also watched '*Godfather: Part II*' the most. This makes absolute sense! It also indicates that the users have watched *Goodfellas* (1990), *One Flew Over the Cuckoo's Nest* (1975), and *Untouchables, The* (1987).

So, in future, if any user watches '*Godfather, The*', the other movies can be recommended to them. Let us find out which movies are similar to the movie Dumb and Dumber.

```
movies_df[movies_df.movieId == 231]
```

	movieId	title	similarity
203	231	Dumb & Dumber (Dumb and Dumber) (1994)	0.054116

```
get_similar_movies(231)
```

Table 9.14 shows the movies that are similar to Dumb and Dumber

TABLE 9.14 Movies similar to Dumb and Dumber

	movieId	title	similarity
203	231	Dumb & Dumber (Dumb and Dumber) (1994)	1.000000
309	344	Ace Ventura: Pet Detective (1994)	0.635735
18	19	Ace Ventura: When Nature Calls (1995)	0.509839
447	500	Mrs. Doubtfire (1993)	0.485764
331	367	Mask, The (1994)	0.461103

We can see from the table, most of the movies are of *comedy* genre and belong to the actor *Jim Carrey*.

9.4 | USING SURPRISE LIBRARY

For real-world implementations, we need a more extensive library which hides all the implementation details and provides abstract Application Programming Interfaces (APIs) to build recommender systems. *Surprise* is a Python library for accomplishing this. It provides the following features:

1. Various ready-to-use prediction algorithms like neighborhood methods (user similarity and item similarity), and matrix factorization-based. It also has built-in similarity measures such as cosine, mean square distance (MSD), Pearson correlation coefficient, etc.
2. Tools to evaluate, analyze, and compare the performance of the algorithms. It also provides methods to recommend.

We import the required modules or classes from surprise library. All modules or classes and their purpose are discussed in the subsequent sections.

```
from surprise import Dataset, Reader, KNNBasic, evaluate, accuracy
```

The *surprise.Dataset* is used to load the datasets and has a method *load_from_df* to convert DataFrames to Dataset. *Reader* class can be used to provide the range of rating scales that is being used.

```
reader = Reader(rating_scale=(1, 5))
data = Dataset.load_from_df(rating_df[['userId',
                                       'movieId',
                                       'rating']], reader=reader)
```

9.4.1 | User-Based Similarity Algorithm

The `surprise.prediction_algorithms.knns.KNNBasic` provides the collaborative filtering algorithm and takes the following parameters:

1. **K**: The (max) number of neighbors to take into account for aggregation.
2. **min_k**: The minimum number of neighbors to take into account for aggregation, if there are not enough neighbors.
3. **sim_options - (dict)**: A dictionary of options for the similarity measure.
 - (a) **name**: Name of the similarity to be used, e.g., `cosine`, `msd` or `pearson`.
 - (b) **user_based**: True for user-based similarity and False for item-based similarity.

The following code implements movies recommendation based on *Pearson* correlation and 20 nearest similar users.

```
## Set coefficient and similarity parameters for building model
item_based_cosine_sim = {'name': 'pearson',
                           'user_based': True}

knn = KNNBasic(k= 20,
                min_k = 5,
                sim_options = item_based_cosine_sim)
```

The `surprise.model_selection` provides `cross_validate` method to split the dataset into multiple folds, runs the algorithm, and reports the accuracy measures. It takes the following parameters:

1. **algo**: The algorithm to evaluate.
2. **data (Dataset)**: The dataset on which to evaluate the algorithm.
3. **Measures**: The performance measures to compute. Allowed names are function names as defined in the accuracy module. Default is `['rmse', 'mae']`.
4. **cv**: The number of folds for K-Fold cross validation strategy.

We can do 5-fold cross-validation to measure *RMSE* score to find out how the algorithm performs on the dataset.

```
from surprise.model_selection import cross_validate

cv_results = cross_validate(knn,
                            data,
                            measures=[ 'RMSE' ],
                            cv=5,
                            verbose=False)
```

It reports test accuracy for each fold along with the time it takes to build and test the models. Let us take the average accuracy across all the folds.

```
np.mean(cv_results.get('test_rmse'))
```

```
0.9909387452695102
```

This model can predict with root mean square error (RMSE) of 0.99. But can we do better?

9.4.2 | Finding the Best Model

Similar to finding the most optimal hyper-parameters using *GridSearchCV* in *sklearn* as discussed in Chapter 6: Advanced Machine Learning, *Surprise* provides *GridSearchCV* to search through various models and similarity indexes to find the model that gives the highest accuracy.

The *surprise.model_selection.search.GridSearchCV* takes the following parameters:

1. **algo_class:** The class of the algorithm to evaluate (e.g., *KNNBasic*).
2. **param_grid:** Dictionary with hyper parameters as keys and list of corresponding possible values that will be used to search optimal parameter values. All combinations will be evaluated with desired algorithm.
3. **measures:** The performance measures to compute (i.e., ['rmse', 'mae']).
4. **cv:** The number of folds for K-Fold cross validation strategy.
5. **refit:** If True, refit the algorithm on the whole dataset using the set of parameters that gave the best average performance. If False, it does not refit on the whole dataset.

The *GridSearchCV* method returns the best model and its parameters. The possible values for the parameters that will be searched to find the most optimal parameters are as below:

1. Number of neighbors [10, 20].
2. Similarity indexes ['cosine', 'pearson'] .
3. User-based or item-based similarity.

```
from surprise.model_selection.search import GridSearchCV
```

```
param_grid = {'k': [10, 20],
              'sim_options': {'name': ['cosine', 'pearson'],
                             'user_based': [True, False]}
              }

grid_cv = GridSearchCV(KNNBasic,
                       param_grid,
                       measures=['rmse'],
                       cv=5,
                       refit=True)

grid_cv.fit(data)
```

Let us print the score and parameters of the best model.

```
# Best RMSE score
print(grid_cv.best_score['rmse'])

# Combination of parameters that gave the best RMSE score
print(grid_cv.best_params['rmse'])

0.9963783863084851
{'sim_options': {'name': 'cosine', 'user_based': True}, 'k': 20}
```

The best model is user-based collaborative filtering with cosine similarity and 20 similar users. Details of the grid search are captured in the variable *cv_results*. We can convert it to a DataFrame and print a few columns like *param_sim_options* and *mean_test_rmse*.

```
results_df = pd.DataFrame.from_dict(grid_cv.cv_results)
results_df[['param_k', 'param_sim_options', 'mean_test_rmse',
           'rank_test_rmse']]
```

TABLE 9.15 Best model

	param_k	param_sim_options	mean_test_rmse	rank_test_rmse
0	10	{'name': 'cosine', 'user_based': True}	1.009724	4
1	20	{'name': 'cosine', 'user_based': True}	0.996378	1
2	10	{'name': 'cosine', 'user_based': False}	1.048802	8
3	20	{'name': 'cosine', 'user_based': False}	1.015225	6
4	10	{'name': 'pearson', 'user_based': True}	1.012283	5
5	20	{'name': 'pearson', 'user_based': True}	1.000766	2
6	10	{'name': 'pearson', 'user_based': False}	1.030900	7
7	20	{'name': 'pearson', 'user_based': False}	1.004205	3

The detailed output of grid search is shown in Table 9.15. Each record represents the parameters used to build the model and the corresponding RMSE of the model. The last column *rank_test_rmse* shows the rank of the model as per the RMSE on test data (*mean_test_rmse*) among all the models.

9.4.3 | Making Predictions

To make predictions, the weighted ratings are calculated using the ratings of K nearest users, and the movies with highest weighted ratings are recommended. For example, if r_i is the rating of nearest user (or neighbor) u_i and there are K nearest neighbors, s_i is the similarity between the user and the neighbor, then predicted rating for the movie would be

$$r = \frac{\sum_{i=1}^K r_i s_i}{\sum_{i=1}^K r_i} \quad (9.4)$$

All models in *surprise* library provide a method called *predict()*, which takes user id and movie id and predicts its ratings. It returns a *Prediction* class, which has a variable *est* (stands for estimated value) that gives the predicted rating.

```
grid_cv.predict( 1, 2 )

Prediction(uid=1, iid=2, r_ui=None, est=2.5532967054839784,
details= {'was_impossible': False, 'actual_k': 20})
```

9.5 | MATRIX FACTORIZATION

Matrix factorization is a matrix decomposition technique. Matrix decomposition is an approach for reducing a matrix into its constituent parts. Matrix factorization algorithms decompose the user-item matrix into the product of two lower dimensional rectangular matrices.

In Figure 9.4, the original matrix contains users as rows, movies as columns, and rating as values. The matrix can be decomposed into two lower dimensional rectangular matrices.

		Users–Movies Rating Matrix				
		Movies				
		M1	M2	M3	M4	M5
Users	U1	3	4	2	5	1
	U2	2	4	1	2	4
	U3	3	3	5	2	2

		Users–Factors Matrix			Factors–Movies Matrix		
		Factors			Movies		
		F1	F2	F3	M1	M2	M3
Users	U1	0.73	3.22	0	1.47	1	2.73
	U2	0	1.57	2.53	0.6	1.01	1.73
	U3	1.62	0	1.44	0.42	0.95	0.31

FIGURE 9.4 Matrix factorization.

The Users–Movies matrix contains the ratings of 3 users (U1, U2, U3) for 5 movies (M1 through M5). This Users–Movies matrix is factorized into a (3, 3) Users–Factors matrix and (3, 5) Factors–Movies matrix. Multiplying the Users–Factors and Factors–Movies matrix will result in the original Users–Movies matrix.

The idea behind matrix factorization is that there are latent factors that determine why a user rates a movie, and the way he/she rates. The factors could be the story or actors or any other specific attributes of the movies. But we may never know what these factors actually represent. That is why they are called latent factors. A matrix with size (n, m) , where n is the number of users and m is the number of movies, can be factorized into (n, k) and (k, m) matrices, where k is the number of factors.

The Users–Factors matrix represents that there are three factors and how each user has preferences towards these factors. Factors–Movies matrix represents the attributes the movies possess.

In the above example, U1 has the highest preference for factor F2, whereas U2 has the highest preference for factor F3. Similarly, the F2 factor is high in movies M2 and M4. Probably this is the reason why U1 has given high ratings to movies M2 (4) and M4 (5).

One of the popular techniques for matrix factorization is Singular Vector Decomposition (SVD). *Surprise* library provides SVD algorithm, which takes the number of factors (*n_factors*) as a parameter. We will use 5 latent factors for our example.

```
from surprise import SVD
# Use 10 factors for building the model
svd = SVD( n_factors = 5 )
```

Let us use five-fold cross-validation for testing model's performance.

```
cv_results = cross_validate(svd,
                            data,
                            measures=[ 'RMSE' ],
                            cv=5,
                            verbose=True)
```

Evaluating RMSE of algorithm SVD on 5 split(s).

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Mean	Std
RMSE (testset)	0.8904	0.8917	0.8983	0.8799	0.8926	0.8906	0.0060
Fit time	1.91	1.99	1.96	1.87	1.84	1.91	0.06
Test time	0.21	0.18	0.18	0.17	0.18	0.18	0.01

Mean RMSE for SVD is about 0.8906, which is better than earlier algorithms.

CONCLUSION

1. In this chapter, we learnt how customer's purchase history and feedback information can be used to understand their buying behavior and make personalized recommendations.
2. Different recommendation techniques that are used in real world are association rules, collaborative filtering, and matrix factorization.
3. In association rules, metrics such as support, confidence, and lift are used to validate and filter out good recommendation rules for deployment.
4. Collaborative filtering based recommendations are based on user-based and item-based similarity indexes, which use distance metrics such as Euclidean, Cosine, or Pearson correlations.
5. Python library *surprise* can be used to build recommendation systems. It supports grid search mechanism to search for optimal modes from a set of possible model parameters.

EXERCISES

Answer Questions 1 to 5 using the Dataset *Online Retail.xlsx*.

The dataset *Online Retail.xlsx* and the description of the data is taken from <https://archive.ics.uci.edu/ml/datasets/online+retail>

Online Retail.xlsx contains records of transactions that occurred between 01/12/2010 and 09/12/2011 for a UK-based and registered non-store online retail. The company mainly sells unique all-occasion gifts. Many customers of the company are wholesalers. The attributes in the dataset are:

- *InvoiceNo* – Invoice number. Nominal, a 6-digit integral number uniquely assigned to each transaction. If this code starts with letter 'c', it indicates a cancellation.

- *StockCode* – Product (item) code. Nominal, a 5-digit integral number uniquely assigned to each distinct product.
 - *Description* – Product (item) name. Nominal.
 - *Quantity* – The quantities of each product (item) per transaction. Numeric.
 - *InvoiceDate* – Invoice Date and time. Numeric, the day and time when each transaction was generated.
 - *UnitPrice* – Unit price. Numeric, Product price per unit in sterling.
 - *CustomerID* – Customer number. Nominal, a 5-digit integral number uniquely assigned to each customer.
 - *Country* – Country name. Nominal, the name of the country where each customer resides.
1. Select only the transactions that have occurred during the period 09/01/11 and 11/30/11 and create a new subset of data. For answering Questions 2 to 5, use this subset of data.
 2. Transform the above dataset in Question 1 into another dataset where each record represents an invoice. The record will have an *InvoiceNo* as column and *StockCodes* of the corresponding items bought in the invoice. The number of *StockCodes* in an invoice can be variable.
 3. Prepare the data and generate the association rules from the above dataset in Question 2. Filter out all the rules by minimum support of 0.01 and lift of more than 1.0.
 4. Find the top 10 rules from the above association rule set, sorted by confidence in descending order.
 5. Repeat the complete exercise from Question 1 to Question 4 for all invoices of country France only for the period between 01/12/2010 and 09/12/2011. Find the top 10 rules from the above association rule set, sorted by confidence in descending order.

Answer Questions 6 to 8 using the datasets *ratings.csv* and *movies.csv*, which are already used in the chapter.

6. Filter out the ratings records for the movies that belong to only “Action” genre. Calculate item similarity indexes between the movies based on movies the users have bought in “Action” genre and how they have rated them. Use Pearson correlation coefficient to find similarities. Then recommend top 5 similar movies to the following movies:
 - a. Heat
 - b. Eraser
7. Filter out the ratings records for the movies that belong to either “Animation” or “Children” genre. And Calculate item similarity indexes between the movies based on movies the users have bought in either “Animation” or “Children” genre and how they have rated them. Use cosine similarity index to find similarities. Then recommend top 5 similar movies to the following movies:
 - a. Lion King
 - b. The Incredibles
8. Use *surprise* library and grid search mechanism to find the best model to make recommendations for movies in “Action” genre only. Use the following possible parameters to search for best model:
 - a. Number of neighbors [5, 10, 20].
 - b. Similarity indexes ['Cosine', 'Pearson', 'Euclidean'].
 - c. User-based or item-based similarity.

REFERENCES

1. Agrawal R and Srikant R (1994). *Fast Algorithms for Mining Association Rules in Large Databases*, Proceeding VLDB'94, Proceedings of the 20th International Conference on Very Large Databases, pp. 487–499, September 12–15, 1994, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. (See <https://dl.acm.org/citation.cfm?id=672836>)
2. *Groceries* dataset from http://www.sci.csueastbay.edu/~esuess/classes/Statistics_6620/Presentations/ml13/groceries.csv
3. *Movielens* dataset from <https://grouplens.org/datasets/movielens/>
4. *Movies* information <https://www.themoviedb.org/>

CHAPTER 10

Text Analytics

LEARNING OBJECTIVES

After completing this chapter, you will be able to

- Understand the challenges associated with the handling of text data.
- Learn various pre-processing steps to prepare text data for modelling.
- Learn Naïve–Bayes classification algorithm.
- Learn to develop model for sentiment classification.

10.1 | OVERVIEW

In today's world, one of the biggest sources of information is text data, which is unstructured in nature. Finding customer sentiments from product reviews or feedbacks, extracting opinions from social media data are a few examples of text analytics. Finding insights from text data is not as straight forward as structured data and it needs extensive data pre-processing. All the techniques that we have learnt so far require data to be available in a structured format (matrix form). Algorithms that we have explored so far, such as regression, classification, or clustering, can be applied to text data only when the data is cleaned and prepared. For example, predicting stock price movements from news articles is an example of regression in which the features are positive and negative sentiments about a company. Classifying customer sentiment from his or her review comments as positive and negative is an example of classification using text data.

In this chapter, we will use a dataset that is available at <https://www.kaggle.com/c/si650winter11/> data (the original data was contributed by the University of Michigan) for building a classification model to classify sentiment. The data consists of sentiments expressed by users on various movies. Here each comment is a record, which is either classified as positive or negative.

10.2 | SENTIMENT CLASSIFICATION

In the dataset described in the previous paragraph, *sentiment_train* dataset contains review comments on several movies. Comments in the dataset are already labeled as either positive or negative. The dataset contains the following two fields separated by a *tab* character:

1. **text:** Actual review comment on the movie.
2. **sentiment:** Positive sentiments are labelled as 1 and negative sentiments are labelled as 0.

10.2.1 | Loading the Dataset

Loading the data using pandas' `read_csv()` method is done as follows:

```
import pandas as pd
import numpy as np

import warnings
warnings.filterwarnings('ignore')

train_ds = pd.read_csv("sentiment_train", delimiter="\t")
train_ds.head(5)
```

First five records of loaded data are shown in Table 10.1.

TABLE 10.1 First five records of loaded data

Sentiment	Text
0	1 The Da Vinci Code book is just awesome.
1	1 this was the first clive cussler i've ever read, but even books like Relic, and Da Vinci code were more plausible than this.
2	1 i liked the Da Vinci Code a lot.
3	1 i liked the Da Vinci Code a lot.
4	1 I liked the Da Vinci Code but it ultimataly didn't seem to hold it's own.

In Table 10.1, few of the texts may have been truncated while printing as the default column width is limited. This can be changed by setting `max_colwidth` parameter to increase the width size.

Each record or example in the column `text` is called a document. Use the following code to print the first five positive sentiment documents.

```
pd.set_option('max_colwidth', 800)
train_ds[train_ds.sentiment == 1][0:5]
```

Table 10.2 summarizes the first five positive sentiments. Sentiment value of 1 denotes positive sentiment.

TABLE 10.2 First five positive sentiments

Sentiment	Text
0	1 The Da Vinci Code book is just awesome.
1	1 this was the first clive cussler i've ever read, but even books like Relic, and Da Vinci code were more plausible than this.
2	1 i liked the Da Vinci Code a lot.
3	1 i liked the Da Vinci Code a lot.
4	1 I liked the Da Vinci Code but it ultimataly didn't seem to hold it's own.

To print first five negative sentiment documents use

```
train_ds[train_ds.sentiment == 0][0:5]
```

Table 10.3 summarizes the first five negative sentiments. Sentiment value of 0 denotes negative sentiment.

TABLE 10.3 List of negative comments

Sentiment	Text
3943	0 da vinci code was a terrible movie.
3944	0 Then again, the Da Vinci code is super shitty movie, and it made like 700 million.
3945	0 The Da Vinci Code comes out tomorrow, which sucks.
3946	0 i thought the da vinci code movie was really boring.
3947	0 God, Yahoo Games has this truly-awful looking Da Vinci Code-themed skin on its chessboard right now.

In the next section, we will be discussing exploratory data analysis on text data.

10.2.2 | Exploring the Dataset

Exploratory data analysis can be carried out by counting the number of comments, positive comments, negative comments, etc. For example, we can check how many reviews are available in the dataset? Are the positive and negative sentiment reviews well represented in the dataset? Printing metadata of the DataFrame using *info()* method.

```
train_ds.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6918 entries, 0 to 6917
Data columns (total 2 columns):
sentiment    6918 non-null int64
text         6918 non-null object
dtypes: int64(1), object(1)
memory usage: 108.2+ KB
```

From the output we can infer that there are 6918 records available in the dataset. We create a count plot (Figure 10.1) to compare the number of positive and negative sentiments.

```
import matplotlib.pyplot as plt
import seaborn as sn
%matplotlib inline

plt.figure(figsize=(6, 5))
# Create count plot
```

```

ax = sn.countplot(x='sentiment', data=train_ds)
# Annotate
for p in ax.patches:
    ax.annotate(p.get_height(), (p.get_x()+0.1,
    p.get_height()+50))

```

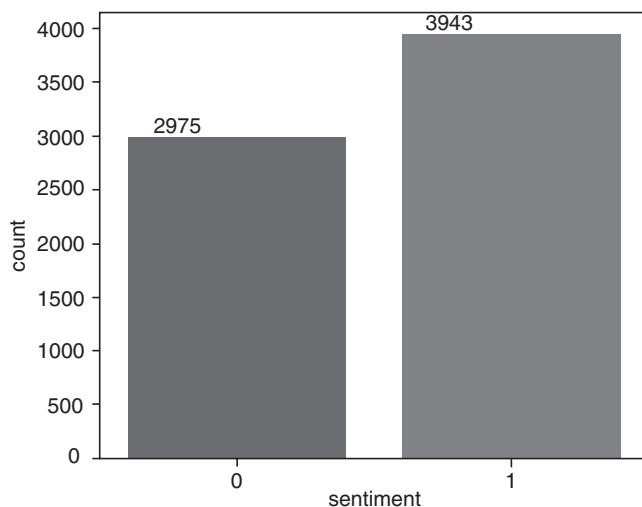


FIGURE 10.1 Number of records of positive and negative sentiments.

From Figure 10.1, we can infer that there are total 6918 records (feedback on movies) in the dataset. Out of 6918 records, 2975 records belong to negative sentiments, while 3943 records belong to positive sentiments. Thus, positive and negative sentiment documents have fairly equal representation in the dataset.

Before building the model, text data needs pre-processing for feature extraction. The following section explains step by step text pre-processing techniques.

10.2.3 | Text Pre-processing

Unlike structured data, features (independent variables) are not explicitly available in text data. Thus, we need to use a process to extract features from the text data. One way is to consider each word as a feature and find a measure to capture whether a word exists or does not exist in a sentence. This is called the bag-of-words (BoW) model. That is, each sentence (comment on a movie or a product) is treated as a bag of words. Each sentence (record) is called a document and collection of all documents is called corpus.

10.2.3.1 *Bag-of-Words (BoW) Model*

The first step in creating a BoW model is to create a dictionary of all the words used in the corpus. At this stage, we will not worry about grammar and only occurrence of the word is captured. Then we will

convert each document to a vector that represents words available in the document. There are three ways to identify the importance of words in a BoW model:

1. Count Vector Model
2. Term Frequency Vector Model
3. Term Frequency-Inverse Document Frequency (TF-IDF) Model

We will discuss these vector models in the following subsections.

Count Vector Model

Consider the following two documents:

1. **Document 1 (positive sentiment):** I really really like IPL.
2. **Document 2 (negative sentiment):** I never like IPL.

Note: IPL stands for Indian Premier League.

The complete vocabulary set (aka dictionary) for the above two documents will have words such as I, really, never, like, IPL. These words can be considered as features (**x1 through x5**). For creating count vectors, we count the occurrence of each word in the document as shown in Table 10.4. The **y**-column in Table 10.4 indicates the sentiment of the statement: 1 for positive and 0 for negative sentiment.

TABLE 10.4 Count vector for document 1 and document 2

Documents	x1	x2	x3	x4	x5	y
	I	really	never	like	ipl	
I really really like ipl	1	2	0	1	1	1
I never like ipl	1	0	1	1	1	0

Term Frequency Vector Model

Term frequency (TF) vector is calculated for each document in the corpus and is the frequency of each term in the document. It is given by,

$$\text{Term Frequency } (TF_i) = \frac{\text{Number of occurrences of word } i \text{ in the document}}{\text{Total number of words in the document}} \quad (10.1)$$

where TF_i is the term frequency for word (aka token). TF representation for the two documents is shown in Table 10.5.

TABLE 10.5 TF vector

	x1	x2	x3	x4	x5	y
	I	really	never	like	ipl	
I really really like ipl	0.2	0.4	0	0.2	0.2	1
I never like ipl	0.25	0	0.25	0.25	0.25	0

Term Frequency-Inverse Document Frequency (TF-IDF)

TF-IDF measures how important a word is to a document in the corpus. The importance of a word (or token) increases proportionally to the number of times a word appears in the document but is reduced by the frequency of the word present in the corpus. TF-IDF for a word i in the document is given by

$$TF-IDF_i = TF_i \times \ln\left(1 + \frac{N}{N_i}\right) \quad (10.2)$$

where N is the total number of documents in the corpus, N_i is the number of documents that contain word i .

The IDF value for each word for the above two documents is given in Table 10.6.

TABLE 10.6 IDF values

	x1	x2	x3	x4	x5
IDF Values	0.693	1.098	1.098	0.693	0.693

The TF-IDF values for the two documents are shown in Table 10.7

TABLE 10.7 TF-IDF values

	x1	x2	x3	x4	x5	y
	1	really	never	like	ipl	
I really really like ipl	0.1386	0.4394	0.0	0.1386	0.1386	1
I never like ipl	0.1732	0.0	0.2746	0.1732	0.1732	0

10.2.3.2 Creating Count Vectors for sentiment_train Dataset

Each document in the dataset needs to be transformed into TF or TF-IDF vectors. *sklearn.feature_extraction.text* module provides classes for creating both TF and TF-IDF vectors from text data. We will use *CountVectorizer* to create count vectors. In CountVectorizer, the documents will be represented by the number of times each word appears in the document.

We use the following code to process and create a dictionary of all words present across all the documents. The dictionary will contain all unique words across the corpus. And each word in the dictionary will be treated as feature.

```
from sklearn.feature_extraction.text import CountVectorizer

# Initialize the CountVectorizer
count_vectorizer = CountVectorizer()
# Create the dictionary from the corpus
feature_vector = count_vectorizer.fit(train_ds.text)
# Get the feature names
features = feature_vector.get_feature_names()
print("Total number of features: ", len(features))
```

Total number of features: 2132

Total number of features or unique words in the corpus are 2132. The random sample of features can be obtained by using the following *random.sample()* method.

```
import random
random.sample(features, 10)
```

Let us look at some of the words randomly:

```
[ 'surprised',
  'apart',
  'rosie',
  'dating',
  'dan',
  'outta',
  'local',
  'eating',
  'aka',
  'learn']
```

Using the above dictionary, we can convert all the documents in the dataset to count vectors using *transform()* method of count vectorizer:

```
train_ds_features = count_vectorizer.transform(train_ds.text)
type(train_ds_features)
```

The dimension of the DataFrame *train_ds_features*, that contains the count vectors of all the documents, is given by *shape* variable of the DataFrame.

```
train_ds_features.shape
```

```
(6918, 2132)
```

After converting the document into a vector, we will have a sparse matrix with 2132 features or dimensions. Each document is represented by a count vector of 2132 dimensions and if a specific word exists in a document, the corresponding dimension of the vector will be set to the count of that word in the document. But most of the documents have only few words in them, hence most of the dimensions in the vectors will have value set to 0. That is a lot of 0's in the matrix! So, the matrix is stored as a sparse matrix. Sparse matrix representation stores only the non-zero values and their index in the vector. This optimizes storage as well as computational needs. To know how many actual non-zero values are present in the matrix, we can use *getnnz()* method on the DataFrame.

```
train_ds_features.getnnz()
```

65398

Computing proportion of non-zero values with respect to zero values in the matrix can be obtained by dividing the number of non-zero values (i.e., 65398) by the dimension of the matrix (i.e., 6918×2132), that is, $65398/(6918 \times 2132)$.

```
print("Density of the matrix: ",
      train_ds_features.getnnz() * 100 /
      (train_ds_features.shape[0] * train_ds_features.shape[1]))
```

Density of the matrix: 0.4434010415225908

The matrix has less than 1% non-zero values, that is, more than 99% values are zero values. This is a very sparse representation.

10.2.3.3 Displaying Document Vectors

To visualize the count vectors, we will convert this matrix into a *DataFrame* and set the column names to the actual feature names. The following commands are used for displaying the count vector:

```
# Converting the matrix to a dataframe
train_ds_df = pd.DataFrame(train_ds_features.todense())
# Setting the column names to the features i.e. words
train_ds_df.columns = features
```

Now, let us print the first record.

```
train_ds[0:1]
```

	Sentiment	Text
0	1	The Da Vinci Code book is just awesome.

We cannot print the complete vector as it has 2132 dimensions. Let us print the dimensions (words) from index 150 to 157. This index range contains the word *awesome*, which actually should have been encoded into 1.

```
train_ds_df.iloc[0:1, 150:157]
```

Away	awesome	awesomely	awesomeness	awesomest	awful	awkward
0	0	1	0	0	0	0

The feature *awesome* is set to 1, while the other features are set to 0. Now select all the columns as per the words in the sentence and print below.

```
train_ds_df[['the', 'da', "vinci", "code", "book", 'is', 'just',
            'awesome']] [0:1]
```

	the	da	vinci	code	book	is	Just	awesome
0	1	1	1	1	1	1	1	1

Yes, the features in the count vector are appropriately set to 1. The vector represents the sentence “*The Da Vinci Code book is just awesome*”.

10.2.3.4 Removing Low-frequency Words

One of the challenges of dealing with text is the number of words or features available in the corpus is too large. The number of features could easily go over tens of thousands. Some words would be common words and be present across most of the documents, while some words would be rare and present only in very few documents.

Frequency of each feature or word can be analyzed using histogram. To calculate the total occurrence of each feature or word, we will use *np.sum()* method.

```
# Summing up the occurrences of features column wise
features_counts = np.sum(train_ds_features.toarray(), axis = 0)
feature_counts_df = pd.DataFrame(dict(features = features,
                                         counts = features_counts))
```

The histogram in Figure 10.2 shows that a large number of features have very rare occurrences.

```
plt.figure(figsize=(12,5))
plt.hist(feature_counts_df.counts, bins=50, range = (0, 2000));
plt.xlabel('Frequency of words')
plt.ylabel('Density');
```

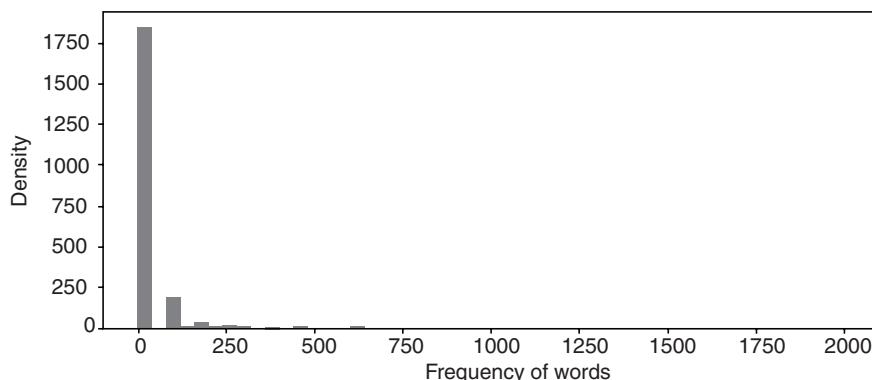


FIGURE 10.2 Histogram of frequently used words across the documents.

To find rare words in the dictionary, for example words that are present in any one of the document, we can filter the features by count equal to 1:

```
len(feature_counts_df[feature_counts_df.counts == 1])
```

1228

There are 1228 words which are present only once across all documents in the corpus. These words can be ignored. We can restrict the number of features by setting `max_features` parameters to 1000 while creating the count vectors.

```
# Initialize the CountVectorizer
count_vectorizer = CountVectorizer(max_features=1000)
# Create the dictionary from the corpus
feature_vector = count_vectorizer.fit(train_ds.text)
# Get the feature names
features = feature_vector.get_feature_names()
# Transform the document into vectors
train_ds_features = count_vectorizer.transform(train_ds.text)
# Count the frequency of the features
features_counts = np.sum(train_ds_features.toarray(), axis = 0)
feature_counts = pd.DataFrame(dict(features = features,
                                     counts = features_counts))
```

```
feature_counts.sort_values('counts',
                           ascending = False) [0:15]
```

Now print first 15 words and their count in descending order.

	Counts	Features
866	3306	The
37	2154	And
358	2093	Harry
675	2093	potter
138	2002	Code
934	2001	Vinci
178	2001	Da
528	2000	mountain
104	2000	brokeback
488	1624	Love
423	1520	ls

	Counts	Features
941	1176	Was
60	1127	awesome
521	1094	mission
413	1093	impossible

It can be noticed that the selected list of features contains words like **the**, **is**, **was**, **and**, etc. These words are irrelevant in determining the sentiment of the document. These words are called **stop words** and can be removed from the dictionary. This will reduce the number of features further.

10.2.3.5 Removing Stop Words

`sklearn.feature_extraction.text` provides a list of pre-defined stop words in English, which can be used as a reference to remove the stop words from the dictionary, that is, feature set.

```
from sklearn.feature_extraction import
text my_stop_words = text.ENGLISH_STOP_WORDS

#Printing first few stop words
print("Few stop words: ", list(my_stop_words)[0:10])
```

Few stop words: ['mill', 'another', 'every', 'whereafter', 'during', 'themselves', 'back', 'five', 'if', 'not']

Also, additional stop words can be added to this list for removal. For example, the movie names and the word “movie” itself can be a stop word in this case. These words can be added to the existing list of stop words for removal. For example,

```
# Adding custom words to the list of stop words
my_stop_words = text.ENGLISH_STOP_WORDS.union(['harry', 'potter',
'code', 'vinci', 'da','harry', 'mountain', 'movie', 'movies'])
```

10.2.3.6 Creating Count Vectors

All vectorizer classes take a list of stop words as a parameter and remove the stop words while building the dictionary or feature set. And these words will not appear in the count vectors representing the documents. We will create new count vectors by passing the `my_stop_words` as stop words list.

```
# Setting stop words list
count_vectorizer = CountVectorizer(stop_words = my_stop_words,
                                    max_features = 1000)
feature_vector = count_vectorizer.fit(train_ds.text)
```

```

train_ds_features = count_vectorizer.transform(train_ds.text)
features = feature_vector.get_feature_names()
features_counts = np.sum(train_ds_features.toarray(), axis = 0)
feature_counts = pd.DataFrame(dict(features = features,
                                     counts = features_counts))

feature_counts.sort_values("counts", ascending = False) [0:15]

```

Print the first 15 words and their count in descending order.

	Counts	Features
73	2000	brokeback
408	1624	love
39	1127	awesome
436	1094	mission
341	1093	impossible
390	974	like
745	602	sucks
743	600	sucked
297	578	hate
652	374	really
741	365	stupid
362	287	just
374	276	know
742	276	suck
409	256	loved

It can be noted that the stop words have been removed. But we also notice another problem. Many words appear in multiple forms. For example, *love* and *loved*. The vectorizer treats the two words as two separate words and hence creates two separate features. But, if a word has similar meaning in all its form, we can use only the root word as a feature. **Stemming** and **Lemmatization** are two popular techniques that are used to convert the words into root words.

1. **Stemming:** This removes the differences between inflected forms of a word to reduce each word to its root form. This is done by mostly chopping off the end of words (suffix). For instance, *love* or *loved* will be reduced to the root word *love*. The root form of a word may not even be a real word. For example, *awesome* and *awesomeness* will be stemmed to *awesom*. One problem with stemming is that chopping of words may result in words that are not part of vocabulary

(e.g., awesom). PorterStemmer and LancasterStemmer are two popular algorithms for stemming, which have rules on how to chop off a word.

2. **Lemmatization:** This takes the morphological analysis of the words into consideration. It uses a language dictionary (i.e., English dictionary) to convert the words to the root word. For example, **stemming** would fail to differentiate between *man* and *men*, while lemmatization can bring these words to its original form *man*.

Natural Language Toolkit (NLTK) is a very popular library in Python that has an extensive set of features for natural language processing. NLTK supports *PorterStemmer*, *EnglishStemmer*, and *LancasterStemmer* for stemming, while *WordNetLemmatizer* for lemmatization.

These features can be used in *CountVectorizer*, while creating count vectors. We need to create a utility method, which takes documents, tokenizes it to create words, stems the words and remove the stop words before returning the final set of words for creating vectors.

```
from nltk.stem.snowball import PorterStemmer

stemmer = PorterStemmer()
analyzer = CountVectorizer().build_analyzer()

#Custom function for stemming and stop word removal

def stemmed_words(doc):
    ### Stemming of words
    stemmed_words = [stemmer.stem(w) for w in analyzer(doc)]
    ### Remove the words in stop words list
    non_stop_words = [word for word in stemmed_words if not word in my_
                      stop_words]
    return non_stop_words
```

CountVectorizer takes a custom analyzer for stemming and stop word removal, before creating count vectors. So, the custom function *stemmed_words()* is passed as an analyzer.

```
count_vectorizer = CountVectorizer(analyzer=stemmed_words,
                                    max_features = 1000)
feature_vector = count_vectorizer.fit(train_ds.text)
train_ds_features = count_vectorizer.transform(train_ds.text)
features = feature_vector.get_feature_names()
features_counts = np.sum(train_ds_features.toarray(), axis = 0)
feature_counts = pd.DataFrame(dict(features =
                                     counts = features_counts))
feature_counts.sort_values("counts", ascending = False)[0:15]
```

Print the first 15 words and their count in descending order.

	Counts	Features
80	1930	brokeback
297	1916	harri
407	1837	love
803	1378	suck
922	1142	wa
43	1116	awesom
345	1090	imposs
433	1090	mission
439	1052	movi
393	823	like
299	636	hate
54	524	becaus
604	370	realli
796	364	stupid
379	354	know

It can be noted that words *love*, *loved*, *awesome* have all been stemmed to the root words.

10.2.3.7 Distribution of Words Across Different Sentiment

The words which have positive or negative meaning occur across documents of different sentiments. This could give an initial idea of how these words can be good features for predicting the sentiment of documents. For example, let us consider the word *awesome*.

```
# Convert the document vector matrix into dataframe
train_ds_df = pd.DataFrame(train_ds_features.todense())
# Assign the features names to the column
train_ds_df.columns = features
# Assign the sentiment labels to the train_ds
train_ds_df['sentiment'] = train_ds.sentiment

sn.barplot(x = 'sentiment', y = 'awesom', data = train_ds_df,
estimator=sum);
```

As shown in Figure 10.3, the word *awesom* (stemmed word for *awesome*) appears mostly in positive sentiment documents. How about a neutral word like *realli*?

```
sn.barplot(x = 'sentiment', y = 'realli', data = train_ds_df,
estimator=sum);
```

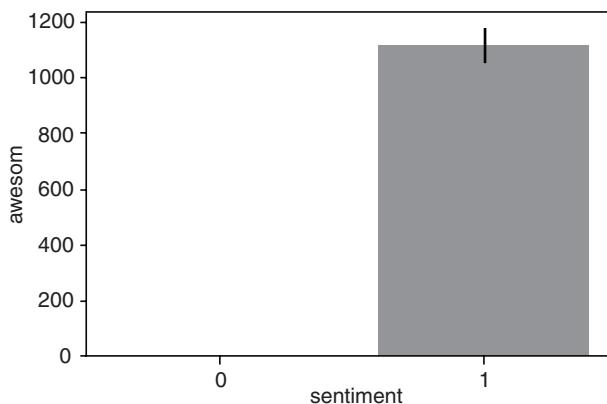


FIGURE 10.3 Frequency of word *awesom* in documents with positive versus negative sentiments.

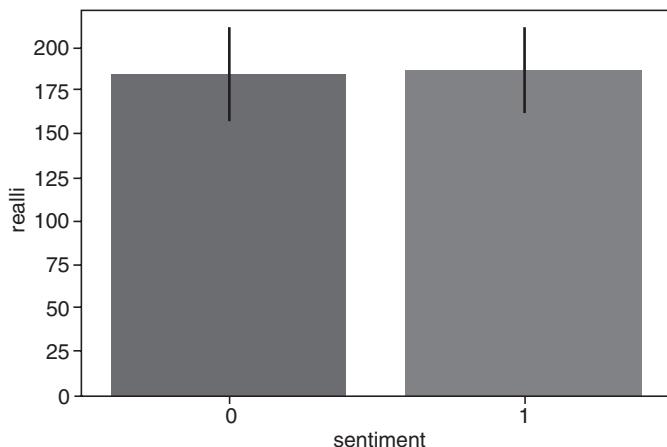


FIGURE 10.4 Frequency of word *realii* in documents with positive versus negative sentiments.

As shown in Figure 10.4, the word *realii* (stemmed word for *really*) occurs almost equally across positive and negative sentiments. How about the word *hate*?

```
sn.barplot(x = 'sentiment', y = 'hate', data = train_ds_df,
estimator=sum);
```

As shown in Figure 10.5, the word *hate* occurs mostly in negative sentiments than positive sentiments. This absolutely makes sense.

This gives us an initial idea that the words *awesom* and *hate* could be good features in determining sentiments of the document.

10.3 | NAÏVE-BAYES MODEL FOR SENTIMENT CLASSIFICATION

We will build a Naïve-Bayes model to classify sentiments. Naïve-Bayes classifier is widely used in Natural Language Processing and proved to give better results. It works on the concept of Bayes' theorem.

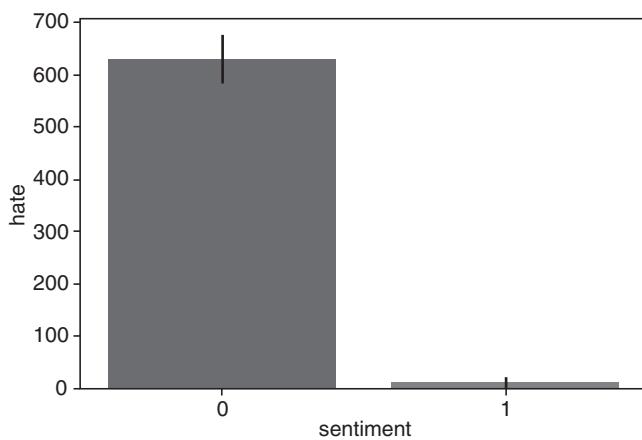


FIGURE 10.5 Frequency of word hate in documents with positive versus negative sentiments.

Assume that we would like to predict whether the probability of a document is positive (or negative) given that the document contains a word *awesome*. This can be computed if the probability of the word *awesome* appearing in a document given that it is a positive (or negative) sentiment multiplied by the probability of the document being positive (or negative).

$$P(\text{doc} = +\text{ve} \mid \text{word} = \text{awesome}) \propto P(\text{word} = \text{awesome} \mid \text{doc} = +\text{ve}) * P(\text{doc} = +\text{ve})$$

The posterior probability of the sentiment is computed from the **prior** probabilities of all the words it contains. The assumption is that the occurrences of the words in a document are considered independent and they do not influence each other. So, if the document contains N words and words are represented as W_1, W_2, \dots, W_N , then

$$P(\text{doc} = +\text{ve} \mid \text{word} = W_1, W_2, \dots, W_N) \propto \prod_{i=1}^N P_i(\text{word} = W_i \mid \text{doc} = +\text{ve}) * P(\text{doc} = +\text{ve})$$

sklearn.naive_bayes provides a class *BernoulliNB* which is a Naïve–Bayes classifier for multivariate Bernoulli models. *BernoulliNB* is designed for Binary/Boolean features (feature is either present or absent), which is the case here.

The steps involved in using Naïve–Bayes Model for sentiment classification are as follows:

1. Split dataset into train and validation sets.
2. Build the Naïve–Bayes model.
3. Find model accuracy.

We will discuss these in the following subsections.

10.3.1 | Split the Dataset

Split the dataset into 70:30 ratio for creating training and test datasets using the following code.

```
from sklearn.model_selection import train_test_split
```

```
train_X, test_X, train_y, test_y = train_test_split(train_ds_.features,
                                                    train_ds_.sentiment,
                                                    test_size = 0.3,
                                                    random_state = 42)
```

10.3.2 | Build Naïve–Bayes Model

Build Naïve–Bayes model using the training set.

```
from sklearn.naive_bayes import BernoulliNB
nb_clf = BernoulliNB()
nb_clf.fit(train_X.toarray(), train_y)

BernoulliNB(alpha=1.0, binarize=0.0, class_prior=None,
fit_prior=True)
```

10.3.3 | Make Prediction on Test Case

Predicted class will be the one which has the higher probability based on the Naïve–Bayes' probability calculation. Predict the sentiments of the test dataset using *predict()* method.

```
test_ds_predicted = nb_clf.predict(test_X.toarray())
```

10.3.4 | Finding Model Accuracy

Let us print the classification report.

```
from sklearn import metrics
print(metrics.classification_report(test_y, test_ds_predicted))

          Precision      recall      f1-score      support
0            0.98       0.97       0.98        873
1            0.98       0.99       0.98       1203
avg / total     0.98       0.98       0.98       2076
```

The model is classifying with very high accuracy. Both average precision and recall is about 98% for identifying positive and negative sentiment documents. Let us draw the confusion matrix (Figure 10.6).

```
from sklearn import metrics
cm = metrics.confusion_matrix(test_y, test_ds_predicted)
sn.heatmap(cm, annot=True, fmt=' .2f');
```

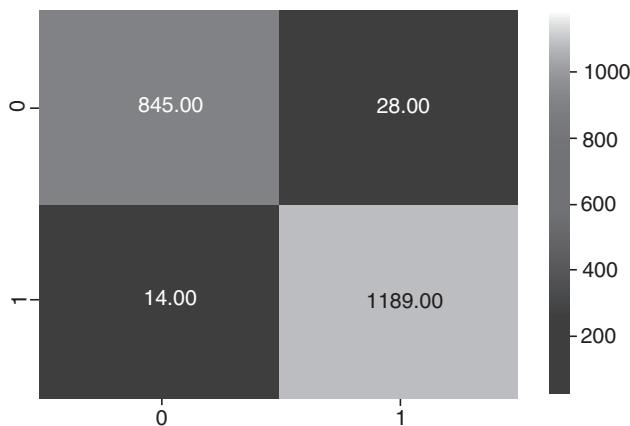


FIGURE 10.6 Confusion matrix of sentiment classification model.

In the confusion matrix, the rows represent the actual number positive and negative documents in the test set, whereas the columns represent what the model has predicted. Label 1 means positive sentiment and label 0 means negative sentiment. Figure 10.6 shows, as per the model prediction, that there are only 14 positive sentiment documents classified wrongly as negative sentiment documents (False Negatives) and there are only 28 negative sentiment documents classified wrongly as positive sentiment documents (False Positives). Rest all have been classified correctly.

10.4 | USING TF-IDF VECTORIZER

TfidfVectorizer is used to create both TF Vectorizer and TF-IDF Vectorizer. It takes a parameter *use_idf* (default *True*) to create TF-IDF vectors. If *use_idf* set to *False*, it will create only TF vectors and if it is set to *True*, it will create TF-IDF vectors.

```
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf_vectorizer = TfidfVectorizer(analyzer=stemmed_words,
                                    max_features = 1000)
feature_vector = tfidf_vectorizer.fit(train_ds.text)
train_ds_features = tfidf_vectorizer.transform(train_ds.text)
features = feature_vector.get_feature_names()
```

TF-IDF are continuous values and these continuous values associated with each class can be assumed to be distributed according to Gaussian distribution. So, Gaussian Naïve-Bayes can be used to classify these documents. We will use *GaussianNB*, which implements the Gaussian Naïve-Bayes algorithm for classification.

```
from sklearn.naive_bayes import GaussianNB
train_X, test_X, train_y, test_y = train_test_split(train_ds_features,
                                                    train_ds.sentiment,
                                                    test_size = 0.3,
                                                    random_state = 42)
```

```

nb_clf = GaussianNB()
nb_clf.fit(train_X.toarray(), train_y)

GaussianNB(priors=None)

test_ds_predicted = nb_clf.predict(test_X.toarray())
print(metrics.classification_report(test_y, test_ds_predicted))

```

	Precision	Recall	f1-score	support
0	0.96	0.96	0.96	873
1	0.97	0.97	0.97	1203
avg/total	0.97	0.97	0.97	2076

The precision and recall seem to be pretty much same. The accuracy is very high in this example as the dataset is clean and carefully curated. But it may not be the case in the real world.

10.5 | CHALLENGES OF TEXT ANALYTICS

The text could be highly context-specific. The language people use to describe movies may not be the same for other products, say apparels. So, the training data needs to come from similar context (distribution) for building the model. The language may be highly informal. The language people use on social media may be a mix of languages or emoticons. The training data also needs to contain similar examples for learning. Bag-of-words model completely ignores the structure of the sentence or sequence of words in the sentence. This can be overcome to a certain extent by using n-grams.

10.5.1 | Using n-Grams

The models we built in this chapter, created features out of each token or word. But the meaning of some of the words might be dependent on the words it precedes or succeeds, for example *not happy*. It should be considered as one feature and not as two different features. n-gram is a contiguous sequence of n words. When two consecutive words are treated as one feature, it is called bigram; three consecutive words is called trigram and so on.

We will write a new custom analyzer *get_stemmed_tokens()*, which splits the sentences and stems the words from them before creating n-grams. The following code block removes non-alphabetic characters and then applies stemming.

```

from nltk.stem import PorterStemmer
# Library for regular expressions
import re

stemmer = PorterStemmer()

def get_stemmed_tokens(doc):
    # Tokenize the documents to words
    all_tokens = [word for word in nltk.word_tokenize(doc)]
    clean_tokens = []

```

```

# Remove all characters other than alphabets. It takes a
# regex for matching.
for each_token in all_tokens:
    if re.search('[a-zA-Z]', each_token):
        clean_tokens.append(each_token)
# Stem the words
stemmed_tokens = [stemmer.stem(t) for t in clean_tokens]
return stemmed_tokens

```

Now TfidfVectorizer takes the above method as a custom tokenizer. It also takes *ngram_range* parameter, which is a tuple value, for creating n-grams. A value (1, 2) means create features with one word and two consecutive words as features.

```

tfidf_vectorizer = TfidfVectorizer(max_features=500,
                                   stop_words='english',
                                   tokenizer=get_stemmed_tokens,
                                   ngram_range=(1, 2))

feature_vector = tfidf_vectorizer.fit(train_ds.text)
train_ds_features = tfidf_vectorizer.transform(train_ds.text)
features = feature_vector.get_feature_names()

```

10.5.2 | Build the Model Using n-Grams

Split the dataset to 70:30 ratio for creating training and test datasets and then apply *BernoulliNB* for classification. We will apply the model to predict the test set and then print the classification report.

```

train_X, test_X, train_y, test_y = train_test_split(train_ds_features,
                                                    train_ds.sentiment,
                                                    test_size = 0.3,
                                                    random_state = 42)

nb_clf = BernoulliNB()
nb_clf.fit(train_X.toarray(), train_y)
test_ds_predicted = nb_clf.predict(test_X.toarray())
print(metrics.classification_report(test_y, test_ds_predicted))

```

	precision	recall	f1-score	support
0	1.00	0.94	0.97	873
1	0.96	1.00	0.98	1203
avg/total	0.97	0.97	0.97	2076

The *recall* for identifying positive sentiment documents (with label 1) have increased to almost 1.0.

CONCLUSION

1. Text data is unstructured data and needs extensive pre-processing before applying models.
2. Documents or sentences can be tokenized into unigrams or n-grams for building features.
3. The documents can be represented as vectors with words or n-grams as features. The vectors can be created using simple counts, TF (Term Frequency) or TF-IDF values.
4. A robust set of features can be created by removing stop words and applying stemming or lemmatization. Number of features can also be limited by selecting only features with higher frequencies.
5. Naïve–Bayes classification models (BernoulliNB) are the most widely used algorithms for classifying texts.

EXERCISES

Use the dataset “*Sentiment Labelled Sentences Data Set*” from University of California Irvine Machine Learning Database to answer the following questions.

The dataset can be downloaded from <https://archive.ics.uci.edu/ml/datasets/Sentiment+Labelled+Sentences>

Answer Questions 1 to 10 using the data file *yelp_labelled.txt*. The file has two columns. The first column is the text sentence and the second column is Score. Score is either 1 (for positive) or 0 (for negative).

1. Load the dataset and find out how many positive and how many negative sentiment documents are available in the dataset. Draw a bar plot to depict the number documents in each sentiment category. Print few samples of positive sentiment and few samples of negative sentiment documents.
2. Create count vectors of all the documents.
 - a. Find out how many total words or features exist in the corpus.
 - b. Which are the 10 words with highest frequency or total count?
 - c. How many words or features exist with total frequency less than 3 in the whole corpus?
3. Create the count vectors by using $\max_df = 0.8$ and $\min_df = 3$.
 - a. Find out how many total words or features exist.
 - b. Are there any stop words in the feature set?
4. Create the count vectors by removing stop words and using $\max_df = 0.8$ and $\min_df = 3$.
 - a. Find out how many total words or features exist.
 - b. Which are the 10 words with highest frequency or total count?
5. Split the documents into 80:20 ratio. 80% documents being in training set and 20% documents in test set.
6. Build a classification model using Logistic Regression to predict the probability of *s* documents being positive.
 - a. Find the top 10 words that have highest positive coefficient value.
 - b. Find the top 10 words that have highest negative coefficient value.
 - c. Explain why the words have positive and negative coefficients.
7. From the model built above, find the optimal cutoff value using Youden's index. Then classify the documents in the test set using the optimal cutoff value.
 - a. Create the confusion matrix.
 - b. Find out the precision and recall for positive sentiment cases.

8. Built a classification model using Random Forest model.
 - a. Use grid search mechanism to find the most optimal values for parameters
 - (i) n_estimators
 - (ii) max_depth
 - (iii) max_features
 - b. Obtain the best model using “roc_auc” score.
 - c. Obtain the feature importance from the model and find out how many words can explain 95% information in the corpus.
 - d. Create the confusion matrix.
 - e. Find out the precision and recall for positive sentiment cases.
9. Create *TF-IDF* vectors with the following parameters.
 - a. max_df = 0.7 and min_df = 3
 - b. Use bigrams for creating vectors
10. Build a classification model using the TF-IDF vectors.
 - a. Create the confusion matrix.
 - b. Find out the precision and recall for positive sentiment cases and compare it with the model built in Question 8.

REFERENCES

1. U Dinesh Kumar (2017). *Business Analytics: The Science of Data-Driven Decision Making*, Wiley India.
2. Data source: University of Michigan (2011) UMICH SI650—Sentiment Classification. <https://www.kaggle.com/c/si650winter11/data>
3. Scikit-learn documentations at <http://scikit-learn.org/>
4. Pandas Library: <https://pandas.pydata.org/>
5. Matplotlib Library: <https://matplotlib.org/>
6. Seaborn Library: <https://seaborn.pydata.org/>
7. From Group to Individual Labels using Deep Features, Kotzias et al., KDD 2015.



Index

A

AdaBoost, 233–234
alternative hypothesis, 81
Anaconda platform, 8–12
analysis of variance (ANOVA), 90–92
 in regression analysis, 105
apply() function, 44
apriori algorithm, 245, 296–297
artificial intelligence (AI), 1, 179
 relationship between DL, ML, and, 1
association rules (association rule mining), 292–299
auto-correlation, 131
auto-correlation function (ACF), 273
auto-regressive and moving average (ARMA), 264,
 278–279
auto-regressive integrated moving average (ARIMA),
 264, 273–287
autos-mpg.data, 45

B

Bag-of-Words (BoW) Model, 320–321
bar chart, 49–50
base classifier, 233
base_estimator, 234
Berkeley Software Distribution (BSD)-licensed
 library, 31
beta distribution, 80
bias-variance trade-off, 194–200
binary classification, 136
binary logistic regression model, 136
binomial distribution, 65
 CDF of, 66
 PMF of, 65
Boosting, 210, 233

Box-Jenkins methodology, 279
box plot, 52–54

C

central limit theorem (CLT), 80
chi-square goodness of fit test, 63, 89
Classification and Regression Tree (CART), 167
classification problem, 135–136
classification_report() method, 150
class probability, 135
clustering
 definition, 245
 interpreting clusters, 256–258
 product segments using, 252–258
co-efficient of determination (*R*-squared),
 103–104
collaborative filtering, 299–309
confusion matrix, 148–149, 229–231, 237
continuous random variables, 64–65
Cook's distance, 109–110
correlation values, 58–59
cosine similarity, 248
cost-based approach, 155–156
cost function, 180
credit classification, 137–156
 area under the curve (AUC), 151–153
 building logistic regression model, 143
 classification cut-off probability, 153–156
 creating confusion matrix, 148–149
 encoding categorical features, 141–142
 measuring accuracies, 150–151
 model diagnostics, 145–146
 predicting test data, 147–148
 printing model summary, 143–144

receiver operating characteristic (ROC) curve, 151–153
 splitting dataset, 143
 cross-validation, 96
 cumulative distribution function (CDF), 65
 of binomial distribution, 66
 cumulative probability distribution, 79–80
 cyclical component, 264

D

DataFrames in Python, 29
 applying operations to multiple columns, 43–44
 creating new columns, 40–41
 data visualization, 48–59
 display of first few records, 33
 filtering of records, 44
 findings, 33–36
 grouping and aggregating, 41–42
 handling of missing values, 45–48
 IPL dataset description using, 30–31
 joining, 42–43
 loading of dataset into, 31–33
 removing columns or rows, 45
 renaming columns, 43
 slicing and indexing of, 36–38
 sorting of column values, 39–40
 value counts and cross tabulations, 38–39

datasets and build recommendations, 291–292
 data visualization, 48–59
 bar chart, 49–50
 box plot, 52–54
 comparing distributions, 54–56
 correlation, 58–59
 distribution or density plot, 51–52
 drawing plots, 49
 heatmap, 58–59
 histogram, 50–51
 pair plot, 57–58
 scatter plot, 56–57

decision tree learning or classification trees, 166–174
 benefits of, 174
 displaying tree, 168–169
 measuring test accuracy, 168
 optimal criteria and `max_depth`, 173–174
 splitting dataset, 167–168
 using entropy criteria, 171–173

using Gini criteria, 168
 using Gini impurity index, 169–171
 deep learning (DL), 1
 relationship between AI, ML, and, 1
 dendrogram, 253–255
`describe()` method, 77
 Dickey-Fuller test, 282–283
 dictionary, 22–23
 discrete random variables, 64
 discriminant analysis, 2
`display.max_columns`, 33
 distribution or density plot, 51–52
`draw_pp_plot()` method, 126
`draw_roc_curve()` method, 234, 236
`drop()` method, 45
`dropna()` method, 48, 75
 Durbin–Watson test, 131

E

ElasticNet regression, 209–210
 Elbow curve method, 255–256
 ensemble methods, 225–226
 error function, 180
 error matrix, 148
 Euclidean distance, 247
 evolutionary learning algorithms, 2
 exponential distribution, 69–71
 example of, 70–71
 PDF of, 70
 exponential smoothing technique, 269–271

F

feature engineering, 4
 feature extraction, 3
`featureimportances`, 231–233
`filter()`, 25
`fit()` method, 102
 forecasting, 263–264
 calculating accuracy, 268–269
 using moving average, 266–268

`for` loop, 15
 F-Score, 150
 F-statistic, 105
 functional programming, 24–25

G

Gain chart, 156–159
 Gamma distribution, 80

Gaussian distribution, 72
 generalized linear models (GLM), 136
 geometric distribution, 80
 German credit rating dataset, 137–138
get_deciles() function, 163
 Gini impurity index, 169–171
 Gower's similarity coefficient, 248
 gradient boosting, 234–239
 gradient descent, 180
 gradient descent (GD) algorithm, 180–190
 bias and weights, 186–188
 cost function, 188–190
 for linear regression model, 182–190
GridSearchCV, 173, 223–225
 grid search for optimal parameters, 227–229

H

heatmap, 58–59
 hierarchical clustering, 258–259
hist() method, 50–51
 histogram, 50–51
 hypothesis tests, 81–90

I

imbalanced (or unbalanced) dataset, 211–214
 Indian Premier League (IPL) dataset, 30–31
influence_plot(), 110
info() method, 35–6
interval() method, 78
 irregular component, 264
isnull() method, 48
 item-based similarity, 307–309

J

Jaccard similarity coefficient, 248
 Jupyter notebook, 8–9, 11

K

key performance indicators (KPIs), 3
 K-fold cross-validation, 200–201
 K-means clustering algorithm, 248–252
 K-Nearest Neighbors (KNN), 210
 K-Nearest Neighbors (KNN) algorithm, 219–223

L

LASSO regression, 208–209
 learning algorithm, 1
learning_rate, 234
 leverage value, 110

lift chart, 156–159
 linear regression, 2
 building model, 201–206, 214–219
 cost function for, 181
 lists, 17–19
 logistic regression, 2
 building model, 143, 160–166
 loss function, 180

M

machine learning (ML), 1
 framework for ML algorithm development, 3–5
 need for, 3
 relationship between AI, DL, and, 1
 uses of, 3
 machine learning models, 191–194
 Markov chain, 2
 Markov decision process, 2
 Matplotlib, 8, 49
 matrix factorization, 313–314
max_depth, 235
max_features, 235
 MBA salary, predicting, 99–101
 mean absolute percentage error (MAPE), 111, 268–269
 mean centered series, 273
 mean square error (MSE), 111
merge() method, 42
 Minkowski distance, 248
 model building and feature selection, 4
 model deployment, 5
 model diagnostics
 analysis of variance (ANOVA) in regression analysis, 105
 co-efficient of determination (*R*-squared), 103–104
 hypothesis test for regression co-efficient, 104–105
 making prediction and measuring accuracy, 110–112
 outlier analysis, 108–110
 residual (error) analysis, 106–108
 using Python, 105–106
 moving average, 265–271
 multi-collinearity, 119–120
 multinomial classification, 136
 multiple linear regression, 2
 multiple linear regression (MLR), 112–126
 detecting influencers, 127–129

developing using Python, 114–116
 encoding categorical features, 116–117
 making predictions on validation set, 130–131
 multi-collinearity, 119–120, 123–125
 predicting auction price of players, 113–114
 residual analysis, 126–127
 split function, 117–118
 transformation in, 129–130
 using training dataset, 118–119
 variance inflation factor (VIF), 120–121
 multivariate time-series data, 264

N

n_estimators, 234–235
 n-grams, 335–336
 normal distribution, 72–80
 confidence interval, 78–79
 mean of, 77
 standard deviation of, 77
 variance of, 77
 null hypothesis, 81
 NumPy, 8

O

one hot encoding (OHE), 141
OnehotTransactions, 295
 one-sample t-test, 83–84
 one-way ANOVA, 90–92
 ordinary least square (OLS), 98
 outlier analysis, 108–110
Out-of-Bag (OOB) records, 225

P

paired sample *t*-test, 88
 pair plot, 57–58
pairplot() method, 57
 Pandas, 8
 cumsum() method, 232
 Pandas library, 29, 31–2
 partial auto-correlation function (PACF), 273
pct_change() method, 75
pd.get_dummies() method, 141
pd.read_csv method, 32
pd.to_numeric(), 47
 Poisson distribution, 68
 example of, 68–69
 PMF of, 68
 precision, 150

predict() method, 111, 147
 probability density function (PDF), 65
 definition, 79
 of exponential distribution, 70
 probability mass function (PMF), 65
 of binomial distribution, 65–66
 of Poisson distribution, 68
 probability value (*p*-value), 63
p-value, 92
 Python, 5–6
 applications, 5–6
 code and execution output, 11
 conditional statements, 13–14
 control flow statements, 15–16
 creating a program, 10
 data science applications, 7
 declaring variables, 12–13
 functional programming, 24–25
 functions, 16–17
 generating sequence of numbers, 14–15
 libraries, 7–8
 mean and median values, 26–27
 modules and packages, 25–26
 renaming a program, 10
 strings, 23–24
 working with collections, 17–23

R

random experiment, 63
 Random Forest, 210
 random forest model, 226–227
 with optimal parameter values, 229
 random variables, 64–65
 regularization, 206–207
 reinforcement learning algorithms, 2
rename() method, 43
reset_index(), 41
 residual (error) analysis, 106–108
 residual plot, 127
 residuals, 98
 Ridge regression, 208
 root mean square error (RMSE), 111, 131, 194, 269
 R-squared value, 103–104, 131, 194

S

sample space, 64
 scatter plot, 56–57
scikit-learn library, 190–201

SciPy, 8
 Seaborn, 49
 seasonal component, 264
 sensitivity, 150
 sentiment classification, 317–331
 creating count vectors, 322–324, 327–330
 displaying document vectors, 324–325
 distribution of words across different sentiment, 330–331
 exploring dataset, 319–320
 loading dataset, 318–319
 Naïve-Bayes model, 331–334
 removing low-frequency words, 325–327
 removing stop words, 327
 text pre-processing, 320–331
 set, 21–22
 Sigmoid function, 136
 significance (α) value, 63
significant_vars, 125
 simple linear regression, 95
 assumptions of, 98
 collection or extraction of data, 96
 complete code for building, 103
 deployment strategy, 97
 descriptive analytics, 97
 diagnostics tests, 97
 examples of, 95–96
 feature set (X) and outcome variable (Y), 101
 fitting of model, 102–103
 functional forms of, 97
 model accuracy, 97
 pre-processing of data, 96
 properties of, 98
 regression parameters, 98
 splitting of training and validation datasets, 102
 steps in building a regression model, 96–97
 sklearn, 8
sklearn.metrics package, 193
 smoothing constant, 269
sort_values() method, 39–40
 specificity, 150
square_me(), 24–25
 standard error of estimate of regression co-efficient, 104
 statistical learning, 180
stats.norm.cdf(), 79
stats.binom.cdf(), 67
stats.binom.pmf(), 66

statsmodel library, 101
stats.norm.cdf(), 79–80
stats.norm.interval(), 78
 stock exchange, daily returns at, 72–76
 strings, 23–24
summary2() function, 105
 sum of squared errors (SSE), 180
 supervised learning, 180
 supervised learning algorithms, 2
Surprise library, 309–313

T

t-distribution, 104
 term frequency-inverse document frequency (TF-IDF), 322
 vectorizer, 334–335
 term frequency (TF) vector, 321
 test for normality of residuals (P-P plot), 126–127
 test statistic, 105
 time-series data, 264
 decomposing, 271–272
 time-to-failure of an avionic system, 70–71
train_test_split() function, 102
t-tests, 63
 tuple, 19–20
 two-sample *t*-test, 85

U

univariate time-series data, 264
 University of California Irvine (UCI) machine learning laboratory, 137
 unsupervised learning, 180
 user-based similarity, 300–307

V

value_counts() method, 38
 variance inflation factor (VIF), 120–121
 vector space model, 248

W

Weibull distribution, 80
while loop, 15–16

Y

Youden's index, 153–155

Z

Z-score, 108–109
Z-test, 63, 82



About the Book

This book is written to provide a strong foundation in Machine Learning using Python libraries by providing real-life case studies and examples. It covers topics such as Foundations of Machine Learning, Introduction to Python, Descriptive Analytics and Predictive Analytics. Advanced Machine Learning concepts such as decision tree learning, random forest, boosting, recommender systems, and text analytics are covered. The book takes a balanced approach between theoretical understanding and practical applications. All the topics include real-world examples and provide step-by-step approach on how to explore, build, evaluate, and optimize machine learning models. The book is enriched with more than 15 years of teaching experience of the authors at various programs at the Indian Institute of Management, Bangalore and various training programs conducted for leading corporates across industries.

Key Features

- Each topic includes real-world examples with attention to theoretical aspects for better understanding.
- Provides step-by-step approach to enable practitioners to prepare, explore datasets, and build machine learning models.
- Topics covered range from basic statistics and probability distributions to advanced topics of machine learning such as regression, classification, clustering, forecasting, recommender systems, and text analytics.

Visit <https://www.wileyindia.com/catalog/product/view/id/6915/s/machine-learning-using-python/> for

- Datasets for readers to practice.
- Codes for all Python examples to provide quick start to the readers and help teachers planning to teach machine learning.

follow us on



facebook.com/wileyindia



twitter.com/wileyindiapl



linkedin.com/in/wileyindia



google.com/+wileyindia

Wiley India Pvt. Ltd.

Customer Care +91 120 6291100

csupport@wiley.com

www.wileyindia.com

www.wiley.com

ISBN 978-81-265-7990-7



9 788126 579907

WILEY