

O'REILLY®

Machine Learning & Data Science Blueprints for Finance

From Building Trading Strategies to
Robo-Advisors Using Python



Hariom Tatsat, Sahil Puri
& Brad Lookabaugh



Machine Learning & Data Science Blueprints for Finance

Machine learning and data science will significantly transform the finance industry in the next few years. With this practical guide, professionals at hedge funds, investment and retail banks, and fintech firms will learn how to build ML algorithms crucial to this industry. You'll examine ML concepts and over 20 case studies in supervised, unsupervised, and reinforcement learning, along with natural language processing (NLP).

Analysts, traders, researchers, and developers will also dive into portfolio management, algorithmic trading, derivative pricing, fraud detection, asset price prediction, sentiment analysis, and chatbot development. You'll explore real-life problems and learn scientifically sound solutions supported by code and examples.

This book includes:

- Supervised learning regression-based models for trading strategies and derivative pricing
- Supervised learning classification-based models for credit default risk prediction and fraud detection
- Dimensionality reduction techniques with case studies in portfolio management and yield curve construction
- Case studies using algorithms and clustering techniques to find similar objects in trading strategies and portfolio management
- Reinforcement learning models and techniques for building trading strategies, derivatives hedging, and portfolio management
- NLP techniques using Python libraries such as NLTK and Scikit-learn

PYTHON / FINANCE

US \$69.99 CAN \$92.99
ISBN: 978-1-492-07305-5
 5 6 9 9 9
9 781492 073055

Hariom Tatsat is a vice president in the Quantitative Analytics Division of an investment bank in New York. He has extensive experience in predictive modeling, financial instrument pricing, and risk management.

Sahil Puri is a quantitative researcher. He applies multiple statistical and machine learning-based techniques to a wide variety of problems.

Brad Lookabaugh is vice president in portfolio management at Unison Investment Management.

Twitter: @oreillymedia
facebook.com/oreilly

Machine Learning and Data Science Blueprints for Finance

*From Building Trading Strategies to
Robo-Advisors Using Python*

Hariom Tatsat, Sahil Puri, and Brad Lookabaugh

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Machine Learning and Data Science Blueprints for Finance

by Hariom Tatsat, Sahil Puri, and Brad Lookabaugh

Copyright © 2021 Hariom Tatsat, Sahil Puri, and Brad Lookabaugh. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Acquisitions Editor: Michelle Smith

Development Editor: Jeff Bleiel

Production Editor: Christopher Faucher

Copyeditor: Piper Editorial, LLC

Proofreader: Arthur Johnson

Indexer: WordCo Indexing Services, Inc.

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

October 2020: First Edition

Revision History for the First Edition

2020-09-29: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781492073055> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Machine Learning and Data Science Blueprints for Finance*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-07305-5

[LSCH]

Table of Contents

Preface.....	ix
--------------	----

Part I. The Framework

1. Machine Learning in Finance: The Landscape.....	1
Current and Future Machine Learning Applications in Finance	2
Algorithmic Trading	2
Portfolio Management and Robo-Advisors	2
Fraud Detection	3
Loans/Credit Card/Insurance Underwriting	3
Automation and Chatbots	3
Risk Management	4
Asset Price Prediction	4
Derivative Pricing	4
Sentiment Analysis	5
Trade Settlement	5
Money Laundering	5
Machine Learning, Deep Learning, Artificial Intelligence, and Data Science	5
Machine Learning Types	7
Supervised	7
Unsupervised	8
Reinforcement Learning	9
Natural Language Processing	10
Chapter Summary	11

2. Developing a Machine Learning Model in Python.....	13
Why Python?	13
Python Packages for Machine Learning	14
Python and Package Installation	15
Steps for Model Development in Python Ecosystem	15
Model Development Blueprint	16
Chapter Summary	29
3. Artificial Neural Networks.....	31
ANNs: Architecture, Training, and Hyperparameters	32
Architecture	32
Training	34
Hyperparameters	36
Creating an Artificial Neural Network Model in Python	40
Installing Keras and Machine Learning Packages	40
Running an ANN Model Faster: GPU and Cloud Services	43
Chapter Summary	45

Part II. Supervised Learning

4. Supervised Learning: Models and Concepts.....	49
Supervised Learning Models: An Overview	51
Linear Regression (Ordinary Least Squares)	52
Regularized Regression	55
Logistic Regression	57
Support Vector Machine	58
K-Nearest Neighbors	60
Linear Discriminant Analysis	62
Classification and Regression Trees	63
Ensemble Models	65
ANN-Based Models	71
Model Performance	73
Overfitting and Underfitting	73
Cross Validation	74
Evaluation Metrics	75
Model Selection	79
Factors for Model Selection	79
Model Trade-off	81
Chapter Summary	82

5. Supervised Learning: Regression (Including Time Series Models).....	83
Time Series Models	86
Time Series Breakdown	87
Autocorrelation and Stationarity	88
Traditional Time Series Models (Including the ARIMA Model)	90
Deep Learning Approach to Time Series Modeling	92
Modifying Time Series Data for Supervised Learning Models	95
Case Study 1: Stock Price Prediction	95
Blueprint for Using Supervised Learning Models to Predict a Stock Price	97
Case Study 2: Derivative Pricing	114
Blueprint for Developing a Machine Learning Model for Derivative Pricing	115
Case Study 3: Investor Risk Tolerance and Robo-Advisors	125
Blueprint for Modeling Investor Risk Tolerance and Enabling a Machine Learning-Based Robo-Advisor	127
Case Study 4: Yield Curve Prediction	141
Blueprint for Using Supervised Learning Models to Predict the Yield Curve	142
Chapter Summary	149
Exercises	150
6. Supervised Learning: Classification.....	151
Case Study 1: Fraud Detection	153
Blueprint for Using Classification Models to Determine Whether a Transaction Is Fraudulent	153
Case Study 2: Loan Default Probability	166
Blueprint for Creating a Machine Learning Model for Predicting Loan Default Probability	167
Case Study 3: Bitcoin Trading Strategy	179
Blueprint for Using Classification-Based Models to Predict Whether to Buy or Sell in the Bitcoin Market	180
Chapter Summary	190
Exercises	191

Part III. Unsupervised Learning

7. Unsupervised Learning: Dimensionality Reduction.....	195
Dimensionality Reduction Techniques	197
Principal Component Analysis	198
Kernel Principal Component Analysis	201

t-distributed Stochastic Neighbor Embedding	202
Case Study 1: Portfolio Management: Finding an Eigen Portfolio	202
Blueprint for Using Dimensionality Reduction for Asset Allocation	203
Case Study 2: Yield Curve Construction and Interest Rate Modeling	217
Blueprint for Using Dimensionality Reduction to Generate a Yield Curve	218
Case Study 3: Bitcoin Trading: Enhancing Speed and Accuracy	227
Blueprint for Using Dimensionality Reduction to Enhance a Trading Strategy	228
Chapter Summary	236
Exercises	236
8. Unsupervised Learning: Clustering.....	237
Clustering Techniques	239
k-means Clustering	239
Hierarchical Clustering	240
Affinity Propagation Clustering	242
Case Study 1: Clustering for Pairs Trading	243
Blueprint for Using Clustering to Select Pairs	244
Case Study 2: Portfolio Management: Clustering Investors	259
Blueprint for Using Clustering for Grouping Investors	260
Case Study 3: Hierarchical Risk Parity	267
Blueprint for Using Clustering to Implement Hierarchical Risk Parity	268
Chapter Summary	277
Exercises	277

Part IV. Reinforcement Learning and Natural Language Processing

9. Reinforcement Learning.....	281
Reinforcement Learning—Theory and Concepts	283
RL Components	284
RL Modeling Framework	288
Reinforcement Learning Models	293
Key Challenges in Reinforcement Learning	298
Case Study 1: Reinforcement Learning-Based Trading Strategy	298
Blueprint for Creating a Reinforcement Learning-Based Trading Strategy	300
Case Study 2: Derivatives Hedging	316
Blueprint for Implementing a Reinforcement Learning-Based Hedging Strategy	317
Case Study 3: Portfolio Allocation	334

Blueprint for Creating a Reinforcement Learning–Based Algorithm for Portfolio Allocation	335
Chapter Summary	344
Exercises	345
10. Natural Language Processing.....	347
Natural Language Processing: Python Packages	349
NLTK	349
TextBlob	349
spaCy	350
Natural Language Processing: Theory and Concepts	350
1. Preprocessing	351
2. Feature Representation	356
3. Inference	360
Case Study 1: NLP and Sentiment Analysis–Based Trading Strategies	362
Blueprint for Building a Trading Strategy Based on Sentiment Analysis	363
Case Study 2: Chatbot Digital Assistant	383
Blueprint for Creating a Custom Chatbot Using NLP	385
Case Study 3: Document Summarization	393
Blueprint for Using NLP for Document Summarization	394
Chapter Summary	400
Exercises	400
Index.....	401

Preface

The value of machine learning (ML) in finance is becoming more apparent each day. Machine learning is expected to become crucial to the functioning of financial markets. Analysts, portfolio managers, traders, and chief investment officers should all be familiar with ML techniques. For banks and other financial institutions striving to improve financial analysis, streamline processes, and increase security, ML is becoming the technology of choice. The use of ML in institutions is an increasing trend, and its potential for improving various systems can be observed in trading strategies, pricing, and risk management.

Although machine learning is making significant inroads across all verticals of the financial services industry, there is a gap between the ideas and the implementation of machine learning algorithms. A plethora of material is available on the web in these areas, yet very little is organized. Additionally, most of the literature is limited to trading algorithms only. *Machine Learning and Data Science Blueprints for Finance* fills this void and provides a machine learning toolbox customized for the financial market that allows the readers to be part of the machine learning revolution. This book is not limited to investing or trading strategies; it focuses on leveraging the art and craft of building ML-driven algorithms that are crucial in the finance industry.

Implementing machine learning models in finance is easier than commonly believed. There is also a misconception that big data is needed for building machine learning models. The case studies in this book span almost all areas of machine learning and aim to handle such misconceptions. This book not only will cover the theory and case studies related to using ML in trading strategies but also will delve deep into other critical “need-to-know” concepts such as portfolio management, derivative pricing, fraud detection, corporate credit ratings, robo-advisor development, and chatbot development. It will address real-life problems faced by practitioners and provide scientifically sound solutions supported by code and examples.

The [Python codebase for this book on GitHub](#) will be useful and serve as a starting point for industry practitioners working on their projects. The examples and case studies shown in the book demonstrate techniques that can easily be applied to a wide range of datasets. The futuristic case studies, such as reinforcement learning for trading, building a robo-advisor, and using machine learning for instrument pricing, inspire readers to think outside the box and motivate them to make the best of the models and data available.

Who This Book Is For

The format of the book and the list of topics covered make it suitable for professionals working in hedge funds, investment and retail banks, and fintech firms. They may have titles such as data scientist, data engineer, quantitative researcher, machine learning architect, or software engineer. Additionally, the book will be useful for those professionals working in support functions, such as compliance and risk.

Whether a quantitative trader in a hedge fund is looking for ideas in using reinforcement learning for trading cryptocurrency or an investment bank quant is looking for machine learning-based techniques to improve the calibration speed of pricing models, this book will add value. The theory, concepts, and codebase mentioned in the book will be extremely useful at every step of the model development lifecycle, from idea generation to model implementation. Readers can use the shared codebase and test the proposed solutions themselves, allowing for a hands-on reader experience. The readers should have a basic knowledge of statistics, machine learning, and Python.

How This Book Is Organized

This book provides a comprehensive introduction to how machine learning and data science can be used to design models across different areas in finance. It is organized into four parts.

Part I: The Framework

The first part provides an overview of machine learning in finance and the building blocks of machine learning implementation. These chapters serve as the foundation for the case studies covering different machine learning types presented in the rest of the book.

The chapters under the first part are as follows:

Chapter 1, Machine Learning in Finance: The Landscape

This chapter provides an overview of applications of machine learning in finance and provides a brief overview of several types of machine learning.

Chapter 2, Developing a Machine Learning Model in Python

This chapter looks at the Python-based ecosystem for machine learning. It also covers the steps for machine learning model development in the Python framework.

Chapter 3, Artificial Neural Networks

Given that an artificial neural network (ANN) is a primary algorithm used across all types of machine learning, this chapter looks at the details of ANNs, followed by a detailed implementation of an ANN model using Python libraries.

Part II: Supervised Learning

The second part covers fundamental supervised learning algorithms and illustrates specific applications and case studies.

The chapters under the second part are as follows:

Chapter 4, Supervised Learning: Models and Concepts

This chapter provides an introduction to supervised learning techniques (both classification and regression). Given that a lot of models are common between classification and regression, the details of those models are presented together along with other concepts such as model selection and evaluation metrics for classification and regression.

Chapter 5, Supervised Learning: Regression (Including Time Series Models)

Supervised learning-based regression models are the most commonly used machine learning models in finance. This chapter covers the models from basic linear regression to advance deep learning. The case studies covered in this section include models for stock price prediction, derivatives pricing, and portfolio management.

Chapter 6, Supervised Learning: Classification

Classification is a subcategory of supervised learning in which the goal is to predict the categorical class labels of new instances, based on past observations. This section discusses several case studies based on classification-based techniques, such as logistic regression, support vector machines, and random forests.

Part III: Unsupervised Learning

The third part covers the fundamental unsupervised learning algorithms and offers applications and case studies.

The chapters under the third part are as follows:

Chapter 7, Unsupervised Learning: Dimensionality Reduction

This chapter describes the essential techniques to reduce the number of features in a dataset while retaining most of their useful and discriminatory information. It also discusses the standard approach to dimensionality reduction via principal component analysis and covers case studies in portfolio management, trading strategy, and yield curve construction.

Chapter 8, Unsupervised Learning: Clustering

This chapter covers the algorithms and techniques related to clustering and identifying groups of objects that share a degree of similarity. The case studies utilizing clustering in trading strategies and portfolio management are covered in this chapter.

Part IV: Reinforcement Learning and Natural Language Processing

The fourth part covers the reinforcement learning and natural language processing (NLP) techniques.

The chapters under the fourth part are as follows:

Chapter 9, Reinforcement Learning

This chapter covers concepts and case studies on reinforcement learning, which have great potential for application in the finance industry. Reinforcement learning's main idea of "maximizing the rewards" aligns perfectly with the core motivation of several areas within finance. Case studies related to trading strategies, portfolio optimization, and derivatives hedging are covered in this chapter.

Chapter 10, Natural Language Processing

This chapter describes the techniques in natural language processing and discusses the essential steps to transform textual data into meaningful representations across several areas in finance. Case studies related to sentiment analysis, chatbots, and document interpretation are covered.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.



This element signifies a tip or suggestion.



This element signifies a general note.



This element indicates a warning or caution.



This element indicates a blueprint.

Using Code Presented in the Book

All code in this book (case studies and master template) is available at the GitHub directory: <https://github.com/tatsath/fin-ml>. The code is hosted on a cloud platform, so every case study can be run without installing a package on a local machine by clicking <https://mybinder.org/v2/gh/tatsath/fin-ml/master>.

This book is here to help you get your job done. In general, if example code is offered, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not

require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: *Machine Learning and Data Science Blueprints for Finance* by Hariom Tatsat, Sahil Puri, and Brad Lookabaugh (O'Reilly, 2021), 978-1-492-07305-5.

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

Python Libraries

The book uses Python 3.7. Installing the Conda package manager is recommended in order to create a Conda environment to install the requisite libraries. Installation instructions are available on the GitHub repo's [README file](#).

O'Reilly Online Learning

 For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit <http://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <https://oreil.ly/ML-and-data-science-blueprints>.

Email bookquestions@oreilly.com to comment or ask technical questions about this book.

For news and information about our books and courses, visit <http://oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://youtube.com/oreillymedia>

Acknowledgments

We want to thank all those who helped to make this book a reality. Special thanks to Jeff Bleiel for honest, insightful feedback, and for guiding us through the entire process. We are incredibly grateful to Juan Manuel Contreras, Chakri Cherukuri, and Gregory Bronner, who took time out of their busy lives to review our book in so much detail. The book benefited from their valuable feedback and suggestions. Many thanks as well to O'Reilly's fantastic staff, in particular Michelle Smith for believing in this project and helping us define its scope.

Special Thanks from Hariom

I would like to thank my wife, Prachi, and my parents for their love and support. Special thanks to my father for encouraging me in all of my pursuits and being a continuous source of inspiration.

Special Thanks from Sahil

Thanks to my family, who always encouraged and supported me in all endeavors.

Special Thanks from Brad

Thank you to my wife, Megan, for her endless love and support.

PART I

The Framework

CHAPTER 1

Machine Learning in Finance: The Landscape

Machine learning promises to shake up large swathes of finance

—The Economist (2017)

There is a new wave of machine learning and data science in finance, and the related applications will transform the industry over the next few decades.

Currently, most financial firms, including hedge funds, investment and retail banks, and fintech firms, are adopting and investing heavily in machine learning. Going forward, financial institutions will need a growing number of machine learning and data science experts.

Machine learning in finance has become more prominent recently due to the availability of vast amounts of data and more affordable computing power. The use of data science and machine learning is exploding exponentially across all areas of finance.

The success of machine learning in finance depends upon building efficient infrastructure, using the correct toolkit, and applying the right algorithms. The concepts related to these building blocks of machine learning in finance are demonstrated and utilized throughout this book.

In this chapter, we provide an introduction to the current and future application of machine learning in finance, including a brief overview of different types of machine learning. This chapter and the two that follow serve as the foundation for the case studies presented in the rest of the book.

Current and Future Machine Learning Applications in Finance

Let's take a look at some promising machine learning applications in finance. The case studies presented in this book cover all the applications mentioned here.

Algorithmic Trading

Algorithmic trading (or simply *algo trading*) is the use of algorithms to conduct trades autonomously. With origins going back to the 1970s, algorithmic trading (sometimes called Automated Trading Systems, which is arguably a more accurate description) involves the use of automated preprogrammed trading instructions to make extremely fast, objective trading decisions.

Machine learning stands to push algorithmic trading to new levels. Not only can more advanced strategies be employed and adapted in real time, but machine learning-based techniques can offer even more avenues for gaining special insight into market movements. Most hedge funds and financial institutions do not openly disclose their machine learning-based approaches to trading (for good reason), but machine learning is playing an increasingly important role in calibrating trading decisions in real time.

Portfolio Management and Robo-Advisors

Asset and wealth management firms are exploring potential artificial intelligence (AI) solutions for improving their investment decisions and making use of their troves of historical data.

One example of this is the use of *robo-advisors*, algorithms built to calibrate a financial portfolio to the goals and risk tolerance of the user. Additionally, they provide automated financial guidance and service to end investors and clients.

A user enters their financial goals (e.g., to retire at age 65 with \$250,000 in savings), age, income, and current financial assets. The advisor (the *allocator*) then spreads investments across asset classes and financial instruments in order to reach the user's goals.

The system then calibrates to changes in the user's goals and real-time changes in the market, aiming always to find the best fit for the user's original goals. Robo-advisors have gained significant traction among consumers who do not need a human advisor to feel comfortable investing.

Fraud Detection

Fraud is a massive problem for financial institutions and one of the foremost reasons to leverage machine learning in finance.

There is currently a significant data security risk due to high computing power, frequent internet use, and an increasing amount of company data being stored online. While previous financial fraud detection systems depended heavily on complex and robust sets of rules, modern fraud detection goes beyond following a checklist of risk factors—it actively learns and calibrates to new potential (or real) security threats.

Machine learning is ideally suited to combating fraudulent financial transactions. This is because machine learning systems can scan through vast datasets, detect unusual activities, and flag them instantly. Given the incalculably high number of ways that security can be breached, genuine machine learning systems will be an absolute necessity in the days to come.

Loans/Credit Card/Insurance Underwriting

Underwriting could be described as a perfect job for machine learning in finance, and indeed there is a great deal of worry in the industry that machines will replace a large swath of underwriting positions that exist today.

Especially at large companies (big banks and publicly traded insurance firms), machine learning algorithms can be trained on millions of examples of consumer data and financial lending or insurance outcomes, such as whether a person defaulted on their loan or mortgage.

Underlying financial trends can be assessed with algorithms and continuously analyzed to detect trends that might influence lending and underwriting risk in the future. Algorithms can perform automated tasks such as matching data records, identifying exceptions, and calculating whether an applicant qualifies for a credit or insurance product.

Automation and Chatbots

Automation is patently well suited to finance. It reduces the strain that repetitive, low-value tasks put on human employees. It tackles the routine, everyday processes, freeing up teams to finish their high-value work. In doing so, it drives enormous time and cost savings.

Adding machine learning and AI into the automation mix adds another level of support for employees. With access to relevant data, machine learning and AI can provide an in-depth data analysis to support finance teams with difficult decisions. In some cases, it may even be able to recommend the best course of action for employees to approve and enact.

AI and automation in the financial sector can also learn to recognize errors, reducing the time wasted between discovery and resolution. This means that human team members are less likely to be delayed in providing their reports and are able to complete their work with fewer errors.

AI chatbots can be implemented to support finance and banking customers. With the rise in popularity of live chat software in banking and finance businesses, chatbots are the natural evolution.

Risk Management

Machine learning techniques are transforming how we approach risk management. All aspects of understanding and controlling risk are being revolutionized through the growth of solutions driven by machine learning. Examples range from deciding how much a bank should lend a customer to improving compliance and reducing model risk.

Asset Price Prediction

Asset price prediction is considered the most frequently discussed and most sophisticated area in finance. Predicting asset prices allows one to understand the factors that drive the market and speculate asset performance. Traditionally, asset price prediction was performed by analyzing past financial reports and market performance to determine what position to take for a specific security or asset class. However, with a tremendous increase in the amount of financial data, the traditional approaches for analysis and stock-selection strategies are being supplemented with ML-based techniques.

Derivative Pricing

Recent machine learning successes, as well as the fast pace of innovation, indicate that ML applications for derivatives pricing should become widely used in the coming years. The world of Black-Scholes models, volatility smiles, and Excel spreadsheet models should wane as more advanced methods become readily available.

The classic derivative pricing models are built on several impractical assumptions to reproduce the empirical relationship between the underlying input data (strike price, time to maturity, option type) and the price of the derivatives observed in the market. Machine learning methods do not rely on several assumptions; they just try to estimate a function between the input data and price, minimizing the difference between the results of the model and the target.

The faster deployment times achieved with state-of-the-art ML tools are just one of the advantages that will accelerate the use of machine learning in derivatives pricing.

Sentiment Analysis

Sentiment analysis involves the perusal of enormous volumes of unstructured data, such as videos, transcriptions, photos, audio files, social media posts, articles, and business documents, to determine market sentiment. Sentiment analysis is crucial for all businesses in today's workplace and is an excellent example of machine learning in finance.

The most common use of sentiment analysis in the financial sector is the analysis of financial news—in particular, predicting the behaviors and possible trends of markets. The stock market moves in response to myriad human-related factors, and the hope is that machine learning will be able to replicate and enhance human intuition about financial activity by discovering new trends and telling signals.

However, much of the future applications of machine learning will be in understanding social media, news trends, and other data sources related to predicting the sentiments of customers toward market developments. It will not be limited to predicting stock prices and trades.

Trade Settlement

Trade settlement is the process of transferring securities into the account of a buyer and cash into the seller's account following a transaction of a financial asset.

Despite the majority of trades being settled automatically, and with little or no interaction by human beings, about 30% of trades need to be settled manually.

The use of machine learning not only can identify the reason for failed trades, but it also can analyze why the trades were rejected, provide a solution, and predict which trades may fail in the future. What usually would take a human being five to ten minutes to fix, machine learning can do in a fraction of a second.

Money Laundering

A United Nations report estimates that the amount of money laundered worldwide per year is 2%–5% of global GDP. Machine learning techniques can analyze internal, publicly existing, and transactional data from a client's broader network in an attempt to spot money laundering signs.

Machine Learning, Deep Learning, Artificial Intelligence, and Data Science

For the majority of people, the terms *machine learning*, *deep learning*, *artificial intelligence*, and *data science* are confusing. In fact, a lot of people use one term interchangeably with the others.

Figure 1-1 shows the relationships between AI, machine learning, deep learning and data science. Machine learning is a subset of AI that consists of techniques that enable computers to identify patterns in data and to deliver AI applications. Deep learning, meanwhile, is a subset of machine learning that enables computers to solve more complex problems.

Data science isn't exactly a subset of machine learning, but it uses machine learning, deep learning, and AI to analyze data and reach actionable conclusions. It combines machine learning, deep learning and AI with other disciplines such as big data analytics and cloud computing.

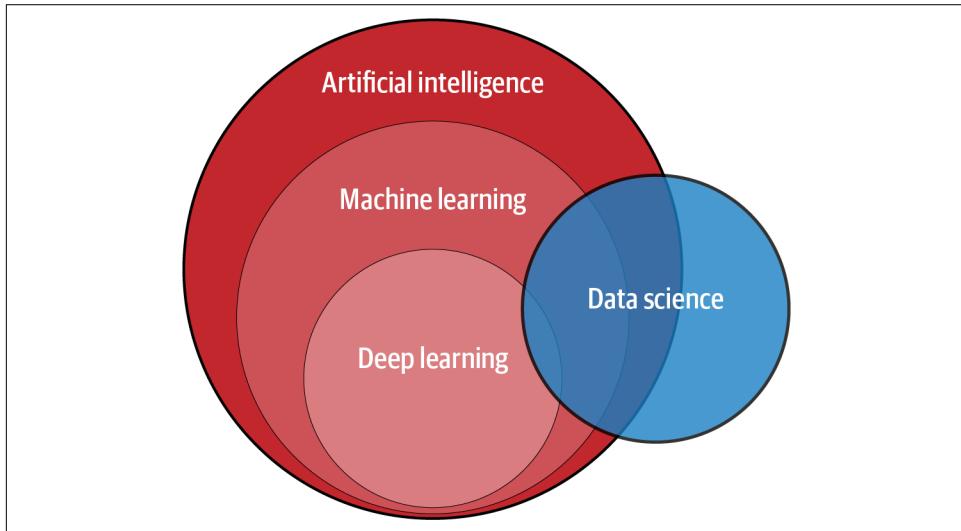


Figure 1-1. AI, machine learning, deep learning, and data science

The following is a summary of the details about artificial intelligence, machine learning, deep learning, and data science:

Artificial intelligence

Artificial intelligence is the field of study by which a computer (and its systems) develop the ability to successfully accomplish complex tasks that usually require human intelligence. These tasks include, but are not limited to, visual perception, speech recognition, decision making, and translation between languages. AI is usually defined as the science of making computers do things that require intelligence when done by humans.

Machine learning

Machine learning is an application of artificial intelligence that provides the AI system with the ability to automatically learn from the environment and apply those lessons to make better decisions. There are a variety of algorithms that

machine learning uses to iteratively learn, describe and improve data, spot patterns, and then perform actions on these patterns.

Deep learning

Deep learning is a subset of machine learning that involves the study of algorithms related to artificial neural networks that contain many blocks (or layers) stacked on each other. The design of deep learning models is inspired by the biological neural network of the human brain. It strives to analyze data with a logical structure similar to how a human draws conclusions.

Data science

Data science is an interdisciplinary field similar to data mining that uses scientific methods, processes, and systems to extract knowledge or insights from data in various forms, either structured or unstructured. Data science is different from ML and AI because its goal is to gain insight into and understanding of the data by using different scientific tools and techniques. However, there are several tools and techniques common to both ML and data science, some of which are demonstrated in this book.

Machine Learning Types

This section will outline all types of machine learning that are used in different case studies presented in this book for various financial applications. The three types of machine learning, as shown in [Figure 1-2](#), are supervised learning, unsupervised learning, and reinforcement learning.

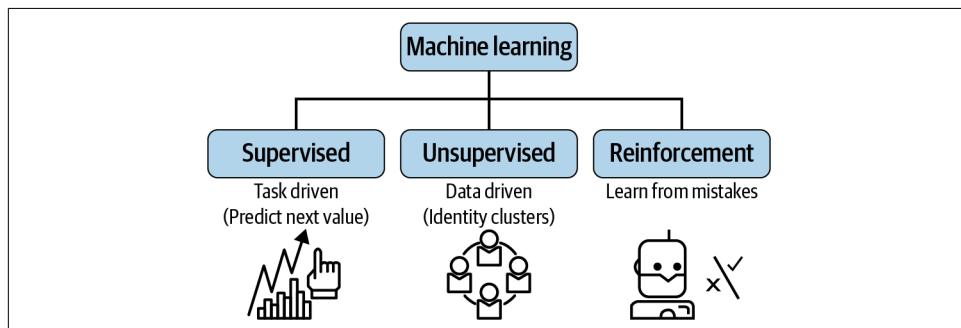


Figure 1-2. Machine learning types

Supervised

The main goal in *supervised learning* is to train a model from labeled data that allows us to make predictions about unseen or future data. Here, the term *supervised* refers to a set of samples where the desired output signals (labels) are already known. There are two types of supervised learning algorithms: classification and regression.

Classification

Classification is a subcategory of supervised learning in which the goal is to predict the categorical class labels of new instances based on past observations.

Regression

Regression is another subcategory of supervised learning used in the prediction of continuous outcomes. In regression, we are given a number of predictor (explanatory) variables and a continuous response variable (outcome or target), and we try to find a relationship between those variables that allows us to predict an outcome.

An example of regression versus classification is shown in [Figure 1-3](#). The chart on the left shows an example of regression. The continuous response variable is return, and the observed values are plotted against the predicted outcomes. On the right, the outcome is a categorical class label, whether the market is bull or bear, and is an example of classification.

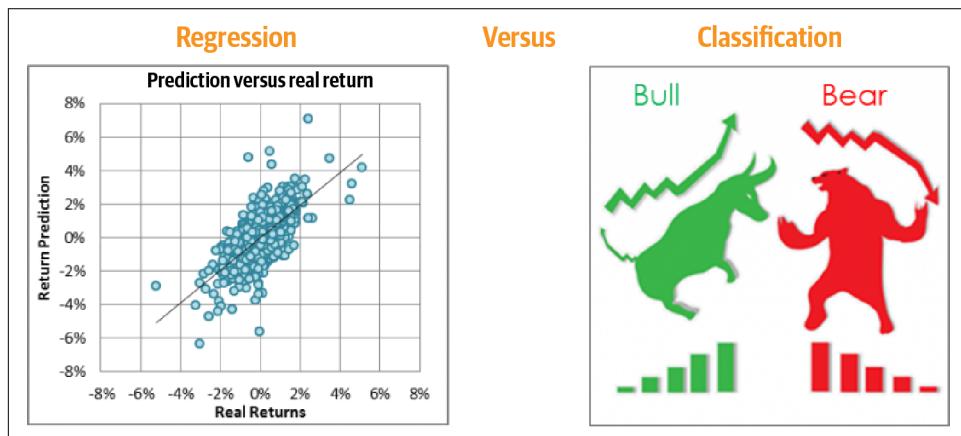


Figure 1-3. Regression versus classification

Unsupervised

Unsupervised learning is a type of machine learning used to draw inferences from datasets consisting of input data without labeled responses. There are two types of unsupervised learning: dimensionality reduction and clustering.

Dimensionality reduction

Dimensionality reduction is the process of reducing the number of features, or variables, in a dataset while preserving information and overall model performance. It is a common and powerful way to deal with datasets that have a large number of dimensions.

Figure 1-4 illustrates this concept, where the dimension of data is converted from two dimensions (X_1 and X_2) to one dimension (Z_1). Z_1 conveys similar information embedded in X_1 and X_2 and reduces the dimension of the data.

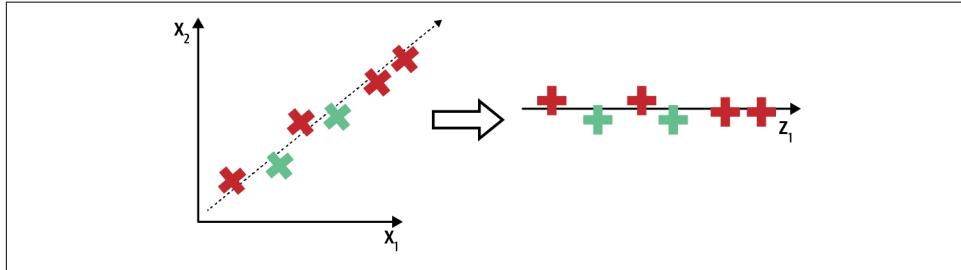


Figure 1-4. Dimensionality reduction

Clustering

Clustering is a subcategory of unsupervised learning techniques that allows us to discover hidden structures in data. The goal of clustering is to find a natural grouping in data so that items in the same cluster are more similar to each other than to those from different clusters.

An example of clustering is shown in **Figure 1-5**, where we can see the entire data clustered into two distinct groups by the clustering algorithm.

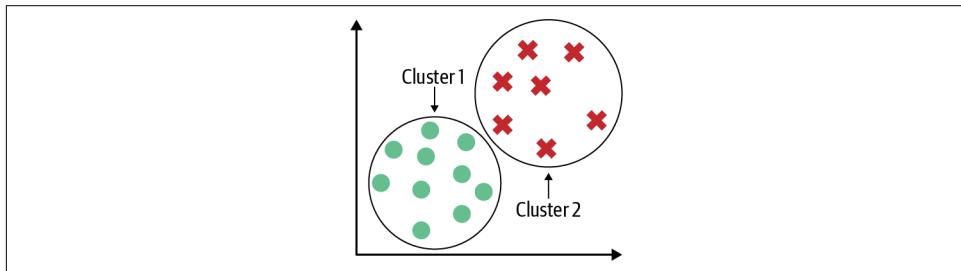


Figure 1-5. Clustering

Reinforcement Learning

Learning from experiences, and the associated rewards or punishments, is the core concept behind *reinforcement learning* (RL). It is about taking suitable actions to maximize reward in a particular situation. The learning system, called an *agent*, can observe the environment, select and perform actions, and receive rewards (or penalties in the form of negative rewards) in return, as shown in **Figure 1-6**.

Reinforcement learning differs from supervised learning in this way: In supervised learning, the training data has the answer key, so the model is trained with the correct answers available. In reinforcement learning, there is no explicit answer. The learning

system (agent) decides what to do to perform the given task and learns whether that was a correct action based on the reward. The algorithm determines the answer key through its experience.

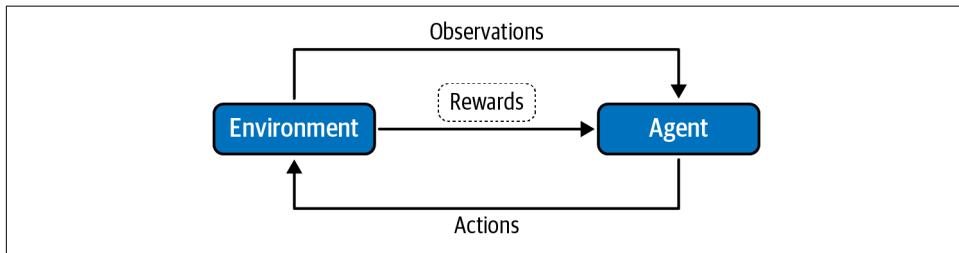


Figure 1-6. Reinforcement learning

The steps of the reinforcement learning are as follows:

1. First, the agent interacts with the environment by performing an action.
2. Then the agent receives a reward based on the action it performed.
3. Based on the reward, the agent receives an observation and understands whether the action was good or bad. If the action was good—that is, if the agent received a positive reward—then the agent will prefer performing that action. If the reward was less favorable, the agent will try performing another action to receive a positive reward. It is basically a trial-and-error learning process.

Natural Language Processing

Natural language processing (NLP) is a branch of AI that deals with the problems of making a machine understand the structure and the meaning of natural language as used by humans. Several techniques of machine learning and deep learning are used within NLP.

NLP has many applications in the finance sectors in areas such as sentiment analysis, chatbots, and document processing. A lot of information, such as sell side reports, earnings calls, and newspaper headlines, is communicated via text message, making NLP quite useful in the financial domain.

Given the extensive application of NLP algorithms based on machine learning in finance, there is a separate chapter of this book ([Chapter 10](#)) dedicated to NLP and related case studies.

Chapter Summary

Machine learning is making significant inroads across all the verticals of the financial services industry. This chapter covered different applications of machine learning in finance, from algorithmic trading to robo-advisors. These applications will be covered in the case studies later in this book.

Next Steps

In terms of the platforms used for machine learning, the Python ecosystem is growing and is one of the most dominant programming languages for machine learning. In the next chapter, we will learn about the model development steps, from data preparation to model deployment in a Python-based framework.

Developing a Machine Learning Model in Python

In terms of the platforms used for machine learning, there are many algorithms and programming languages. However, the Python ecosystem is one of the most dominant and fastest-growing programming languages for machine learning.

Given the popularity and high adoption rate of Python, we will use it as the main programming language throughout the book. This chapter provides an overview of a Python-based machine learning framework. First, we will review the details of Python-based packages used for machine learning, followed by the model development steps in the Python framework.

The steps of model development in Python presented in this chapter serve as the foundation for the case studies presented in the rest of the book. The Python framework can also be leveraged while developing any machine learning-based model in finance.

Why Python?

Some reasons for Python's popularity are as follows:

- High-level syntax (compared to lower-level languages of C, Java, and C++). Applications can be developed by writing fewer lines of code, making Python attractive to beginners and advanced programmers alike.
- Efficient development lifecycle.
- Large collection of community-managed, open-source libraries.
- Strong portability.

The simplicity of Python has attracted many developers to create new libraries for machine learning, leading to strong adoption of Python.

Python Packages for Machine Learning

The main Python packages used for machine learning are highlighted in Figure 2-1.

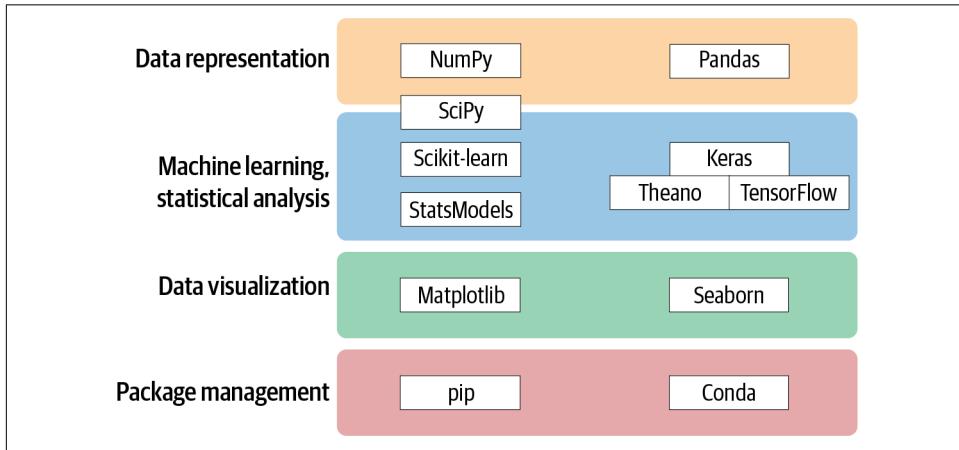


Figure 2-1. Python packages

Here is a brief summary of each of these packages:

NumPy

Provides support for large, multidimensional arrays as well as an extensive collection of mathematical functions.

Pandas

A library for data manipulation and analysis. Among other features, it offers data structures to handle tables and the tools to manipulate them.

Matplotlib

A plotting library that allows the creation of 2D charts and plots.

SciPy

The combination of NumPy, Pandas, and Matplotlib is generally referred to as SciPy. SciPy is an ecosystem of Python libraries for mathematics, science, and engineering.

Scikit-learn (or sklearn)

A machine learning library offering a wide range of algorithms and utilities.

StatsModels

A Python module that provides classes and functions for the estimation of many different statistical models, as well as for conducting statistical tests and statistical data exploration.

TensorFlow and Theano

Dataflow programming libraries that facilitate working with neural networks.

Keras

An artificial neural network library that can act as a simplified interface to TensorFlow/Theano packages.

Seaborn

A data visualization library based on Matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.

pip and Conda

These are Python package managers. pip is a package manager that facilitates installation, upgrade, and uninstallation of Python packages. Conda is a package manager that handles Python packages as well as library dependencies outside of the Python packages.

Python and Package Installation

There are different ways of installing Python. However, it is strongly recommended that you install Python through [Anaconda](#). Anaconda contains Python, SciPy, and Scikit-learn.

After installing Anaconda, a Jupyter server can be started locally by opening the machine's terminal and typing in the following code:

```
$jupyter notebook
```



All code samples in this book use Python 3 and are presented in Jupyter notebooks. Several Python packages, especially Scikit-learn and Keras, are extensively used in the case studies.

Steps for Model Development in Python Ecosystem

Working through machine learning problems from end to end is critically important. Applied machine learning will not come alive unless the steps from beginning to end are well defined.

[Figure 2-2](#) provides an outline of the simple seven-step machine learning project template that can be used to jump-start any machine learning model in Python. The

first few steps include exploratory data analysis and data preparation, which are typical data science-based steps aimed at extracting meaning and insights from data. These steps are followed by model evaluation, fine-tuning, and finalizing the model.

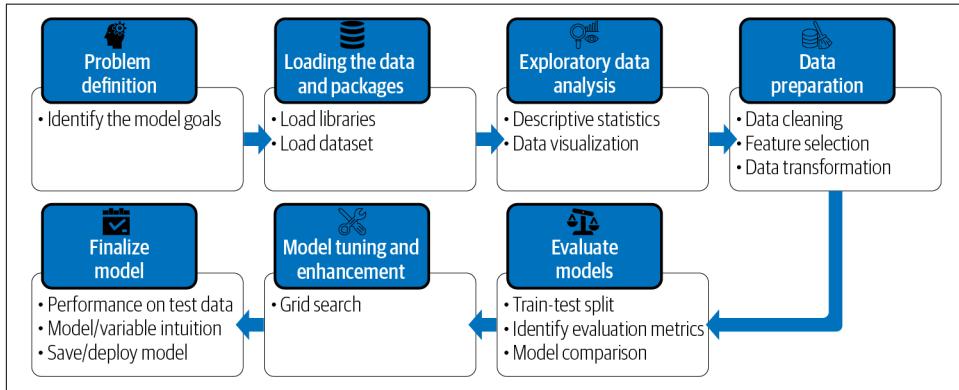


Figure 2-2. Model development steps



All the case studies in this book follow the standard seven-step model development process. However, there are a few case studies in which some of the steps are skipped, renamed, or reordered based on the appropriateness and intuitiveness of the steps.

Model Development Blueprint

The following section covers the details of each model development step with supporting Python code.

1. Problem definition

The first step in any project is defining the problem. Powerful algorithms can be used for solving the problem, but the results will be meaningless if the wrong problem is solved.

The following framework should be used for defining the problem:

1. Describe the problem informally and formally. List assumptions and similar problems.
2. List the motivation for solving the problem, the benefits a solution provides, and how the solution will be used.
3. Describe how the problem would be solved using the domain knowledge.

2. Loading the data and packages

The second step gives you everything needed to start working on the problem. This includes loading libraries, packages, and individual functions needed for the model development.

2.1. Load libraries.

A sample code for loading libraries is as follows:

```
# Load libraries
import pandas as pd
from matplotlib import pyplot
```

The details of the libraries and modules for specific functionalities are defined further in the individual case studies.

2.2. Load data.

The following items should be checked and removed before loading the data:

- Column headers
- Comments or special characters
- Delimiter

There are many ways of loading data. Some of the most common ways are as follows:

Load CSV files with Pandas

```
from pandas import read_csv
filename = 'xyz.csv'
data = read_csv(filename, names=names)
```

Load file from URL

```
from pandas import read_csv
url = 'https://goo.gl/vhm1eu'
names = ['age', 'class']
data = read_csv(url, names=names)
```

Load file using pandas_datareader

```
import pandas_datareader.data as web

ccy_tickers = ['DEXJPUS', 'DEXUSUK']
idx_tickers = ['SP500', 'DJIA', 'VIXCLS']

stk_data = web.DataReader(stk_tickers, 'yahoo')
ccy_data = web.DataReader(ccy_tickers, 'fred')
idx_data = web.DataReader(idx_tickers, 'fred')
```

3. Exploratory data analysis

In this step, we look at the dataset.

3.1. Descriptive statistics. Understanding the dataset is one of the most important steps of model development. The steps to understanding data include:

1. Viewing the raw data.
2. Reviewing the dimensions of the dataset.
3. Reviewing the data types of attributes.
4. Summarizing the distribution, descriptive statistics, and relationship among the variables in the dataset.

These steps are demonstrated below using sample Python code:

Viewing the data

```
set_option('display.width', 100)
dataset.head(1)
```

Output

	Age	Sex	Job	Housing	SavingAccounts	CheckingAccount	CreditAmount	Duration	Purpose	Risk
0	67	male	2	own	NaN	little	1169	6	radio/TV	good

Reviewing the dimensions of the dataset

```
dataset.shape
```

Output

```
(284807, 31)
```

The results show the dimension of the dataset and mean that the dataset has 284,807 rows and 31 columns.

Reviewing the data types of the attributes in the data

```
# types
set_option('display.max_rows', 500)
dataset.dtypes
```

Summarizing the data using descriptive statistics

```
# describe data
set_option('precision', 3)
dataset.describe()
```

Output

	Age	Job	CreditAmount	Duration
count	1000.000	1000.000	1000.000	1000.000
mean	35.546	1.904	3271.258	20.903
std	11.375	0.654	2822.737	12.059
min	19.000	0.000	250.000	4.000
25%	27.000	2.000	1365.500	12.000
50%	33.000	2.000	2319.500	18.000
75%	42.000	2.000	3972.250	24.000
max	75.000	3.000	18424.000	72.000

3.2. Data visualization. The fastest way to learn more about the data is to visualize it. Visualization involves independently understanding each attribute of the dataset.

Some of the plot types are as follows:

Univariate plots

Histograms and density plots

Multivariate plots

Correlation matrix plot and scatterplot

The Python code for univariate plot types is illustrated with examples below:

Univariate plot: histogram

```
from matplotlib import pyplot
dataset.hist(sharex=False, sharey=False, xlabelsize=1, ylabelsize=1,
            figsize=(10,4))
pyplot.show()
```

Univariate plot: density plot

```
from matplotlib import pyplot
dataset.plot(kind='density', subplots=True, layout=(3,3), sharex=False,
             legend=True, fontsize=1, figsize=(10,4))
pyplot.show()
```

Figure 2-3 illustrates the output.

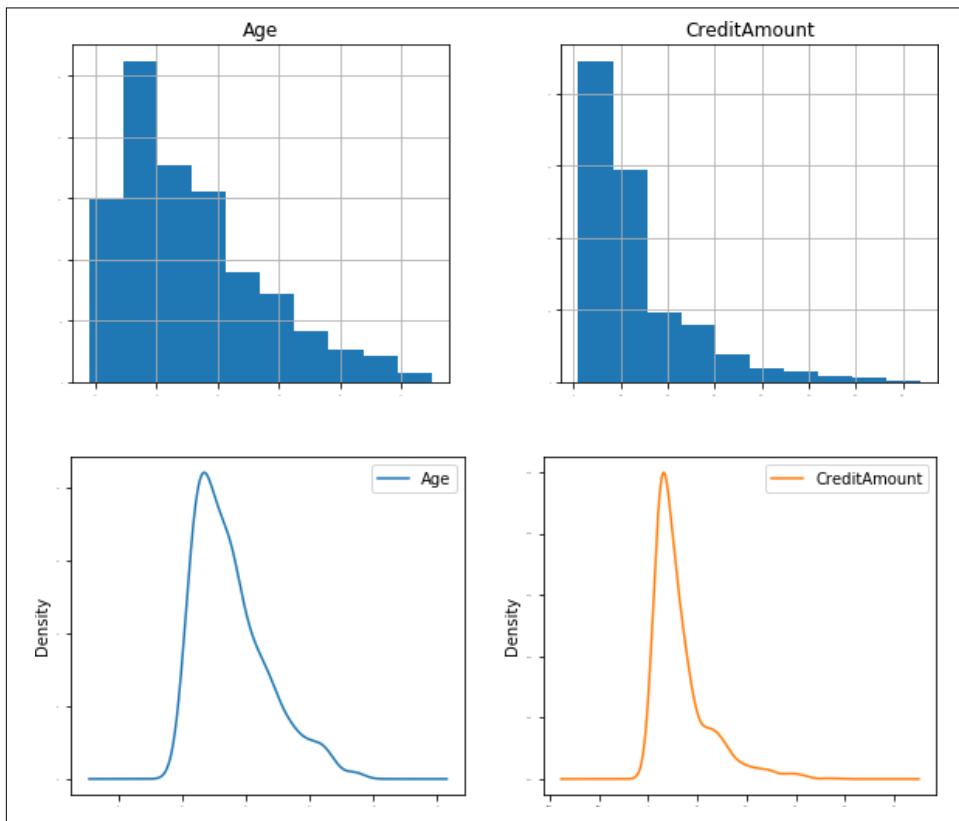


Figure 2-3. Histogram (top) and density plot (bottom)

The Python code for multivariate plot types is illustrated with examples below:

Multivariate plot: correlation matrix plot

```
from matplotlib import pyplot
import seaborn as sns
correlation = dataset.corr()
pyplot.figure(figsize=(5,5))
pyplot.title('Correlation Matrix')
sns.heatmap(correlation, vmax=1, square=True, annot=True, cmap='cubehelix')
```

Multivariate plot: scatterplot matrix

```
from pandas.plotting import scatter_matrix
scatter_matrix(dataset)
```

Figure 2-4 illustrates the output.

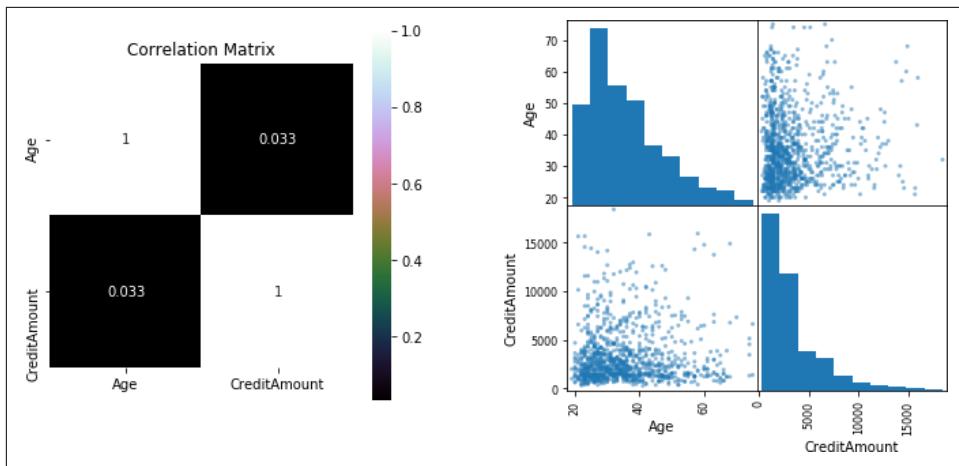


Figure 2-4. Correlation (left) and scatterplot (right)

4. Data preparation

Data preparation is a preprocessing step in which data from one or more sources is cleaned and transformed to improve its quality prior to its use.

4.1. Data cleaning. In machine learning modeling, incorrect data can be costly. Data cleaning involves checking the following:

Validity

The data type, range, etc.

Accuracy

The degree to which the data is close to the true values.

Completeness

The degree to which all required data is known.

Uniformity

The degree to which the data is specified using the same unit of measure.

The different options for performing data cleaning include:

Dropping “NA” values within data

```
dataset.dropna(axis=0)
```

Filling “NA” with 0

```
dataset.fillna(0)
```

Filling NAs with the mean of the column

```
dataset['col'] = dataset['col'].fillna(dataset['col'].mean())
```

4.2. Feature selection. The data features used to train the machine learning models have a huge influence on the performance. Irrelevant or partially relevant features can negatively impact model performance. Feature selection¹ is a process in which features in data that contribute most to the prediction variable or output are automatically selected.

The benefits of performing feature selection before modeling the data are:

*Reduces overfitting*²

Less redundant data means fewer opportunities for the model to make decisions based on noise.

Improves performance

Less misleading data means improved modeling performance.

Reduces training time and memory footprint

Less data means faster training and lower memory footprint.

The following sample feature is an example demonstrating when the best two features are selected using the **SelectKBest** function under sklearn. The SelectKBest function scores the features using an underlying function and then removes all but the k highest scoring feature:

```
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2
bestfeatures = SelectKBest( k=5)
fit = bestfeatures.fit(X,Y)
dfscores = pd.DataFrame(fit.scores_)
dfcolumns = pd.DataFrame(X.columns)
featureScores = pd.concat([dfcolumns,dfscores],axis=1)
print(featureScores.nlargest(2,'Score')) #print 2 best features
```

Output

	Specs	Score
2	Variable1	58262.490
3	Variable2	321.031

When features are irrelevant, they should be dropped. Dropping the irrelevant features is illustrated in the following sample code:

```
#dropping the old features
dataset.drop(['Feature1','Feature2','Feature3'],axis=1,inplace=True)
```

¹ Feature selection is more relevant for supervised learning models and is described in detail in the individual case studies in Chapters 5 and 6.

² Overfitting is covered in detail in Chapter 4.

4.3. Data transformation. Many machine learning algorithms make assumptions about the data. It is a good practice to perform the data preparation in such a way that exposes the data in the best possible manner to the machine learning algorithms. This can be accomplished through data transformation.

The different data transformation approaches are as follows:

Rescaling

When data comprises attributes with varying scales, many machine learning algorithms can benefit from *rescaling* all the attributes to the same scale. Attributes are often rescaled in the range between zero and one. This is useful for optimization algorithms used in the core of machine learning algorithms, and it also helps to speed up the calculations in an algorithm:

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler(feature_range=(0, 1))
rescaledX = pd.DataFrame(scaler.fit_transform(X))
```

Standardization

Standardization is a useful technique to transform attributes to a standard **normal distribution** with a mean of zero and a standard deviation of one. It is most suitable for techniques that assume the input variables represent a normal distribution:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler().fit(X)
StandardisedX = pd.DataFrame(scaler.fit_transform(X))
```

Normalization

Normalization refers to rescaling each observation (row) to have a length of one (called a unit norm or a vector). This preprocessing method can be useful for sparse datasets of attributes of varying scales when using algorithms that weight input values:

```
from sklearn.preprocessing import Normalizer
scaler = Normalizer().fit(X)
NormalizedX = pd.DataFrame(scaler.fit_transform(X))
```

5. Evaluate models

Once we estimate the performance of our algorithm, we can retrain the final algorithm on the entire training dataset and get it ready for operational use. The best way to do this is to evaluate the performance of the algorithm on a new dataset. Different machine learning techniques require different evaluation metrics. Other than model performance, several other factors such as simplicity, interpretability, and training time are considered when selecting a model. The details regarding these factors are covered in [Chapter 4](#).

5.1. Training and test split. The simplest method we can use to evaluate the performance of a machine learning algorithm is to use different training and testing datasets. We can take our original dataset and split it into two parts: train the algorithm on the first part, make predictions on the second part, and evaluate the predictions against the expected results. The size of the split can depend on the size and specifics of the dataset, although it is common to use 80% of the data for training and the remaining 20% for testing. The differences in the training and test datasets can result in meaningful differences in the estimate of accuracy. The data can easily be split into the training and test sets using the `train_test_split` function available in `sklearn`:

```
# split out validation dataset for the end
validation_size = 0.2
seed = 7
X_train, X_validation, Y_train, Y_validation =\
train_test_split(X, Y, test_size=validation_size, random_state=seed)
```

5.2. Identify evaluation metrics. Choosing which metric to use to evaluate machine learning algorithms is very important. An important aspect of evaluation metrics is the capability to discriminate among model results. Different types of evaluation metrics used for different kinds of ML models are covered in detail across several chapters of this book.

5.3. Compare models and algorithms. Selecting a machine learning model or algorithm is both an art and a science. There is no one solution or approach that fits all. There are several factors over and above the model performance that can impact the decision to choose a machine learning algorithm.

Let's understand the process of model comparison with a simple example. We define two variables, X and Y , and try to build a model to predict Y using X . As a first step, the data is divided into training and test split as mentioned in the preceding section:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
validation_size = 0.2
seed = 7
X = 2 - 3 * np.random.normal(0, 1, 20)
Y = X - 2 * (X ** 2) + 0.5 * (X ** 3) + np.exp(-X)+np.random.normal(-3, 3, 20)
# transforming the data to include another axis
X = X[:, np.newaxis]
Y = Y[:, np.newaxis]
X_train, X_test, Y_train, Y_test = train_test_split(X, Y,\n    test_size=validation_size, random_state=seed)
```

We have no idea which algorithms will do well on this problem. Let's design our test now. We will use two models—one linear regression and the second polynomial regression to fit Y against X . We will evaluate algorithms using the *Root Mean*

Squared Error (RMSE) metric, which is one of the measures of the model performance. RMSE will give a gross idea of how wrong all predictions are (zero is perfect):

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.preprocessing import PolynomialFeatures

model = LinearRegression()
model.fit(X_train, Y_train)
Y_pred = model.predict(X_train)

rmse_lin = np.sqrt(mean_squared_error(Y_train,Y_pred))
r2_lin = r2_score(Y_train,Y_pred)
print("RMSE for Linear Regression:", rmse_lin)

polynomial_features= PolynomialFeatures(degree=2)
x_poly = polynomial_features.fit_transform(X_train)

model = LinearRegression()
model.fit(x_poly, Y_train)
Y_poly_pred = model.predict(x_poly)

rmse = np.sqrt(mean_squared_error(Y_train,Y_poly_pred))
r2 = r2_score(Y_train,Y_poly_pred)
print("RMSE for Polynomial Regression:", rmse)
```

Output

```
RMSE for Linear Regression: 6.772942423315028
RMSE for Polynomial Regression: 6.420495127266883
```

We can see that the RMSE of the polynomial regression is slightly better than that of the linear regression.³ With the former having the better fit, it is the preferred model in this step.

6. Model tuning

Finding the best combination of hyperparameters of a model can be treated as a search problem.⁴ This searching exercise is often known as *model tuning* and is one of the most important steps of model development. It is achieved by searching for the best parameters of the model by using techniques such as a *grid search*. In a grid search, you create a grid of all possible hyperparameter combinations and train the model using each one of them. Besides a grid search, there are several other

³ It should be noted that the difference in RMSE is small in this case and may not replicate with a different split of the train/test data.

⁴ Hyperparameters are the external characteristics of the model, can be considered the model's settings, and are not estimated based on data-like model parameters.

techniques for model tuning, including randomized search, [Bayesian optimization](#), and hyperbrand.

In the case studies presented in this book, we focus primarily on grid search for model tuning.

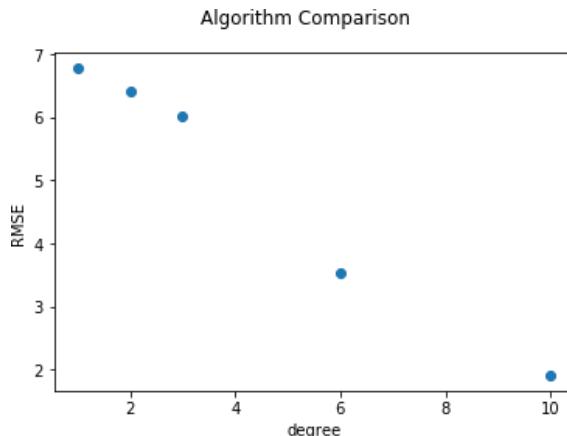
Continuing on from the preceding example, with the polynomial as the best model: next, run a grid search for the model, refitting the polynomial regression with different degrees. We compare the RMSE results for all the models:

```
Deg= [1,2,3,6,10]
results=[]
names=[]
for deg in Deg:
    polynomial_features= PolynomialFeatures(degree=deg)
    x_poly = polynomial_features.fit_transform(X_train)

    model = LinearRegression()
    model.fit(x_poly, Y_train)
    Y_poly_pred = model.predict(x_poly)

    rmse = np.sqrt(mean_squared_error(Y_train,Y_poly_pred))
    r2 = r2_score(Y_train,Y_poly_pred)
    results.append(rmse)
    names.append(deg)
plt.plot(names, results,'o')
plt.suptitle('Algorithm Comparison')
```

Output



The RMSE decreases when the degree increases, and the lowest RMSE is for the model with degree 10. However, models with degrees lower than 10 performed very well, and the test set will be used to finalize the best model.

While the generic set of input parameters for each algorithm provides a starting point for analysis, it may not have the optimal configurations for the particular dataset and business problem.

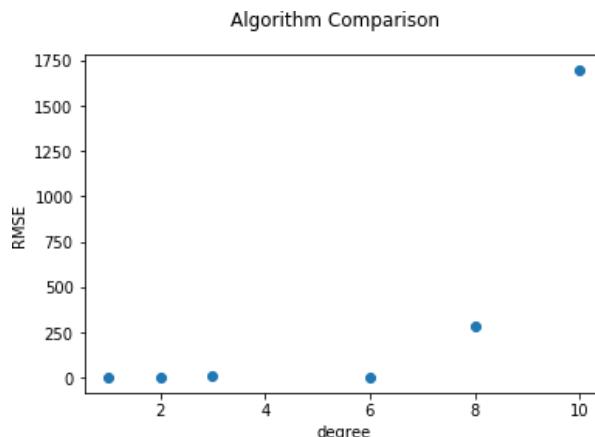
7. Finalize the model

Here, we perform the final steps for selecting the model. First, we run predictions on the test dataset with the trained model. Then we try to understand the model intuition and save it for further usage.

7.1. Performance on the test set. The model selected during the training steps is further evaluated on the test set. The test set allows us to compare different models in an unbiased way, by basing the comparisons in data that were not used in any part of the training. The test results for the model developed in the previous step are shown in the following example:

```
Deg= [1,2,3,6,8,10]
for deg in Deg:
    polynomial_features= PolynomialFeatures(degree=deg)
    x_poly = polynomial_features.fit_transform(X_train)
    model = LinearRegression()
    model.fit(x_poly, Y_train)
    x_poly_test = polynomial_features.fit_transform(X_test)
    Y_poly_pred_test = model.predict(x_poly_test)
    rmse = np.sqrt(mean_squared_error(Y_test,Y_poly_pred_test))
    r2 = r2_score(Y_test,Y_poly_pred_test)
    results_test.append(rmse)
    names_test.append(deg)
plt.plot(names_test, results_test,'o')
plt.suptitle('Algorithm Comparison')
```

Output



In the training set we saw that the RMSE decreases with an increase in the degree of polynomial model, and the polynomial of degree 10 had the lowest RMSE. However, as shown in the preceding output for the polynomial of degree 10, although the training set had the best results, the results in the test set are poor. For the polynomial of degree 8, the RMSE in the test set is relatively higher. The polynomial of degree 6 shows the best result in the test set (although the difference is small compared to other lower-degree polynomials in the test set) as well as good results in the training set. For these reasons, this is the preferred model.

In addition to the model performance, there are several other factors to consider when selecting a model, such as simplicity, interpretability, and training time. These factors will be covered in the upcoming chapters.

7.2. Model/variable intuition. This step involves considering a holistic view of the approach taken to solve the problem, including the model's limitations as it relates to the desired outcome, the variables used, and the selected model parameters. Details on model and variable intuition regarding different types of machine learning models are presented in the subsequent chapters and case studies.

7.3. Save/deploy. After finding an accurate machine learning model, it must be saved and loaded in order to ensure its usage later.

Pickle is one of the packages for saving and loading a trained model in Python. Using pickle operations, trained machine learning models can be saved in the *serialized* format to a file. Later, this serialized file can be loaded to *de-serialize* the model for its usage. The following sample code demonstrates how to save the model to a file and load it to make predictions on new data:

```
# Save Model Using Pickle
from pickle import dump
from pickle import load
# save the model to disk
filename = 'finalized_model.sav'
dump(model, open(filename, 'wb'))
# load the model from disk
loaded_model = load(open(filename))
```



In recent years, frameworks such as *AutoML* have been built to automate the maximum number of steps in a machine learning model development process. Such frameworks allow the model developers to build ML models with high scale, efficiency, and productivity. Readers are encouraged to explore such frameworks.

Chapter Summary

Given its popularity, rate of adoption, and flexibility, Python is often the preferred language for machine learning development. There are many available Python packages to perform numerous tasks, including data cleaning, visualization, and model development. Some of these key packages are Scikit-learn and Keras.

The seven steps of model development mentioned in this chapter can be leveraged while developing any machine learning–based model in finance.

Next Steps

In the next chapter, we will cover the key algorithm for machine learning—the artificial neural network. The artificial neural network is another building block of machine learning in finance and is used across all types of machine learning and deep learning algorithms.

Artificial Neural Networks

There are many different types of models used in machine learning. However, one class of machine learning models that stands out is artificial neural networks (ANNs). Given that artificial neural networks are used across all machine learning types, this chapter will cover the basics of ANNs.

ANNs are computing systems based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain. Each connection, like the synapses in a biological brain, can transmit a signal from one artificial neuron to another. An artificial neuron that receives a signal can process it and then signal additional artificial neurons connected to it.

Deep learning involves the study of complex ANN-related algorithms. The complexity is attributed to elaborate patterns of how information flows throughout the model. Deep learning has the ability to represent the world as a nested hierarchy of concepts, with each concept defined in relation to a simpler concept. Deep learning techniques are extensively used in reinforcement learning and natural language processing applications that we will look at in Chapters 9 and 10.

We will review detailed terminology and processes used in the field of ANNs¹ and cover the following topics:

- Architecture of ANNs: Neurons and layers
- Training an ANN: Forward propagation, backpropagation and gradient descent
- Hyperparameters of ANNs: Number of layers and nodes, activation function, loss function, learning rate, etc.
- Defining and training a deep neural network-based model in Python
- Improving the training speed of ANNs and deep learning models

ANNs: Architecture, Training, and Hyperparameters

ANNs contain multiple neurons arranged in layers. An ANN goes through a training phase by comparing the modeled output to the desired output, where it learns to recognize patterns in data. Let us go through the components of ANNs.

Architecture

An ANN architecture comprises neurons, layers, and weights.

Neurons

The building blocks for ANNs are neurons (also known as artificial neurons, nodes, or perceptrons). Neurons have one or more inputs and one output. It is possible to build a network of neurons to compute complex logical propositions. Activation functions in these neurons create complicated, nonlinear functional mappings between the inputs and the output.²

As shown in [Figure 3-1](#), a neuron takes an input (x_1, x_2, \dots, x_n), applies the learning parameters to generate a weighted sum (z), and then passes that sum to an activation function (f) that computes the output $f(z)$.

¹ Readers are encouraged to refer to the book *Deep Learning* by Aaron Courville, Ian Goodfellow, and Yoshua Bengio (MIT Press) for more details on ANN and deep learning.

² Activation functions are described in detail later in this chapter.

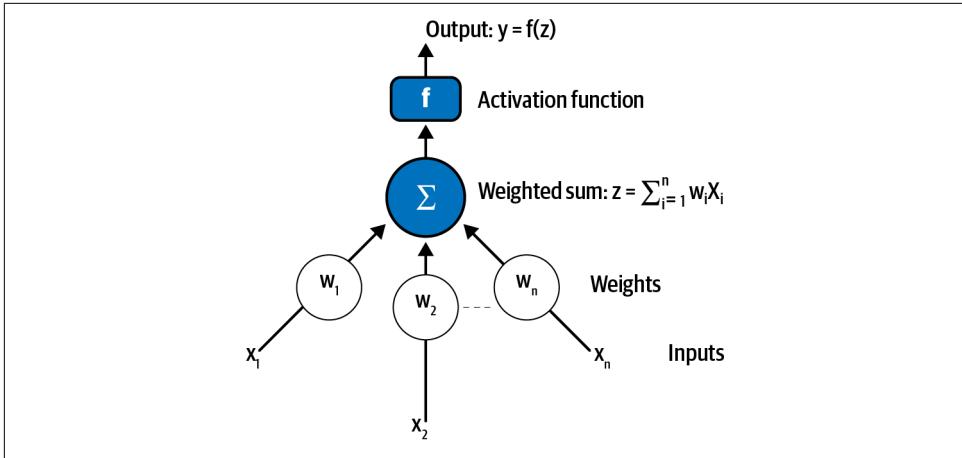


Figure 3-1. An artificial neuron

Layers

The output $f(z)$ from a single neuron (as shown in Figure 3-1) will not be able to model complex tasks. So, in order to handle more complex structures, we have multiple layers of such neurons. As we keep stacking neurons horizontally and vertically, the class of functions we can get becomes increasing complex. Figure 3-2 shows an architecture of an ANN with an input layer, an output layer, and a hidden layer.

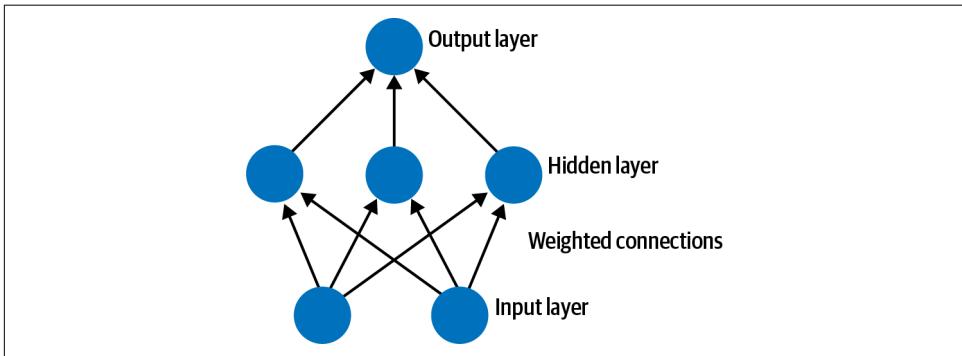


Figure 3-2. Neural network architecture

Input layer. The input layer takes input from the dataset and is the exposed part of the network. A neural network is often drawn with an input layer of one neuron per input value (or column) in the dataset. The neurons in the input layer simply pass the input value through to the next layer.

Hidden layers. Layers after the input layer are called hidden layers because they are not directly exposed to the input. The simplest network structure is to have a single neuron in the hidden layer that directly outputs the value.

A multilayer ANN is capable of solving more complex machine learning-related tasks due to its hidden layer(s). Given increases in computing power and efficient libraries, neural networks with many layers can be constructed. ANNs with many hidden layers (more than three) are known as *deep neural networks*. Multiple hidden layers allow deep neural networks to learn features of the data in a so-called feature hierarchy, because simple features recombine from one layer to the next to form more complex features. ANNs with many layers pass input data (features) through more mathematical operations than do ANNs with few layers and are therefore more computationally intensive to train.

Output layer. The final layer is called the output layer; it is responsible for outputting a value or vector of values that correspond to the format required to solve the problem.

Neuron weights

A neuron weight represents the strength of the connection between units and measures the influence the input will have on the output. If the weight from neuron one to neuron two has greater magnitude, it means that neuron one has a greater influence over neuron two. Weights near zero mean changing this input will not change the output. Negative weights mean increasing this input will decrease the output.

Training

Training a neural network basically means calibrating all of the weights in the ANN. This optimization is performed using an iterative approach involving forward propagation and backpropagation steps.

Forward propagation

Forward propagation is a process of feeding input values to the neural network and getting an output, which we call *predicted value*. When we feed the input values to the neural network's first layer, it goes without any operations. The second layer takes values from the first layer and applies multiplication, addition, and activation operations before passing this value to the next layer. The same process repeats for any subsequent layers until an output value from the last layer is received.

Backpropagation

After forward propagation, we get a predicted value from the ANN. Suppose the desired output of a network is Y and the predicted value of the network from forward propagation is Y' . The difference between the predicted output and the desired output ($Y - Y'$) is converted into the loss (or cost) function $J(w)$, where w represents the weights in ANN.³ The goal is to optimize the loss function (i.e., make the loss as small as possible) over the training set.

The optimization method used is *gradient descent*. The goal of the gradient descent method is to find the gradient of $J(w)$ with respect to w at the current point and take a small step in the direction of the negative gradient until the minimum value is reached, as shown in [Figure 3-3](#).

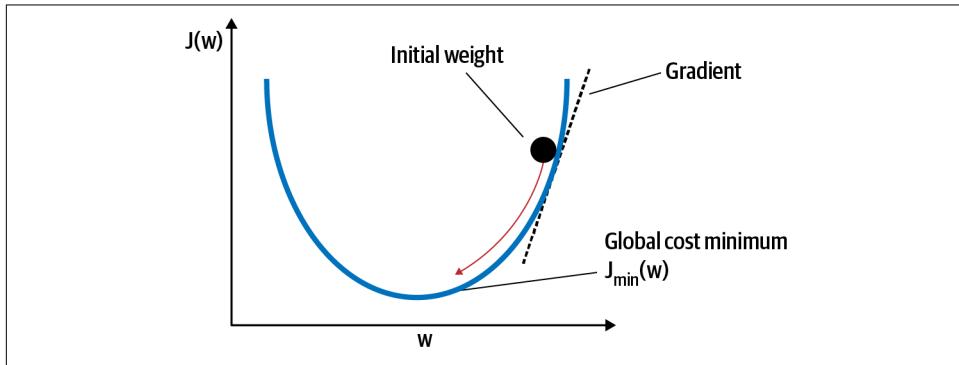


Figure 3-3. Gradient descent

In an ANN, the function $J(w)$ is essentially a composition of multiple layers, as explained in the preceding text. So, if layer one is represented as function $p()$, layer two as $q()$, and layer three as $r()$, then the overall function is $J(w)=r(q(p()))$. w consists of all weights in all three layers. We want to find the gradient of $J(w)$ with respect to each component of w .

Skipping the mathematical details, the above essentially implies that the gradient of a component w in the first layer would depend on the gradients in the second and third layers. Similarly, the gradients in the second layer will depend on the gradients in the third layer. Therefore, we start computing the derivatives in the reverse direction, starting with the last layer, and use backpropagation to compute gradients of the previous layer.

³ There are many available loss functions discussed in the next section. The nature of our problem dictates our choice of loss function.

Overall, in the process of backpropagation, the model error (difference between predicted and desired output) is propagated back through the network, one layer at a time, and the weights are updated according to the amount they contributed to the error.

Almost all ANNs use gradient descent and backpropagation. Backpropagation is one of the cleanest and most efficient ways to find the gradient.

Hyperparameters

Hyperparameters are the variables that are set before the training process, and they cannot be learned during training. ANNs have a large number of hyperparameters, which makes them quite flexible. However, this flexibility makes the model tuning process difficult. Understanding the hyperparameters and the intuition behind them helps give an idea of what values are reasonable for each hyperparameter so we can restrict the search space. Let's start with the number of hidden layers and nodes.

Number of hidden layers and nodes

More hidden layers or nodes per layer means more parameters in the ANN, allowing the model to fit more complex functions. To have a trained network that generalizes well, we need to pick an optimal number of hidden layers, as well as of the nodes in each hidden layer. Too few nodes and layers will lead to high errors for the system, as the predictive factors might be too complex for a small number of nodes to capture. Too many nodes and layers will overfit to the training data and not generalize well.

There is no hard-and-fast rule to decide the number of layers and nodes.

The number of hidden layers primarily depends on the complexity of the task. Very complex tasks, such as large image classification or speech recognition, typically require networks with dozens of layers and a huge amount of training data. For the majority of the problems, we can start with just one or two hidden layers and then gradually ramp up the number of hidden layers until we start overfitting the training set.

The number of hidden nodes should have a relationship to the number of input and output nodes, the amount of training data available, and the complexity of the function being modeled. As a rule of thumb, the number of hidden nodes in each layer should be somewhere between the size of the input layer and the size of the output layer, ideally the mean. The number of hidden nodes shouldn't exceed twice the number of input nodes in order to avoid overfitting.

Learning rate

When we train ANNs, we use many iterations of forward propagation and backpropagation to optimize the weights. At each iteration we calculate the derivative of the

loss function with respect to each weight and subtract it from that weight. The learning rate determines how quickly or slowly we want to update our weight (parameter) values. This learning rate should be high enough so that it converges in a reasonable amount of time. Yet it should be low enough so that it finds the minimum value of the loss function.

Activation functions

Activation functions (as shown in [Figure 3-1](#)) refer to the functions used over the weighted sum of inputs in ANNs to get the desired output. Activation functions allow the network to combine the inputs in more complex ways, and they provide a richer capability in the relationship they can model and the output they can produce. They decide which neurons will be activated—that is, what information is passed to further layers.

Without activation functions, ANNs lose a bulk of their representation learning power. There are several activation functions. The most widely used are as follows:

Linear (identity) function

Represented by the equation of a straight line (i.e., $f(x) = mx + c$), where activation is proportional to the input. If we have many layers, and all the layers are linear in nature, then the final activation function of the last layer is the same as the linear function of the first layer. The range of a linear function is $-inf$ to $+inf$.

Sigmoid function

Refers to a function that is projected as an S-shaped graph (as shown in [Figure 3-4](#)). It is represented by the mathematical equation $f(x) = 1 / (1 + e^{-x})$ and ranges from 0 to 1. A large positive input results in a large positive output; a large negative input results in a large negative output. It is also referred to as logistic activation function.

Tanh function

Similar to sigmoid activation function with a mathematical equation $Tanh(x) = 2Sigmoid(2x) - 1$, where *Sigmoid* represents the sigmoid function discussed above. The output of this function ranges from -1 to 1 , with an equal mass on both sides of the zero-axis, as shown in [Figure 3-4](#).

ReLU function

ReLU stands for the Rectified Linear Unit and is represented as $f(x) = max(x, 0)$. So, if the input is a positive number, the function returns the number itself, and if the input is a negative number, then the function returns zero. It is the most commonly used function because of its simplicity.

Figure 3-4 shows a summary of the activation functions discussed in this section.

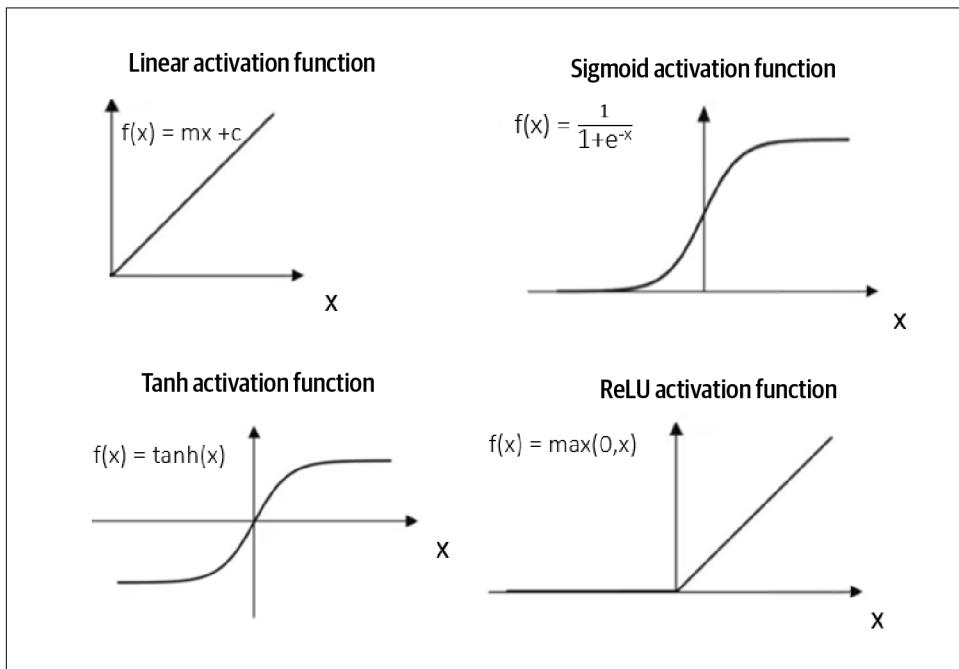


Figure 3-4. Activation functions

There is no hard-and-fast rule for activation function selection. The decision completely relies on the properties of the problem and the relationships being modeled. We can try different activation functions and select the one that helps provide faster convergence and a more efficient training process. The choice of activation function in the output layer is strongly constrained by the type of problem that is modeled.⁴

Cost functions

Cost functions (also known as loss functions) are a measure of the ANN performance, measuring how well the ANN fits empirical data. The two most common cost functions are:

Mean squared error (MSE)

This is the cost function used primarily for regression problems, where output is a continuous value. MSE is measured as the average of the squared difference

⁴ Deriving a regression or classification output by changing the activation function of the output layer is described further in [Chapter 4](#).

between predictions and actual observation. MSE is described further in [Chapter 4](#).

Cross-entropy (or log loss)

This cost function is used primarily for classification problems, where output is a probability value between zero and one. Cross-entropy loss increases as the predicted probability diverges from the actual label. A perfect model would have a cross-entropy of zero.

Optimizers

Optimizers update the weight parameters to minimize the loss function.⁵ Cost function acts as a guide to the terrain, telling the optimizer if it is moving in the right direction to reach the global minimum. Some of the common optimizers are as follows:

Momentum

The *momentum optimizer* looks at previous gradients in addition to the current step. It will take larger steps if the previous updates and the current update move the weights in the same direction (gaining momentum). It will take smaller steps if the direction of the gradient is opposite. A clever way to visualize this is to think of a ball rolling down a valley—it will gain momentum as it approaches the valley bottom.

AdaGrad (Adaptive Gradient Algorithm)

AdaGrad adapts the learning rate to the parameters, performing smaller updates for parameters associated with frequently occurring features, and larger updates for parameters associated with infrequent features.

RMSProp

RMSProp stands for Root Mean Square Propagation. In RMSProp, the learning rate gets adjusted automatically, and it chooses a different learning rate for each parameter.

Adam (Adaptive Moment Estimation)

Adam combines the best properties of the AdaGrad and RMSProp algorithms to provide an optimization and is one of the most popular gradient descent optimization algorithms.

⁵ Refer to <https://oreil.ly/FSt-8> for more details on optimization.

Epoch

One round of updating the network for the entire training dataset is called an *epoch*. A network may be trained for tens, hundreds, or many thousands of epochs depending on the data size and computational constraints.

Batch size

The batch size is the number of training examples in one forward/backward pass. A batch size of 32 means that 32 samples from the training dataset will be used to estimate the error gradient before the model weights are updated. The higher the batch size, the more memory space is needed.

Creating an Artificial Neural Network Model in Python

In [Chapter 2](#) we discussed the steps for end-to-end model development in Python. In this section, we dig deeper into the steps involved in building an ANN-based model in Python.

Our first step will be to look at Keras, the Python package specifically built for ANN and deep learning.

Installing Keras and Machine Learning Packages

There are several Python libraries that allow building ANN and deep learning models easily and quickly without getting into the details of underlying algorithms. Keras is one of the most user-friendly packages that enables an efficient numerical computation related to ANNs. Using Keras, complex deep learning models can be defined and implemented in a few lines of code. We will primarily be using Keras packages for implementing deep learning models in several of the book's case studies.

Keras is simply a wrapper around more complex numerical computation engines such as [TensorFlow](#) and [Theano](#). In order to install Keras, TensorFlow or Theano needs to be installed first.

This section describes the steps to define and compile an ANN-based model in Keras, with a focus on the following steps.⁶

Importing the packages

Before you can start to build an ANN model, you need to import two modules from the Keras package: **Sequential** and **Dense**:

⁶ The steps and Python code related to implementing deep learning models using Keras, as demonstrated in this section, are used in several case studies in the subsequent chapters.

```
from Keras.models import Sequential
from Keras.layers import Dense
import numpy as np
```

Loading data

This example makes use of the `random` module of NumPy to quickly generate some data and labels to be used by ANN that we build in the next step. Specifically, an array with size $(1000, 10)$ is first constructed. Next, we create a labels array that consists of zeros and ones with a size $(1000, 1)$:

```
data = np.random.random((1000,10))
Y = np.random.randint(2,size= (1000,1))
model = Sequential()
```

Model construction—defining the neural network architecture

A quick way to get started is to use the Keras Sequential model, which is a linear stack of layers. We create a Sequential model and add layers one at a time until the network topology is finalized. The first thing to get right is to ensure the input layer has the right number of inputs. We can specify this when creating the first layer. We then select a dense or fully connected layer to indicate that we are dealing with an input layer by using the argument `input_dim`.

We add a layer to the model with the `add()` function, and the number of nodes in each layer is specified. Finally, another dense layer is added as an output layer.

The architecture for the model shown in [Figure 3-5](#) is as follows:

- The model expects rows of data with 10 variables (`input_dim=10` argument).
- The first hidden layer has 32 nodes and uses the `relu` activation function.
- The second hidden layer has 32 nodes and uses the `relu` activation function.
- The output layer has one node and uses the `sigmoid` activation function.

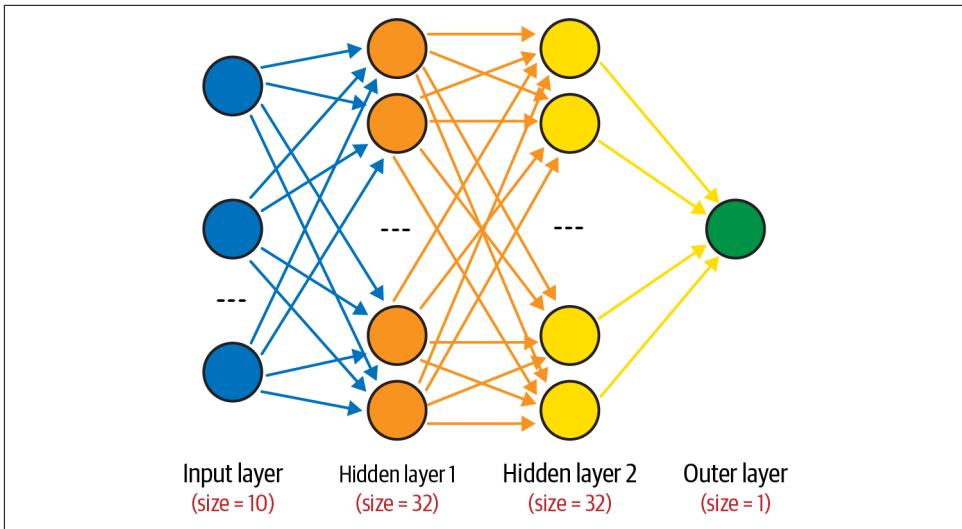


Figure 3-5. An ANN architecture

The Python code for the network in [Figure 3-5](#) is shown below:

```
model = Sequential()
model.add(Dense(32, input_dim=10, activation= 'relu' ))
model.add(Dense(32, activation= 'relu' ))
model.add(Dense(1, activation= 'sigmoid'))
```

Compiling the model

With the model constructed, it can be compiled with the help of the `compile()` function. Compiling the model leverages the efficient numerical libraries in the Theano or TensorFlow packages. When compiling, it is important to specify the additional properties required when training the network. Training a network means finding the best set of weights to make predictions for the problem at hand. So we must specify the loss function used to evaluate a set of weights, the optimizer used to search through different weights for the network, and any optional metrics we would like to collect and report during training.

In the following example, we use `cross-entropy` loss function, which is defined in Keras as `binary_crossentropy`. We will also use the `adam` optimizer, which is the default option. Finally, because it is a classification problem, we will collect and report the classification accuracy as the metric.⁷ The Python code follows:

```
model.compile(loss= 'binary_crossentropy' , optimizer= 'adam' , \
metrics=[ 'accuracy' ])
```

⁷ A detailed discussion of the evaluation metrics for classification models is presented in [Chapter 4](#).

Fitting the model

With our model defined and compiled, it is time to execute it on data. We can train or fit our model on our loaded data by calling the `fit()` function on the model.

The training process will run for a fixed number of iterations (epochs) through the dataset, specified using the `nb_epoch` argument. We can also set the number of instances that are evaluated before a weight update in the network is performed. This is set using the `batch_size` argument. For this problem we will run a small number of epochs (10) and use a relatively small batch size of 32. Again, these can be chosen experimentally through trial and error. The Python code follows:

```
model.fit(data, Y, nb_epoch=10, batch_size=32)
```

Evaluating the model

We have trained our neural network on the entire dataset and can evaluate the performance of the network on the same dataset. This will give us an idea of how well we have modeled the dataset (e.g., training accuracy) but will not provide insight on how well the algorithm will perform on new data. For this, we separate the data into training and test datasets. The model is evaluated on the training dataset using the `evaluation()` function. This will generate a prediction for each input and output pair and collect scores, including the average loss and any metrics configured, such as accuracy. The Python code follows:

```
scores = model.evaluate(X_test, Y_test)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

Running an ANN Model Faster: GPU and Cloud Services

For training ANNs (especially deep neural networks with many layers), a large amount of computation power is required. Available CPUs, or Central Processing Units, are responsible for processing and executing instructions on a local machine. Since CPUs are limited in the number of cores and take up the job sequentially, they cannot do rapid matrix computations for the large number of matrices required for training deep learning models. Hence, the training of deep learning models can be extremely slow on the CPUs.

The following alternatives are useful for running ANNs that generally require a significant amount of time to run on a CPU:

- Running notebooks locally on a GPU.
- Running notebooks on Kaggle Kernels or Google Colaboratory.
- Using Amazon Web Services.

GPU

A GPU is composed of hundreds of cores that can handle thousands of threads simultaneously. Running ANNs and deep learning models can be accelerated by the use of GPUs.

GPUs are particularly adept at processing complex matrix operations. The GPU cores are highly specialized, and they massively accelerate processes such as deep learning training by offloading the processing from CPUs to the cores in the GPU subsystem.

All the Python packages related to machine learning, including Tensorflow, Theano, and Keras, can be configured for the use of GPUs.

Cloud services such as Kaggle and Google Colab

If you have a GPU-enabled computer, you can run ANNs locally. If you do not, we recommend you use a service such as Kaggle Kernels, Google Colab, or AWS:

Kaggle

A popular data science website owned by Google that hosts Jupyter service and is also referred to as **Kaggle Kernels**. Kaggle Kernels are free to use and come with the most frequently used packages preinstalled. You can connect a kernel to any dataset hosted on Kaggle, or alternatively, you can just upload a new dataset on the fly.

Google Colaboratory

A free Jupyter Notebook environment provided by Google where you can use free GPUs. The features of **Google Colaboratory** are the same as Kaggle.

Amazon Web Services (AWS)

AWS Deep Learning provides an infrastructure to accelerate deep learning in the cloud, at any scale. You can quickly launch AWS server instances preinstalled with popular deep learning frameworks and interfaces to train sophisticated, custom AI models, experiment with new algorithms, or learn new skills and techniques. These web servers can run longer than Kaggle Kernels. So for big projects, it might be worth using an AWS instead of a kernel.

Chapter Summary

ANNs comprise a family of algorithms used across all types of machine learning. These models are inspired by the biological neural networks containing neurons and layers of neurons that constitute animal brains. ANNs with many layers are referred to as deep neural networks. Several steps, including forward propagation and back-propagation, are required for training these ANNs. Python packages such as Keras make the training of these ANNs possible in a few lines of code. The training of these deep neural networks require more computational power, and CPUs alone might not be enough. Alternatives include using a GPU or cloud service such as Kaggle Kernels, Google Colaboratory, or Amazon Web Services for training deep neural networks.

Next Steps

As a next step, we will be going into the details of the machine learning concepts for supervised learning, followed by case studies using the concepts covered in this chapter.

PART II

Supervised Learning

Supervised Learning: Models and Concepts

Supervised learning is an area of machine learning where the chosen algorithm tries to fit a target using the given input. A set of training data that contains labels is supplied to the algorithm. Based on a massive set of data, the algorithm will learn a rule that it uses to predict the labels for new observations. In other words, supervised learning algorithms are provided with historical data and asked to find the relationship that has the best predictive power.

There are two varieties of supervised learning algorithms: regression and classification algorithms. Regression-based supervised learning methods try to predict outputs based on input variables. Classification-based supervised learning methods identify which category a set of data items belongs to. Classification algorithms are probability-based, meaning the outcome is the category for which the algorithm finds the highest probability that the dataset belongs to it. Regression algorithms, in contrast, estimate the outcome of problems that have an infinite number of solutions (continuous set of possible outcomes).

In the context of finance, supervised learning models represent one of the most-used class of machine learning models. Many algorithms that are widely applied in algorithmic trading rely on supervised learning models because they can be efficiently trained, they are relatively robust to noisy financial data, and they have strong links to the theory of finance.

Regression-based algorithms have been leveraged by academic and industry researchers to develop numerous asset pricing models. These models are used to predict returns over various time periods and to identify significant factors that drive asset returns. There are many other use cases of regression-based supervised learning in portfolio management and derivatives pricing.

Classification-based algorithms, on the other hand, have been leveraged across many areas within finance that require predicting a categorical response. These include fraud detection, default prediction, credit scoring, directional forecast of asset price movement, and Buy/Sell recommendations. There are many other use cases of classification-based supervised learning in portfolio management and algorithmic trading.

Many use cases of regression-based and classification-based supervised machine learning are presented in Chapters 5 and 6.

Python and its libraries provide methods and ways to implement these supervised learning models in few lines of code. Some of these libraries were covered in Chapter 2. With easy-to-use machine learning libraries like Scikit-learn and Keras, it is straightforward to fit different machine learning models on a given predictive modeling dataset.

In this chapter, we present a high-level overview of supervised learning models. For a thorough coverage of the topics, the reader is referred to *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, 2nd Edition, by Aurélien Géron (O'Reilly).

The following topics are covered in this chapter:

- Basic concepts of supervised learning models (both regression and classification).
- How to implement different supervised learning models in Python.
- How to tune the models and identify the optimal parameters of the models using grid search.
- Overfitting versus underfitting and bias versus variance.
- Strengths and weaknesses of several supervised learning models.
- How to use ensemble models, ANN, and deep learning models for both regression and classification.
- How to select a model on the basis of several factors, including model performance.
- Evaluation metrics for classification and regression models.
- How to perform cross validation.

Supervised Learning Models: An Overview

Classification predictive modeling problems are different from regression predictive modeling problems, as classification is the task of predicting a discrete class label and regression is the task of predicting a continuous quantity. However, both share the same concept of utilizing known variables to make predictions, and there is a significant overlap between the two models. Hence, the models for classification and regression are presented together in this chapter. [Figure 4-1](#) summarizes the list of the models commonly used for classification and regression.

Some models can be used for both classification and regression with small modifications. These are K -nearest neighbors, decision trees, support vector, ensemble bagging/boosting methods, and ANNs (including deep neural networks), as shown in [Figure 4-1](#). However, some models, such as linear regression and logistic regression, cannot (or cannot easily) be used for both problem types.

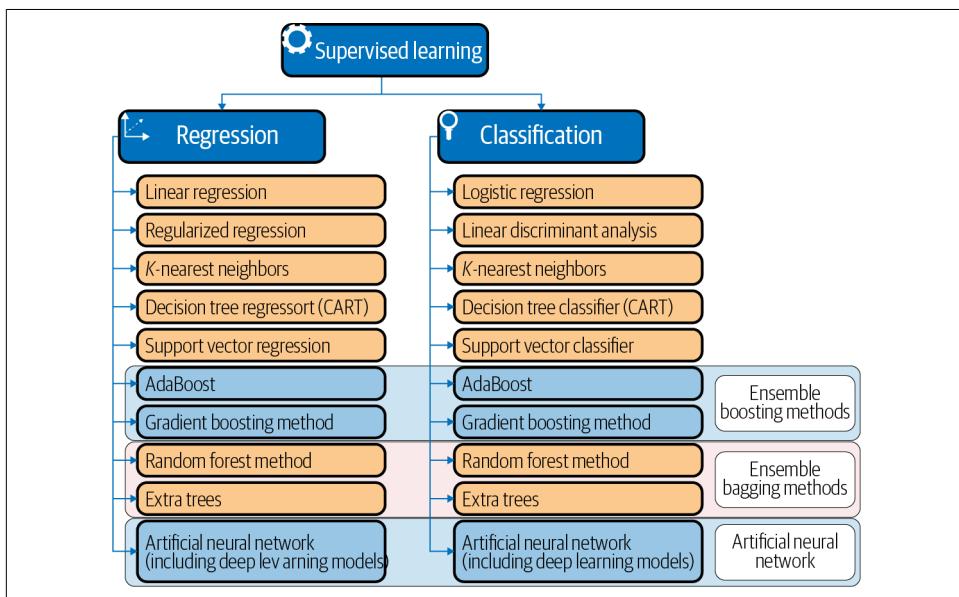


Figure 4-1. Models for regression and classification

This section contains the following details about the models:

- Theory of the models.
- Implementation in Scikit-learn or Keras.
- Grid search for different models.
- Pros and cons of the models.



In finance, a key focus is on models that extract signals from previously observed data in order to predict future values for the same time series. This family of time series models predicts continuous output and is more aligned with the supervised regression models. Time series models are covered separately in the supervised regression chapter ([Chapter 5](#)).

Linear Regression (Ordinary Least Squares)

Linear regression (Ordinary Least Squares Regression or OLS Regression) is perhaps one of the most well-known and best-understood algorithms in statistics and machine learning. Linear regression is a linear model, e.g., a model that assumes a linear relationship between the input variables (x) and the single output variable (y). The goal of linear regression is to train a linear model to predict a new y given a previously unseen x with as little error as possible.

Our model will be a function that predicts y given $x_1, x_2 \dots x_i$:

$$y = \beta_0 + \beta_1 x_1 + \dots + \beta_i x_i$$

where, β_0 is called intercept and $\beta_1 \dots \beta_i$ are the coefficient of the regression.

Implementation in Python

```
from sklearn.linear_model import LinearRegression  
model = LinearRegression()  
model.fit(X, Y)
```

In the following section, we cover the training of a linear regression model and grid search of the model. However, the overall concepts and related approaches are applicable to all other supervised learning models.

Training a model

As we mentioned in [Chapter 3](#), training a model basically means retrieving the model parameters by minimizing the cost (loss) function. The two steps for training a linear regression model are:

Define a cost function (or loss function)

Measures how inaccurate the model's predictions are. The *sum of squared residuals* (RSS) as defined in [Equation 4-1](#) measures the squared sum of the difference between the actual and predicted value and is the cost function for linear regression.

Equation 4-1. Sum of squared residuals

$$RSS = \sum_{i=1}^n (y_i - \beta_0 - \sum_{j=1}^n \beta_j x_{ij})^2$$

In this equation, β_0 is the intercept; β_j represents the coefficient; β_1, \dots, β_j are the coefficients of the regression; and x_{ij} represents the i^{th} observation and j^{th} variable.

Find the parameters that minimize loss

For example, make our model as accurate as possible. Graphically, in two dimensions, this results in a line of best fit as shown in [Figure 4-2](#). In higher dimensions, we would have higher-dimensional hyperplanes. Mathematically, we look at the difference between each real data point (y) and our model's prediction (\hat{y}). Square these differences to avoid negative numbers and penalize larger differences, and then add them up and take the average. This is a measure of how well our data fits the line.

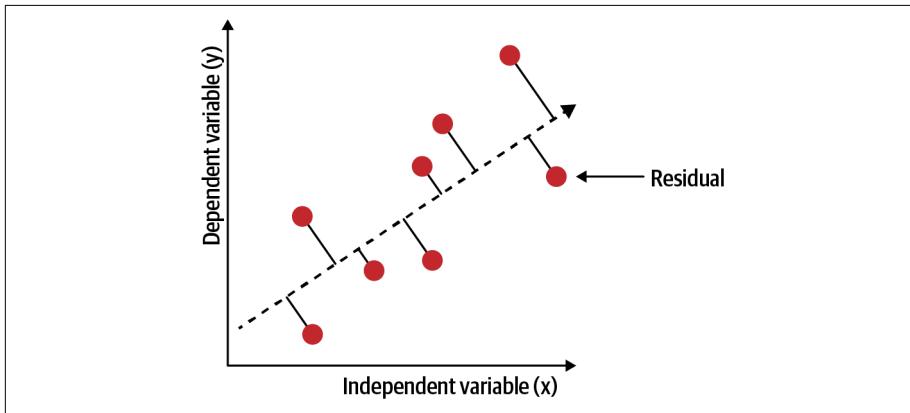


Figure 4-2. Linear regression

Grid search

The overall idea of the grid search is to create a grid of all possible hyperparameter combinations and train the model using each one of them. Hyperparameters are the external characteristic of the model, can be considered the model's settings, and are not estimated based on data-like model parameters. These hyperparameters are tuned during grid search to achieve better model performance.

Due to its exhaustive search, a grid search is guaranteed to find the optimal parameter within the grid. The drawback is that the size of the grid grows exponentially with the addition of more parameters or more considered values.

The `GridSearchCV` class in the `model_selection` module of the `sklearn` package facilitates the systematic evaluation of all combinations of the hyperparameter values that we would like to test.

The first step is to create a model object. We then define a dictionary where the keywords name the hyperparameters and the values list the parameter settings to be tested. For linear regression, the hyperparameter is `fit_intercept`, which is a boolean variable that determines whether or not to calculate the *intercept* for this model. If set to `False`, no intercept will be used in calculations:

```
model = LinearRegression()  
param_grid = {'fit_intercept': [True, False]}  
}
```

The second step is to instantiate the `GridSearchCV` object and provide the estimator object and parameter grid, as well as a scoring method and cross validation choice, to the initialization method. Cross validation is a resampling procedure used to evaluate machine learning models, and scoring parameter is the evaluation metrics of the model:¹

With all settings in place, we can fit `GridSearchCV`:

```
grid = GridSearchCV(estimator=model, param_grid=param_grid, scoring='r2', \  
                    cv=kfold)  
grid_result = grid.fit(X, Y)
```

Advantages and disadvantages

In terms of advantages, linear regression is easy to understand and interpret. However, it may not work well when there is a nonlinear relationship between predicted and predictor variables. Linear regression is prone to *overfitting* (which we will discuss in the next section) and when a large number of features are present, it may not handle irrelevant features well. Linear regression also requires the data to follow certain **assumptions**, such as the absence of multicollinearity. If the assumptions fail, then we cannot trust the results obtained.

¹ Cross validation will be covered in detail later in this chapter.

Regularized Regression

When a linear regression model contains many independent variables, their coefficients will be poorly determined, and the model will have a tendency to fit extremely well to the training data (data used to build the model) but fit poorly to testing data (data used to test how good the model is). This is known as overfitting or high variance.

One popular technique to control overfitting is *regularization*, which involves the addition of a *penalty* term to the error or loss function to discourage the coefficients from reaching large values. Regularization, in simple terms, is a penalty mechanism that applies shrinkage to model parameters (driving them closer to zero) in order to build a model with higher prediction accuracy and interpretation. Regularized regression has two advantages over linear regression:

Prediction accuracy

The performance of the model working better on the testing data suggests that the model is trying to generalize from training data. A model with too many parameters might try to fit noise specific to the training data. By shrinking or setting some coefficients to zero, we trade off the ability to fit complex models (higher bias) for a more generalizable model (lower variance).

Interpretation

A large number of predictors may complicate the interpretation or communication of the big picture of the results. It may be preferable to sacrifice some detail to limit the model to a smaller subset of parameters with the strongest effects.

The common ways to regularize a linear regression model are as follows:

L1 regularization or Lasso regression

Lasso regression performs *L1 regularization* by adding a factor of the sum of the absolute value of coefficients in the cost function (RSS) for linear regression, as mentioned in [Equation 4-1](#). The equation for lasso regularization can be represented as follows:

$$\text{CostFunction} = \text{RSS} + \lambda * \sum_{j=1}^p |\beta_j|$$

L1 regularization can lead to zero coefficients (i.e., some of the features are completely neglected for the evaluation of output). The larger the value of λ , the more features are shrunk to zero. This can eliminate some features entirely and give us a subset of predictors, reducing model complexity. So Lasso regression not only helps in reducing overfitting, but also can help in feature selection. Predictors not shrunk toward zero signify that they are important, and thus L1 regularization allows for feature selection (sparse selection). The regularization parameter (λ) can be controlled, and a `lambda` value of zero produces the basic linear regression equation.

A lasso regression model can be constructed using the `Lasso` class of the `sklearn` package of Python, as shown in the code snippet that follows:

```
from sklearn.linear_model import Lasso
model = Lasso()
model.fit(X, Y)
```

L2 regularization or Ridge regression

Ridge regression performs *L2 regularization* by adding a factor of the sum of the square of coefficients in the cost function (RSS) for linear regression, as mentioned in [Equation 4-1](#). The equation for ridge regularization can be represented as follows:

$$\text{CostFunction} = \text{RSS} + \lambda * \sum_{j=1}^p \beta_j^2$$

Ridge regression puts constraint on the coefficients. The penalty term (λ) regularizes the coefficients such that if the coefficients take large values, the optimization function is penalized. So ridge regression shrinks the coefficients and helps to reduce the model complexity. Shrinking the coefficients leads to a lower variance and a lower error value. Therefore, ridge regression decreases the complexity of a model but does not reduce the number of variables; it just shrinks their effect. When λ is closer to zero, the cost function becomes similar to the linear regression cost function. So the lower the constraint (low λ) on the features, the more the model will resemble the linear regression model.

A ridge regression model can be constructed using the `Ridge` class of the `sklearn` package of Python, as shown in the code snippet that follows:

```
from sklearn.linear_model import Ridge
model = Ridge()
model.fit(X, Y)
```

Elastic net

Elastic nets add regularization terms to the model, which are a combination of both L1 and L2 regularization, as shown in the following equation:

$$\text{CostFunction} = \text{RSS} + \lambda * ((1 - \alpha) / 2 * \sum_{j=1}^p \beta_j^2 + \alpha * \sum_{j=1}^p |\beta_j|)$$

In addition to setting and choosing a λ value, an elastic net also allows us to tune the alpha parameter, where $\alpha = 0$ corresponds to ridge and $\alpha = 1$ to lasso. Therefore, we can choose an α value between 0 and 1 to optimize the elastic net. Effectively, this will shrink some coefficients and set some to 0 for sparse selection.

An elastic net regression model can be constructed using the `ElasticNet` class of the `sklearn` package of Python, as shown in the following code snippet:

```
from sklearn.linear_model import ElasticNet
model = ElasticNet()
model.fit(X, Y)
```

For all the regularized regression, λ is the key parameter to tune during grid search in Python. In an elastic net, α can be an additional parameter to tune.

Logistic Regression

Logistic regression is one of the most widely used algorithms for classification. The logistic regression model arises from the desire to model the probabilities of the output classes given a function that is linear in x , at the same time ensuring that output probabilities sum up to one and remain between zero and one as we would expect from probabilities.

If we train a linear regression model on several examples where $Y = 0$ or 1 , we might end up predicting some probabilities that are less than zero or greater than one, which doesn't make sense. Instead, we use a logistic regression model (or *logit* model), which is a modification of linear regression that makes sure to output a probability between zero and one by applying the *sigmoid* function.²

[Equation 4-2](#) shows the equation for a logistic regression model. Similar to linear regression, input values (x) are combined linearly using weights or coefficient values to predict an output value (y). The output coming from [Equation 4-2](#) is a probability that is transformed into a binary value (0 or 1) to get the model prediction.

Equation 4-2. Logistic regression equation

$$y = \frac{\exp(\beta_0 + \beta_1 x_1 + \dots + \beta_i x_i)}{1 + \exp(\beta_0 + \beta_1 x_1 + \dots + \beta_i x_i)}$$

Where y is the predicted output, β_0 is the bias or intercept term and β_1 is the coefficient for the single input value (x). Each column in the input data has an associated β coefficient (a constant real value) that must be learned from the training data.

In logistic regression, the cost function is basically a measure of how often we predicted one when the true answer was zero, or vice versa. Training the logistic regression coefficients is done using techniques such as maximum likelihood estimation (MLE) to predict values close to 1 for the default class and close to 0 for the other class.³

² See the activation function section of [Chapter 3](#) for details on the *sigmoid* function.

³ [MLE](#) is a method of estimating the parameters of a probability distribution so that under the assumed statistical model the observed data is most probable.

A logistic regression model can be constructed using the `LogisticRegression` class of the `sklearn` package of Python, as shown in the following code snippet:

```
from sklearn.linear_model import LogisticRegression  
model = LogisticRegression()  
model.fit(X, Y)
```

Hyperparameters

Regularization (penalty in sklearn)

Similar to linear regression, logistic regression can have regularization, which can be $L1$, $L2$, or *elasticnet*. The values in the `sklearn` library are $[l1, l2, elasticnet]$.

Regularization strength (C in sklearn)

This parameter controls the regularization strength. Good values of the penalty parameters can be $[100, 10, 1.0, 0.1, 0.01]$.

Advantages and disadvantages

In terms of the advantages, the logistic regression model is easy to implement, has good interpretability, and performs very well on linearly separable classes. The output of the model is a probability, which provides more insight and can be used for ranking. The model has small number of hyperparameters. Although there may be risk of overfitting, this may be addressed using $L1/L2$ regularization, similar to the way we addressed overfitting for the linear regression models.

In terms of disadvantages, the model may overfit when provided with large numbers of features. Logistic regression can only learn linear functions and is less suitable to complex relationships between features and the target variable. Also, it may not handle irrelevant features well, especially if the features are strongly correlated.

Support Vector Machine

The objective of the *support vector machine* (SVM) algorithm is to maximize the margin (shown as shaded area in [Figure 4-3](#)), which is defined as the distance between the separating hyperplane (or decision boundary) and the training samples that are closest to this hyperplane, the so-called support vectors. The margin is calculated as the perpendicular distance from the line to only the closest points, as shown in [Figure 4-3](#). Hence, SVM calculates a maximum-margin boundary that leads to a homogeneous partition of all data points.

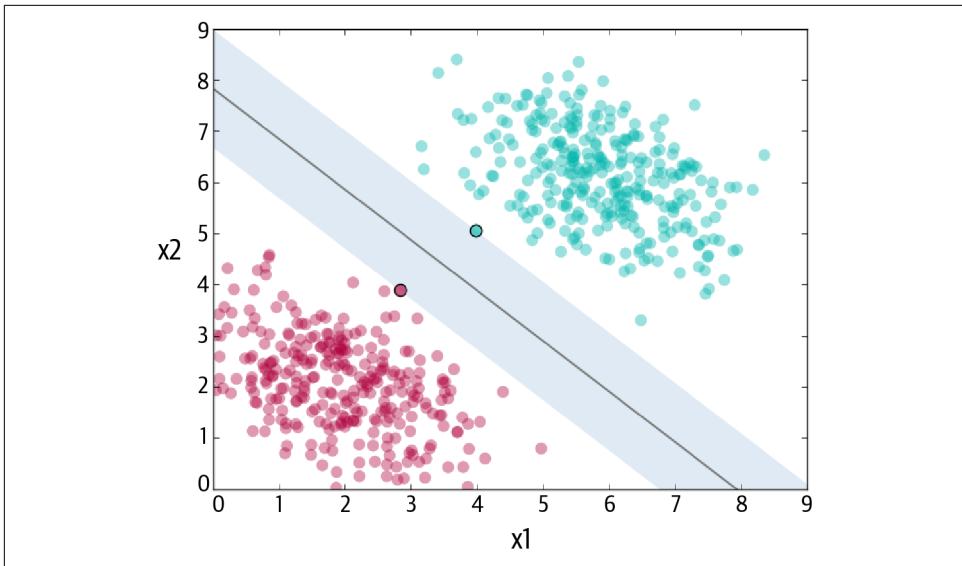


Figure 4-3. Support vector machine

In practice, the data is messy and cannot be separated perfectly with a hyperplane. The constraint of maximizing the margin of the line that separates the classes must be relaxed. This change allows some points in the training data to violate the separating line. An additional set of coefficients is introduced that give the margin wiggle room in each dimension. A tuning parameter is introduced, simply called C , that defines the magnitude of the wiggle allowed across all dimensions. The larger the value of C , the more violations of the hyperplane are permitted.

In some cases, it is not possible to find a hyperplane or a linear decision boundary, and kernels are used. A kernel is just a transformation of the input data that allows the SVM algorithm to treat/process the data more easily. Using kernels, the original data is projected into a higher dimension to classify the data better.

SVM is used for both classification and regression. We achieve this by converting the original optimization problem into a dual problem. For regression, the trick is to reverse the objective. Instead of trying to fit the largest possible street between two classes while limiting margin violations, SVM regression tries to fit as many instances as possible on the street (shaded area in [Figure 4-3](#)) while limiting margin violations. The width of the street is controlled by a hyperparameter.

The SVM regression and classification models can be constructed using the `sklearn` package of Python, as shown in the following code snippets:

Regression

```
from sklearn.svm import SVR  
model = SVR()  
model.fit(X, Y)
```

Classification

```
from sklearn.svm import SVC  
model = SVC()  
model.fit(X, Y)
```

Hyperparameters

The following key parameters are present in the sklearn implementation of SVM and can be tweaked while performing the grid search:

Kernels (kernel in sklearn)

The choice of kernel controls the manner in which the input variables will be projected. There are many kernels to choose from, but *linear* and *RBF* are the most common.

Penalty (C in sklearn)

The penalty parameter tells the SVM optimization how much you want to avoid misclassifying each training example. For large values of the penalty parameter, the optimization will choose a smaller-margin hyperplane. Good values might be a log scale from 10 to 1,000.

Advantages and disadvantages

In terms of advantages, SVM is fairly robust against overfitting, especially in higher dimensional space. It handles the nonlinear relationships quite well, with many kernels to choose from. Also, there is no distributional requirement for the data.

In terms of disadvantages, SVM can be inefficient to train and memory-intensive to run and tune. It doesn't perform well with large datasets. It requires the feature scaling of the data. There are also many hyperparameters, and their meanings are often not intuitive.

K-Nearest Neighbors

K-nearest neighbors (KNN) is considered a “lazy learner,” as there is no learning required in the model. For a new data point, predictions are made by searching through the entire training set for the K most similar instances (the neighbors) and summarizing the output variable for those K instances.

To determine which of the K instances in the training dataset are most similar to a new input, a distance measure is used. The most popular distance measure is *Eucli-*

dean distance, which is calculated as the square root of the sum of the squared differences between a point a and a point b across all input attributes i , and which is represented as $d(a, b) = \sqrt{\sum_{i=1}^n (a_i - b_i)^2}$. Euclidean distance is a good distance measure to use if the input variables are similar in type.

Another distance metric is *Manhattan distance*, in which the distance between point a and point b is represented as $d(a, b) = \sum_{i=1}^n |a_i - b_i|$. Manhattan distance is a good measure to use if the input variables are not similar in type.

The steps of KNN can be summarized as follows:

1. Choose the number of K and a distance metric.
2. Find the K -nearest neighbors of the sample that we want to classify.
3. Assign the class label by majority vote.

KNN regression and classification models can be constructed using the `sklearn` package of Python, as shown in the following code:

Classification

```
from sklearn.neighbors import KNeighborsClassifier  
model = KNeighborsClassifier()  
model.fit(X, Y)
```

Regression

```
from sklearn.neighbors import KNeighborsRegressor  
model = KNeighborsRegressor()  
model.fit(X, Y)
```

Hyperparameters

The following key parameters are present in the `sklearn` implementation of KNN and can be tweaked while performing the grid search:

Number of neighbors (n_neighbors in sklearn)

The most important hyperparameter for KNN is the number of neighbors (`n_neighbors`). Good values are between 1 and 20.

Distance metric (metric in sklearn)

It may also be interesting to test different distance metrics for choosing the composition of the neighborhood. Good values are *euclidean* and *manhattan*.

Advantages and disadvantages

In terms of advantages, no training is involved and hence there is no learning phase. Since the algorithm requires no training before making predictions, ~~new data can be added seamlessly without impacting the accuracy of the algorithm~~. It is intuitive and

easy to understand. The model naturally handles multiclass classification and can learn complex decision boundaries. KNN is effective if the training data is large. It is also robust to noisy data, and there is no need to filter the outliers.

In terms of the disadvantages, the distance metric to choose is not obvious and difficult to justify in many cases. KNN performs poorly on high dimensional datasets. It is expensive and slow to predict new instances because the distance to all neighbors must be recalculated. KNN is sensitive to noise in the dataset. We need to manually input missing values and remove outliers. Also, feature scaling (standardization and normalization) is required before applying the KNN algorithm to any dataset; otherwise, KNN may generate wrong predictions.

Linear Discriminant Analysis

The objective of the *linear discriminant analysis* (LDA) algorithm is to project the data onto a lower-dimensional space in a way that the class separability is maximized and the variance within a class is minimized.⁴

During the training of the LDA model, the statistical properties (i.e., mean and covariance matrix) of each class are computed. The statistical properties are estimated on the basis of the following assumptions about the data:

- Data is **normally distributed**, so that each variable is shaped like a bell curve when plotted.
- Each attribute has the same variance, and the values of each variable vary around the mean by the same amount on average.

To make a prediction, LDA estimates the probability that a new set of inputs belongs to every class. The output class is the one that has the highest probability.

Implementation in Python and hyperparameters

The LDA classification model can be constructed using the `sklearn` package of Python, as shown in the following code snippet:

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis  
model = LinearDiscriminantAnalysis()  
model.fit(X, Y)
```

The key hyperparameter for the LDA model is `number of components` for dimensionality reduction, which is represented by `n_components` in `sklearn`.

⁴ The approach of projecting data is similar to the PCA algorithm discussed in [Chapter 7](#).

Advantages and disadvantages

In terms of advantages, LDA is a relatively simple model with fast implementation and is easy to implement. In terms of disadvantages, it requires feature scaling and involves complex matrix operations.

Classification and Regression Trees

In the most general terms, the purpose of an analysis via tree-building algorithms is to determine a set of *if-then* logical (split) conditions that permit accurate prediction or classification of cases. *Classification and regression trees* (or *CART* or *decision tree classifiers*) are attractive models if we care about interpretability. We can think of this model as breaking down our data and making a decision based on asking a series of questions. This algorithm is the foundation of ensemble methods such as random forest and gradient boosting method.

Representation

The model can be represented by a *binary tree* (or *decision tree*), where each node is an input variable x with a split point and each leaf contains an output variable y for prediction.

Figure 4-4 shows an example of a simple classification tree to predict whether a person is a male or a female based on two inputs of height (in centimeters) and weight (in kilograms).

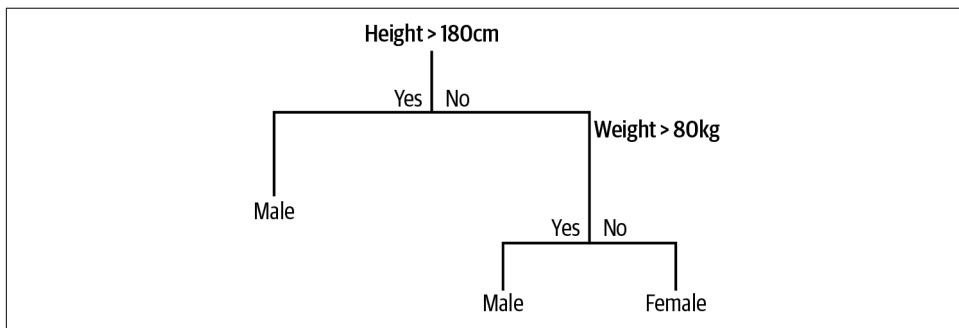


Figure 4-4. Classification and regression tree example

Learning a CART model

Creating a binary tree is actually a process of dividing up the input space. A *greedy approach* called *recursive binary splitting* is used to divide the space. This is a numerical procedure in which all the values are lined up and different split points are tried and tested using a cost (loss) function. The split with the best cost (lowest cost, because we minimize cost) is selected. All input variables and all possible split points

are evaluated and chosen in a greedy manner (e.g., the very best split point is chosen each time).

For regression predictive modeling problems, the cost function that is minimized to choose split points is the *sum of squared errors* across all training samples that fall within the rectangle:

$$\sum_{i=1}^n (y_i - \text{prediction}_i)^2$$

where y_i is the output for the training sample and prediction is the predicted output for the rectangle. For classification, the *Gini cost function* is used; it provides an indication of how pure the leaf nodes are (i.e., how mixed the training data assigned to each node is) and is defined as:

$$G = \sum_{i=1}^n p_k * (1 - p_k)$$

where G is the Gini cost over all classes and p_k is the number of training instances with class k in the rectangle of interest. A node that has all classes of the same type (perfect class purity) will have $G = 0$, while a node that has a 50–50 split of classes for a binary classification problem (worst purity) will have $G = 0.5$.

Stopping criterion

The recursive binary splitting procedure described in the preceding section needs to know when to stop splitting as it works its way down the tree with the training data. The most common stopping procedure is to use a minimum count on the number of training instances assigned to each leaf node. If the count is less than some minimum, then the split is not accepted and the node is taken as a final leaf node.

Pruning the tree

The stopping criterion is important as it strongly influences the performance of the tree. Pruning can be used after learning the tree to further lift performance. The complexity of a decision tree is defined as the number of splits in the tree. Simpler trees are preferred as they are faster to run and easy to understand, consume less memory during processing and storage, and are less likely to overfit the data. The fastest and simplest pruning method is to work through each leaf node in the tree and evaluate the effect of removing it using a test set. A leaf node is removed only if doing so results in a drop in the overall cost function on the entire test set. The removal of nodes can be stopped when no further improvements can be made.

Implementation in Python

CART regression and classification models can be constructed using the `sklearn` package of Python, as shown in the following code snippet:

Classification

```
from sklearn.tree import DecisionTreeClassifier  
model = DecisionTreeClassifier()  
model.fit(X, Y)
```

Regression

```
from sklearn.tree import DecisionTreeRegressor  
model = DecisionTreeRegressor()  
model.fit(X, Y)
```

Hyperparameters

CART has many hyperparameters. However, the key hyperparameter is the maximum depth of the tree model, which is the number of components for dimensionality reduction, and which is represented by `max_depth` in the `sklearn` package. Good values can range from 2 to 30 depending on the number of features in the data.

Advantages and disadvantages

In terms of advantages, CART is easy to interpret and can adapt to learn complex relationships. It requires little data preparation, and data typically does not need to be scaled. Feature importance is built in due to the way decision nodes are built. It performs well on large datasets. It works for both regression and classification problems.

In terms of disadvantages, CART is prone to overfitting unless pruning is used. It can be very nonrobust, meaning that small changes in the training dataset can lead to quite major differences in the hypothesis function that gets learned. CART generally has worse performance than ensemble models, which are covered next.

Ensemble Models

The goal of *ensemble models* is to combine different classifiers into a meta-classifier that has better generalization performance than each individual classifier alone. For example, assuming that we collected predictions from 10 experts, ensemble methods would allow us to strategically combine their predictions to come up with a prediction that is more accurate and robust than the experts' individual predictions.

The two most popular ensemble methods are bagging and boosting. *Bagging* (or *bootstrap aggregation*) is an ensemble technique of training several individual models in a parallel way. Each model is trained by a random subset of the data. *Boosting*, on the other hand, is an ensemble technique of training several individual models in a sequential way. This is done by building a model from the training data and then

creating a second model that attempts to correct the errors of the first model. Models are added until the training set is predicted perfectly or a maximum number of models is added. Each individual model learns from mistakes made by the previous model. Just like the decision trees themselves, bagging and boosting can be used for classification and regression problems.

By combining individual models, the ensemble model tends to be more flexible (less bias) and less data-sensitive (less variance).⁵ Ensemble methods combine multiple, simpler algorithms to obtain better performance.

In this section we will cover random forest, AdaBoost, the gradient boosting method, and extra trees, along with their implementation using sklearn package.

Random forest. *Random forest* is a tweaked version of bagged decision trees. In order to understand a random forest algorithm, let us first understand the *bagging algorithm*. Assuming we have a dataset of one thousand instances, the steps of bagging are:

1. Create many (e.g., one hundred) random subsamples of our dataset.
2. Train a CART model on each sample.
3. Given a new dataset, calculate the average prediction from each model and aggregate the prediction by each tree to assign the final label by majority vote.

A problem with decision trees like CART is that they are greedy. They choose the variable to split by using a greedy algorithm that minimizes error. Even after bagging, the decision trees can have a lot of structural similarities and result in high correlation in their predictions. Combining predictions from multiple models in ensembles works better if the predictions from the submodels are uncorrelated, or at best are weakly correlated. Random forest changes the learning algorithm in such a way that the resulting predictions from all of the subtrees have less correlation.

In CART, when selecting a split point, the learning algorithm is allowed to look through all variables and all variable values in order to select the most optimal split point. The random forest algorithm changes this procedure such that each subtree can access only a random sample of features when selecting the split points. The number of features that can be searched at each split point (m) must be specified as a parameter to the algorithm.

As the bagged decision trees are constructed, we can calculate how much the error function drops for a variable at each split point. In regression problems, this may be the drop in sum squared error, and in classification, this might be the Gini cost. The

⁵ Bias and variance are described in detail later in this chapter.

bagged method can provide feature importance by calculating and averaging the error function drop for individual variables.

Implementation in Python. Random forest regression and classification models can be constructed using the sklearn package of Python, as shown in the following code:

Classification

```
from sklearn.ensemble import RandomForestClassifier  
model = RandomForestClassifier()  
model.fit(X, Y)
```

Regression

```
from sklearn.ensemble import RandomForestRegressor  
model = RandomForestRegressor()  
model.fit(X, Y)
```

Hyperparameters. Some of the main hyperparameters that are present in the sklearn implementation of random forest and that can be tweaked while performing the grid search are:

Maximum number of features (max_features in sklearn)

This is the most important parameter. It is the number of random features to sample at each split point. You could try a range of integer values, such as 1 to 20, or 1 to half the number of input features.

Number of estimators (n_estimators in sklearn)

This parameter represents the number of trees. Ideally, this should be increased until no further improvement is seen in the model. Good values might be a log scale from 10 to 1,000.

Advantages and disadvantages. The random forest algorithm (or model) has gained huge popularity in ML applications during the last decade due to its good performance, scalability, and ease of use. It is flexible and naturally assigns feature importance scores, so it can handle redundant feature columns. It scales to large datasets and is generally robust to overfitting. The algorithm doesn't need the data to be scaled and can model a nonlinear relationship.

In terms of disadvantages, random forest can feel like a black box approach, as we have very little control over what the model does, and the results may be difficult to interpret. Although random forest does a good job at classification, it may not be good for regression problems, as it does not give a precise continuous nature prediction. In the case of regression, it doesn't predict beyond the range in the training data and may overfit datasets that are particularly noisy.

Extra trees

Extra trees, otherwise known as *extremely randomized trees*, is a variant of a random forest; it builds multiple trees and splits nodes using random subsets of features similar to random forest. However, unlike random forest, where observations are drawn with replacement, the observations are drawn without replacement in extra trees. So there is no repetition of observations.

Additionally, random forest selects the best split to convert the parent into the two most homogeneous child nodes.⁶ However, extra trees selects a random split to divide the parent node into two random child nodes. In extra trees, randomness doesn't come from bootstrapping the data; it comes from the random splits of all observations.

In real-world cases, performance is comparable to an ordinary random forest, sometimes a bit better. The advantages and disadvantages of extra trees are similar to those of random forest.

Implementation in Python. Extra trees regression and classification models can be constructed using the `sklearn` package of Python, as shown in the following code snippet. The hyperparameters of extra trees are similar to random forest, as shown in the previous section:

Classification

```
from sklearn.ensemble import ExtraTreesClassifier
model = ExtraTreesClassifier()
model.fit(X, Y)
```

Regression

```
from sklearn.ensemble import ExtraTreesRegressor
model = ExtraTreesRegressor()
model.fit(X, Y)
```

Adaptive Boosting (AdaBoost)

Adaptive Boosting or *AdaBoost* is a boosting technique in which the basic idea is to try predictors sequentially, and each subsequent model attempts to fix the errors of its predecessor. At each iteration, the AdaBoost algorithm changes the sample distribution by modifying the weights attached to each of the instances. It increases the weights of the wrongly predicted instances and decreases the ones of the correctly predicted instances.

⁶ Split is the process of converting a nonhomogeneous parent node into two homogeneous child nodes (best possible).

The steps of the AdaBoost algorithm are:

1. Initially, all observations are given equal weights.
2. A model is built on a subset of data, and using this model, predictions are made on the whole dataset. Errors are calculated by comparing the predictions and actual values.
3. While creating the next model, higher weights are given to the data points that were predicted incorrectly. Weights can be determined using the error value. For instance, the higher the error, the more weight is assigned to the observation.
4. This process is repeated until the error function does not change, or until the maximum limit of the number of estimators is reached.

Implementation in Python. AdaBoost regression and classification models can be constructed using the `sklearn` package of Python, as shown in the following code snippet:

Classification

```
from sklearn.ensemble import AdaBoostClassifier  
model = AdaBoostClassifier()  
model.fit(X, Y)
```

Regression

```
from sklearn.ensemble import AdaBoostRegressor  
model = AdaBoostRegressor()  
model.fit(X, Y)
```

Hyperparameters. Some of the main hyperparameters that are present in the `sklearn` implementation of AdaBoost and that can be tweaked while performing the grid search are as follows:

Learning rate (learning_rate in sklearn)

Learning rate shrinks the contribution of each classifier/regressor. It can be considered on a log scale. The sample values for grid search can be 0.001, 0.01, and 0.1.

Number of estimators (n_estimators in sklearn)

This parameter represents the number of trees. Ideally, this should be increased until no further improvement is seen in the model. Good values might be a log scale from 10 to 1,000.

Advantages and disadvantages. In terms of advantages, AdaBoost has a high degree of precision. AdaBoost can achieve similar results to other models with much less tweaking of parameters or settings. The algorithm doesn't need the data to be scaled and can model a nonlinear relationship.

In terms of disadvantages, the training of AdaBoost is time consuming. AdaBoost can be sensitive to noisy data and outliers, and data imbalance leads to a decrease in classification accuracy.

Gradient boosting method

Gradient boosting method (GBM) is another boosting technique similar to AdaBoost, where the general idea is to try predictors sequentially. Gradient boosting works by sequentially adding the previous underfitted predictions to the ensemble, ensuring the errors made previously are corrected.

The following are the steps of the gradient boosting algorithm:

1. A model (which can be referred to as the first weak learner) is built on a subset of data. Using this model, predictions are made on the whole dataset.
2. Errors are calculated by comparing the predictions and actual values, and the loss is calculated using the loss function.
3. A new model is created using the errors of the previous step as the target variable. The objective is to find the best split in the data to minimize the error. The predictions made by this new model are combined with the predictions of the previous. New errors are calculated using this predicted value and actual value.
4. This process is repeated until the error function does not change or until the maximum limit of the number of estimators is reached.

Contrary to AdaBoost, which tweaks the instance weights at every interaction, this method tries to fit the new predictor to the residual errors made by the previous predictor.

Implementation in Python and hyperparameters. Gradient boosting method regression and classification models can be constructed using the `sklearn` package of Python, as shown in the following code snippet. The hyperparameters of gradient boosting method are similar to AdaBoost, as shown in the previous section:

Classification

```
from sklearn.ensemble import GradientBoostingClassifier  
model = GradientBoostingClassifier()  
model.fit(X, Y)
```

Regression

```
from sklearn.ensemble import GradientBoostingRegressor  
model = GradientBoostingRegressor()  
model.fit(X, Y)
```

Advantages and disadvantages. In terms of advantages, gradient boosting method is robust to missing data, highly correlated features, and irrelevant features in the same way as random forest. It naturally assigns feature importance scores, with slightly better performance than random forest. The algorithm doesn't need the data to be scaled and can model a nonlinear relationship.

In terms of disadvantages, it may be more prone to overfitting than random forest, as the main purpose of the boosting approach is to reduce bias and not variance. It has many hyperparameters to tune, so model development may not be as fast. Also, feature importance may not be robust to variation in the training dataset.

ANN-Based Models

In [Chapter 3](#) we covered the basics of ANNs, along with the architecture of ANNs and their training and implementation in Python. The details provided in that chapter are applicable across all areas of machine learning, including supervised learning. However, there are a few additional details from the supervised learning perspective, which we will cover in this section.

Neural networks are reducible to a classification or regression model with the activation function of the node in the output layer. In the case of a regression problem, the output node has linear activation function (or no activation function). A linear function produces a continuous output ranging from $-\infty$ to $+\infty$. Hence, the output layer will be the linear function of the nodes in the layer before the output layer, and it will be a regression-based model.

In the case of a classification problem, the output node has a sigmoid or softmax activation function. A sigmoid or softmax function produces an output ranging from zero to one to represent the probability of target value. Softmax function can also be used for multiple groups for classification.

ANN using sklearn

ANN regression and classification models can be constructed using the `sklearn` package of Python, as shown in the following code snippet:

Classification

```
from sklearn.neural_network import MLPClassifier
model = MLPClassifier()
model.fit(X, Y)
```

Regression

```
from sklearn.neural_network import MLPRegressor
model = MLPRegressor()
model.fit(X, Y)
```

Hyperparameters

As we saw in [Chapter 3](#), ANN has many hyperparameters. Some of the hyperparameters that are present in the sklearn implementation of ANN and can be tweaked while performing the grid search are:

Hidden Layers (hidden_layer_sizes in sklearn)

It represents the number of layers and nodes in the ANN architecture. In sklearn implementation of ANN, the *i*th element represents the number of neurons in the *i*th hidden layer. A sample value for grid search in the sklearn implementation can be [(20,), (50,), (20, 20), (20, 30, 20)].

Activation Function (activation in sklearn)

It represents the activation function of a hidden layer. Some of the activation functions defined in [Chapter 3](#), such as `sigmoid`, `relu`, or `tanh`, can be used.

Deep neural network

ANNs with more than a single hidden layer are often called deep networks. We prefer using the library Keras to implement such networks, given the flexibility of the library. The detailed implementation of a deep neural network in Keras was shown in [Chapter 3](#). Similar to `MLPClassifier` and `MLPRegressor` in sklearn for classification and regression, Keras has modules called `KerasClassifier` and `KerasRegressor` that can be used for creating classification and regression models with deep network.

A popular problem in finance is time series prediction, which is predicting the next value of a time series based on a historical overview. Some of the deep neural networks, such as recurrent neural network (RNN), can be directly used for time series prediction. The details of this approach are provided in [Chapter 5](#).

Advantages and disadvantages

The main advantage of an ANN is that it captures the nonlinear relationship between the variables quite well. ANN can more easily learn rich representations and is good with a large number of input features with a large dataset. ANN is flexible in how it can be used. This is evident from its use across a wide variety of areas in machine learning and AI, including reinforcement learning and NLP, as discussed in [Chapter 3](#).

The main disadvantage of ANN is the interpretability of the model, which is a drawback that often cannot be ignored and is sometimes the determining factor when choosing a model. ANN is not good with small datasets and requires a lot of tweaking and guesswork. Choosing the right topology/algorithms to solve a problem is difficult. Also, ANN is computationally expensive and can take a lot of time to train.

Using ANNs for supervised learning in finance

If a simple model such as linear or logistic regression perfectly fits your problem, don't bother with ANN. However, if you are modeling a complex dataset and feel a need for better prediction power, give ANN a try. ANN is one of the most flexible models in adapting itself to the shape of the data, and using it for supervised learning problems can be an interesting and valuable exercise.

Model Performance

In the previous section, we discussed grid search as a way to find the right hyperparameter to achieve better performance. In this section, we will expand on that process by discussing the key components of evaluating the model performance, which are overfitting, cross validation, and evaluation metrics.

Overfitting and Underfitting

A common problem in machine learning is *overfitting*, which is defined by learning a function that perfectly explains the training data that the model learned from but doesn't generalize well to unseen test data. Overfitting happens when a model over-learns from the training data to the point that it starts picking up idiosyncrasies that aren't representative of patterns in the real world. This becomes especially problematic as we make our models increasingly more complex. *Underfitting* is a related issue in which the model is not complex enough to capture the underlying trend in the data. [Figure 4-5](#) illustrates overfitting and underfitting. The left-hand panel of [Figure 4-5](#) shows a linear regression model; a straight line clearly underfits the true function. The middle panel shows that a high degree polynomial approximates the true relationship reasonably well. On the other hand, a polynomial of a very high degree fits the small sample almost perfectly, and performs best on the training data, but this doesn't generalize, and it would do a horrible job at explaining a new data point.

The concepts of overfitting and underfitting are closely linked to *bias-variance trade-off*. *Bias* refers to the error due to overly simplistic assumptions or faulty assumptions in the learning algorithm. Bias results in underfitting of the data, as shown in the left-hand panel of [Figure 4-5](#). A high bias means our learning algorithm is missing important trends among the features. *Variance* refers to the error due to an overly complex model that tries to fit the training data as closely as possible. In high variance cases, the model's predicted values are extremely close to the actual values from the training set. High variance gives rise to overfitting, as shown in the right-hand panel of [Figure 4-5](#). Ultimately, in order to have a good model, we need low bias and low variance.

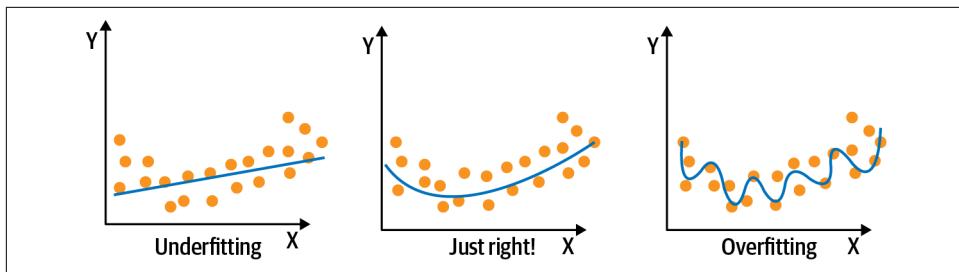


Figure 4-5. Overfitting and underfitting

There can be two ways to combat overfitting:

Using more training data

The more training data we have, the harder it is to overfit the data by learning too much from any single training example.

Using regularization

Adding a penalty in the loss function for building a model that assigns too much explanatory power to any one feature, or allows too many features to be taken into account.

The concept of overfitting and the ways to combat it are applicable across all the supervised learning models. For example, regularized regressions address overfitting in linear regression, as discussed earlier in this chapter.

Cross Validation

One of the challenges of machine learning is training models that are able to generalize well to unseen data (overfitting versus underfitting or a bias-variance trade-off). The main idea behind *cross validation* is to split the data one time or several times so that each split is used once as a validation set and the remainder is used as a training set: part of the data (the training sample) is used to train the algorithm, and the remaining part (the validation sample) is used for estimating the risk of the algorithm. Cross validation allows us to obtain reliable estimates of the model's generalization error. It is easiest to understand it with an example. When doing k -fold cross validation, we randomly split the training data into k folds. Then we train the model using $k-1$ folds and evaluate the performance on the k th fold. We repeat this process k times and average the resulting scores.

Figure 4-6 shows an example of cross validation, where the data is split into five sets and in each round one of the sets is used for validation.



Figure 4-6. Cross validation

A potential drawback of cross validation is the computational cost, especially when paired with a grid search for hyperparameter tuning. Cross validation can be performed in a couple of lines using the `sklearn` package; we will perform cross validation in the supervised learning case studies.

In the next section, we cover the evaluation metrics for the supervised learning models that are used to measure and compare the models' performance.

Evaluation Metrics

The metrics used to evaluate the machine learning algorithms are very important. The choice of metrics to use influences how the performance of machine learning algorithms is measured and compared. The metrics influence both how you weight the importance of different characteristics in the results and your ultimate choice of algorithm.

The main evaluation metrics for regression and classification are illustrated in [Figure 4-7](#).

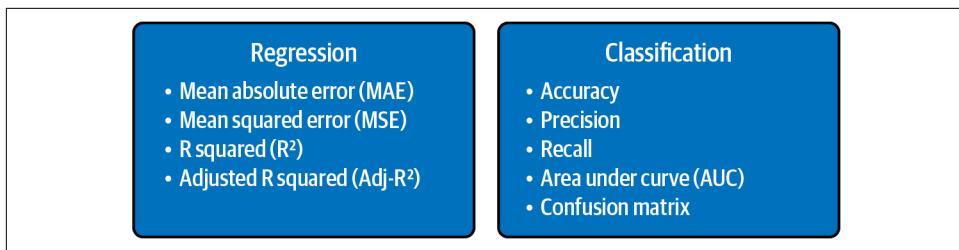


Figure 4-7. Evaluation metrics for regression and classification

Let us first look at the evaluation metrics for supervised regression.

Mean absolute error

The *mean absolute error* (MAE) is the sum of the absolute differences between predictions and actual values. The MAE is a linear score, which means that all the individual differences are weighted equally in the average. It gives an idea of how wrong the predictions were. The measure gives an idea of the magnitude of the error, but no idea of the direction (e.g., over- or underpredicting).

Mean squared error

The *mean squared error* (MSE) represents the sample standard deviation of the differences between predicted values and observed values (called residuals). This is much like the mean absolute error in that it provides a gross idea of the magnitude of the error. Taking the square root of the mean squared error converts the units back to the original units of the output variable and can be meaningful for description and presentation. This is called the *root mean squared error* (RMSE).

R² metric

The *R² metric* provides an indication of the “goodness of fit” of the predictions to actual value. In statistical literature this measure is called the coefficient of determination. This is a value between zero and one, for no-fit and perfect fit, respectively.

Adjusted R² metric

Just like R^2 , *adjusted R²* also shows how well terms fit a curve or line but adjusts for the number of terms in a model. It is given in the following formula:

$$R_{adj}^2 = 1 - \left[\frac{(1 - R^2)(n - 1)}{n - k - 1} \right]$$

where n is the total number of observations and k is the number of predictors. Adjusted R^2 will always be less than or equal to R^2 .

Selecting an evaluation metric for supervised regression

In terms of a preference among these evaluation metrics, if the main goal is predictive accuracy, then RMSE is best. It is computationally simple and is easily differentiable. The loss is symmetric, but larger errors weigh more in the calculation. The MAEs are symmetric but do not weigh larger errors more. R^2 and adjusted R^2 are often used for explanatory purposes by indicating how well the selected independent variable(s) explains the variability in the dependent variable(s).

Let us first look at the evaluation metrics for supervised classification.

Classification

For simplicity, we will mostly discuss things in terms of a binary classification problem (i.e., only two outcomes, such as true or false); some common terms are:

True positives (TP)

Predicted positive and are actually positive.

False positives (FP)

Predicted positive and are actually negative.

True negatives (TN)

Predicted negative and are actually negative.

False negatives (FN)

Predicted negative and are actually positive.

The difference between three commonly used evaluation metrics for classification, accuracy, precision, and recall, is illustrated in [Figure 4-8](#).

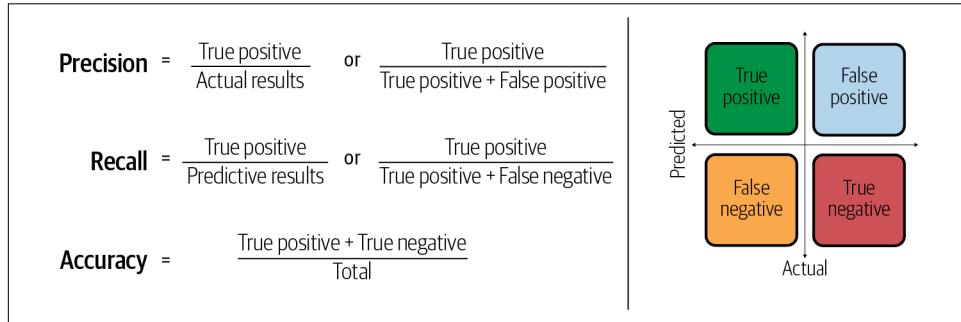


Figure 4-8. Accuracy, precision, and recall

Accuracy

As shown in [Figure 4-8](#), accuracy is the number of correct predictions made as a ratio of all predictions made. This is the most common evaluation metric for classification problems and is also the most misused. It is most suitable when there are an equal number of observations in each class (which is rarely the case) and when all predictions and the related prediction errors are equally important, which is often not the case.

Precision

Precision is the percentage of positive instances out of the total predicted positive instances. Here, the denominator is the model prediction done as positive from the whole given dataset. Precision is a good measure to determine when the cost of false positives is high (e.g., email spam detection).

Recall

Recall (or *sensitivity* or *true positive rate*) is the percentage of positive instances out of the total actual positive instances. Therefore, the denominator (true positive + false negative) is the actual number of positive instances present in the dataset. Recall is a good measure when there is a high cost associated with false negatives (e.g., fraud detection).

In addition to accuracy, precision, and recall, some of the other commonly used evaluation metrics for classification are discussed in the following sections.

Area under ROC curve

Area under ROC curve (AUC) is an evaluation metric for binary classification problems. ROC is a probability curve, and AUC represents degree or measure of separability. It tells how much the model is capable of distinguishing between classes. The higher the AUC, the better the model is at predicting zeros as zeros and ones as ones. An AUC of 0.5 means that the model has no class separation capacity whatsoever. The probabilistic interpretation of the AUC score is that if you randomly choose a positive case and a negative case, the probability that the positive case outranks the negative case according to the classifier is given by the AUC.

Confusion matrix

A confusion matrix lays out the performance of a learning algorithm. The confusion matrix is simply a square matrix that reports the counts of the true positive (TP), true negative (TN), false positive (FP), and false negative (FN) predictions of a classifier, as shown in [Figure 4-9](#).

		Predictive values	
		Positive (1)	Negative (0)
Actual values	Positive (1)	TP	FN
	Negative (0)	FP	TN

Figure 4-9. Confusion matrix

The confusion matrix is a handy presentation of the accuracy of a model with two or more classes. The table presents predictions on the *x-axis* and accuracy outcomes on the *y-axis*. The cells of the table are the number of predictions made by the model. For example, a model can predict zero or one, and each prediction may actually have been a zero or a one. Predictions for zero that were actually zero appear in the cell for

$\text{prediction} = 0$ and $\text{actual} = 0$, whereas predictions for zero that were actually one appear in the cell for $\text{prediction} = 0$ and $\text{actual} = 1$.

Selecting an evaluation metric for supervised classification

The evaluation metric for classification depends heavily on the task at hand. For example, recall is a good measure when there is a high cost associated with false negatives such as fraud detection. We will further examine these evaluation metrics in the case studies.

Model Selection

Selecting the perfect machine learning model is both an art and a science. Looking at machine learning models, there is no one solution or approach that fits all. There are several factors that can affect your choice of a machine learning model. The main criteria in most of the cases is the model performance that we discussed in the previous section. However, there are many other factors to consider while performing model selection. In the following section, we will go over all such factors, followed by a discussion of model trade-offs.

Factors for Model Selection

The factors considered for the model selection process are as follows:

Simplicity

The degree of simplicity of the model. Simplicity usually results in quicker, more scalable, and easier to understand models and results.

Training time

Speed, performance, memory usage and overall time taken for model training.

Handle nonlinearity in the data

The ability of the model to handle the nonlinear relationship between the variables.

Robustness to overfitting

The ability of the model to handle overfitting.

Size of the dataset

The ability of the model to handle large number of training examples in the dataset.

Number of features

The ability of the model to handle high dimensionality of the feature space.

Model interpretation

How explainable is the model? Model interpretability is important because it allows us to take concrete actions to solve the underlying problem.

Feature scaling

Does the model require variables to be scaled or normally distributed?

Figure 4-10 compares the supervised learning models on the factors mentioned previously and outlines a general rule-of-thumb to narrow down the search for the best machine learning algorithm⁷ for a given problem. The table is based on the advantages and disadvantages of different models discussed in the individual model section in this chapter.

	Linear regression	Logistic regression	SVM	CART	Gradient boosting	Random forest	Artificial neural network	KNN	LDA
Simplicity	✓	✓	✓	✓	✗	✗	✗	✓	✓
Training Time	✓	✓	✗	✓	✗	✗	✗	✓	✓
Handle non-linearity	✗	✗	✓	✓	✓	✓	✓	✓	✓
Robust to overfitting	✗	✗	✓	✗	✗	✓	✗	✓	✗
Large datasets	✗	✗	✗	✓	✓	✓	✓	✗	✓
Many features	✗	✗	✓	✓	✓	✓	✓	✗	✓
Model interpretation	✓	✓	✗	✓	✓	✓	✗	✓	✓
Feature scaling needed	✗	✗	✓	✗	✗	✗	✗	✗	✗

Figure 4-10. Model selection

We can see from the table that relatively simple models include linear and logistic regression and as we move towards the ensemble and ANN, the complexity increases. In terms of the training time, the linear models and CART are relatively faster to train as compared to ensemble methods and ANN.

Linear and logistic regression can't handle nonlinear relationships, while all other models can. SVM can handle the nonlinear relationship between dependent and independent variables with nonlinear kernels.

⁷ In this table we do not include AdaBoost and extra trees as their overall behavior across all the parameters are similar to Gradient Boosting and Random Forest, respectively.

SVM and random forest tend to overfit less as compared to the linear regression, logistic regression, gradient boosting, and ANN. The degree of overfitting also depends on other parameters, such as size of the data and model tuning, and can be checked by looking at the results of the test set for each model. Also, the boosting methods such as gradient boosting have higher overfitting risk compared to the bagging methods, such as random forest. Recall the focus of gradient boosting is to minimize the bias and not variance.

Linear and logistic regressions are not able to handle large datasets and large number of features well. However, CART, ensemble methods, and ANN are capable of handling large datasets and many features quite well. The linear and logistic regression generally perform better than other models in case the size of the dataset is small. Application of variable reduction techniques (shown in [Chapter 7](#)) enables the linear models to handle large datasets. The performance of ANN increases with an increase in the size of the dataset.

Given linear regression, logistic regression, and CART are relatively simpler models, they have better model interpretation as compared to the ensemble models and ANN.

Model Trade-off

Often, it's a trade-off between different factors when selecting a model. ANN, SVM, and some ensemble methods can be used to create very accurate predictive models, but they may lack simplicity and interpretability and may take a significant amount of resources to train.

In terms of selecting the final model, models with lower interpretability may be preferred when predictive performance is the most important goal, and it's not necessary to explain how the model works and makes predictions. In some cases, however, model interpretability is mandatory.

Interpretability-driven examples are often seen in the financial industry. In many cases, choosing a machine learning algorithm has less to do with the optimization or the technical aspects of the algorithm and more to do with business decisions. Suppose a machine learning algorithm is used to accept or reject an individual's credit card application. If the applicant is rejected and decides to file a complaint or take legal action, the financial institution will need to explain how that decision was made. While that can be nearly impossible for ANN, it's relatively straightforward for decision tree-based models.

Different classes of models are good at modeling different types of underlying patterns in data. So a good first step is to quickly test out a few different classes of models to know which ones capture the underlying structure of the dataset most

efficiently. We will follow this approach while performing model selection in all our supervised learning-based case studies.

Chapter Summary

In this chapter, we discussed the importance of supervised learning models in finance, followed by a brief introduction to several supervised learning models, including linear and logistic regression, SVM, decision trees, ensemble, KNN, LDA, and ANN. We demonstrated training and tuning of these models in a few lines of code using `sklearn` and `Keras` libraries.

We discussed the most common error metrics for regression and classification models, explained the bias-variance trade-off, and illustrated the various tools for managing the model selection process using cross validation.

We introduced the strengths and weaknesses of each model and discussed the factors to consider when selecting the best model. We also discussed the trade-off between model performance and interpretability.

In the following chapter, we will dive into the case studies for regression and classification. All case studies in the next two chapters leverage the concepts presented in this chapter and in the previous two chapters.

Supervised Learning: Regression (Including Time Series Models)

Supervised regression-based machine learning is a predictive form of modeling in which the goal is to model the relationship between a target and the predictor variable(s) in order to estimate a continuous set of possible outcomes. These are the most used machine learning models in finance.

One of the focus areas of analysts in financial institutions (and finance in general) is to predict investment opportunities, typically predictions of asset prices and asset returns. Supervised regression-based machine learning models are inherently suitable in this context. These models help investment and financial managers understand the properties of the predicted variable and its relationship with other variables, and help them identify significant factors that drive asset returns. This helps investors estimate return profiles, trading costs, technical and financial investment required in infrastructure, and thus ultimately the risk profile and profitability of a strategy or portfolio.

With the availability of large volumes of data and processing techniques, supervised regression-based machine learning isn't just limited to asset price prediction. These models are applied to a wide range of areas within finance, including portfolio management, insurance pricing, instrument pricing, hedging, and risk management.

In this chapter we cover three supervised regression-based case studies that span diverse areas, including asset price prediction, instrument pricing, and portfolio management. All of the case studies follow the standardized seven-step model development process presented in [Chapter 2](#); those steps include defining the problem, loading the data, exploratory data analysis, data preparation, model evaluation, and

model tuning.¹ The case studies are designed not only to cover a diverse set of topics from the finance standpoint but also to cover multiple machine learning and modeling concepts, including models from basic linear regression to advanced deep learning that were presented in [Chapter 4](#).

A substantial amount of asset modeling and prediction problems in the financial industry involve a time component and estimation of a continuous output. As such, it is also important to address *time series models*. In its broadest form, time series analysis is about inferring what has happened to a series of data points in the past and attempting to predict what will happen to it in the future. There have been a lot of comparisons and debates in academia and the industry regarding the differences between supervised regression and time series models. Most time series models are *parametric* (i.e., a known function is assumed to represent the data), while the majority of supervised regression models are *nonparametric*. Time series models primarily use historical data of the predicted variables for prediction, and supervised learning algorithms use *exogenous variables* as predictor variables.² However, supervised regression can embed the historical data of the predicted variable through a time-delay approach (covered later in this chapter), and a time series model (such as ARI-MAX, also covered later in this chapter) can use exogenous variables for prediction. Hence, time series and supervised regression models are similar in the sense that both can use exogenous variables as well as historical data of the predicted variable for forecasting. In terms of the final output, both supervised regression and time series models estimate a continuous set of possible outcomes of a variable.

In [Chapter 4](#), we covered the concepts of models that are common between supervised regression and supervised classification. Given that time series models are more closely aligned with supervised regression than supervised classification, we will go through the concepts of time series models separately in this chapter. We will also demonstrate how we can use time series models on financial data to predict future values. Comparison of time series models against the supervised regression models will be presented in the case studies. Additionally, some machine learning and deep learning models (such as LSTM) can be directly used for time series forecasting, and those will be discussed as well.

¹ There may be reordering or renaming of the steps or substeps based on the appropriateness and intuitiveness of the steps/substeps.

² An exogenous variable is one whose value is determined outside the model and imposed on the model.

In “[Case Study 1: Stock Price Prediction](#)” on page 95, we demonstrate one of the most popular prediction problems in finance, that of predicting stock returns. In addition to predicting future stock prices accurately, the purpose of this case study is to discuss the machine learning–based framework for general asset class price prediction in finance. In this case study we will discuss several machine learning and time series concepts, along with focusing on visualization and model tuning.

In “[Case Study 2: Derivative Pricing](#)” on page 114, we will delve into derivative pricing using supervised regression and show how to deploy machine learning techniques in the context of traditional quant problems. As compared to traditional derivative pricing models, machine learning techniques can lead to faster derivative pricing without relying on the several impractical assumptions. Efficient numerical computation using machine learning can be increasingly beneficial in areas such as financial risk management, where a trade-off between efficiency and accuracy is often inevitable.

In “[Case Study 3: Investor Risk Tolerance and Robo-Advisors](#)” on page 125, we illustrate supervised regression–based framework to estimate the risk tolerance of investors. In this case study, we build a robo-advisor dashboard in Python and implement this risk tolerance prediction model in the dashboard. We demonstrate how such models can lead to the automation of portfolio management processes, including the use of robo-advisors for investment management. The purpose is also to illustrate how machine learning can efficiently be used to overcome the problem of traditional risk tolerance profiling or risk tolerance questionnaires that suffer from several behavioral biases.

In “[Case Study 4: Yield Curve Prediction](#)” on page 141, we use a supervised regression–based framework to forecast different yield curve tenors simultaneously. We demonstrate how we can produce multiple tenors at the same time to model the yield curve using machine learning models.

In this chapter, we will learn about the following concepts related to supervised regression and time series techniques:

- Application and comparison of different time series and machine learning models.
- Interpretation of the models and results. Understanding the potential overfitting and underfitting and intuition behind linear versus nonlinear models.
- Performing data preparation and transformations to be used in machine learning models.
- Performing feature selection and engineering to improve model performance.
- Using data visualization and data exploration to understand outcomes.

- Algorithm tuning to improve model performance. Understanding, implementing, and tuning time series models such as ARIMA for prediction.
- Framing a problem statement related to portfolio management and behavioral finance in a regression-based machine learning framework.
- Understanding how deep learning-based models such as LSTM can be used for time series prediction.

The models used for supervised regression were presented in Chapters 3 and 4. Prior to the case studies, we will discuss time series models. We highly recommend readers turn to *Time Series Analysis and Its Applications*, 4th Edition, by Robert H. Shumway and David S. Stoffer (Springer) for a more in-depth understanding of time series concepts, and to *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, 2nd Edition, by Aurélien Géron (O'Reilly) for more on concepts in supervised regression models.



This Chapter's Code Repository

A Python-based master template for supervised regression, a time series model template, and the Jupyter notebook for all case studies presented in this chapter are included in the folder *Chapter 5 - Sup. Learning - Regression and Time Series models* of the code repository for this book.

For any new supervised regression-based case study, use the common template from the code repository, modify the elements specific to the case study, and borrow the concepts and insights from the case studies presented in this chapter. The template also includes the implementation and tuning of the ARIMA and LSTM models.³ The templates are designed to run on the cloud (i.e., Kaggle, Google Colab, and AWS). All the case studies have been designed on a uniform regression template.⁴

Time Series Models

A *time series* is a sequence of numbers that are ordered by a time index.

In this section we will cover the following aspects of time series models, which we further leverage in the case studies:

³ These models are discussed later in this chapter.

⁴ There may be reordering or renaming of the steps or substeps based on the appropriateness and intuitiveness of the steps/substeps.

- The components of a time series
- Autocorrelation and stationarity of time series
- Traditional time series models (e.g., ARIMA)
- Use of deep learning models for time series modeling
- Conversion of time series data for use in a supervised learning framework

Time Series Breakdown

A time series can be broken down into the following components:

Trend Component

A trend is a consistent directional movement in a time series. These trends will be either *deterministic* or *stochastic*. The former allows us to provide an underlying rationale for the trend, while the latter is a random feature of a series that we will be unlikely to explain. Trends often appear in financial series, and many trading models use sophisticated trend identification algorithms.

Seasonal Component

Many time series contain seasonal variation. This is particularly true in series representing business sales or climate levels. In quantitative finance we often see seasonal variation, particularly in series related to holiday seasons or annual temperature variation (such as natural gas).

We can write the components of a time series y_t as

$$y_t = S_t + T_t + R_t$$

where S_t is the seasonal component, T_t is the trend component, and R_t represents the remainder component of the time series not captured by seasonal or trend component.

The Python code for breaking down a time series (Y) into its component is as follows:

```
import statsmodels.api as sm
sm.tsa.seasonal_decompose(Y,freq=52).plot()
```

Figure 5-1 shows the time series broken down into trend, seasonality, and remainder components. Breaking down a time series into these components may help us better understand the time series and identify its behavior for better prediction.

The three time series components are shown separately in the bottom three panels. These components can be added together to reconstruct the actual time series shown in the top panel (shown as “observed”). Notice that the time series shows a trending component after 2017. Hence, the prediction model for this time series should incor-

porate the information regarding the trending behavior after 2017. In terms of seasonality there is some increase in the magnitude in the beginning of the calendar year. The residual component shown in the bottom panel is what is left over when the seasonal and trend components have been subtracted from the data. The residual component is mostly flat with some spikes and noise around 2018 and 2019. Also, each of the plots are on different scales, and the trend component has maximum range as shown by the scale on the plot.

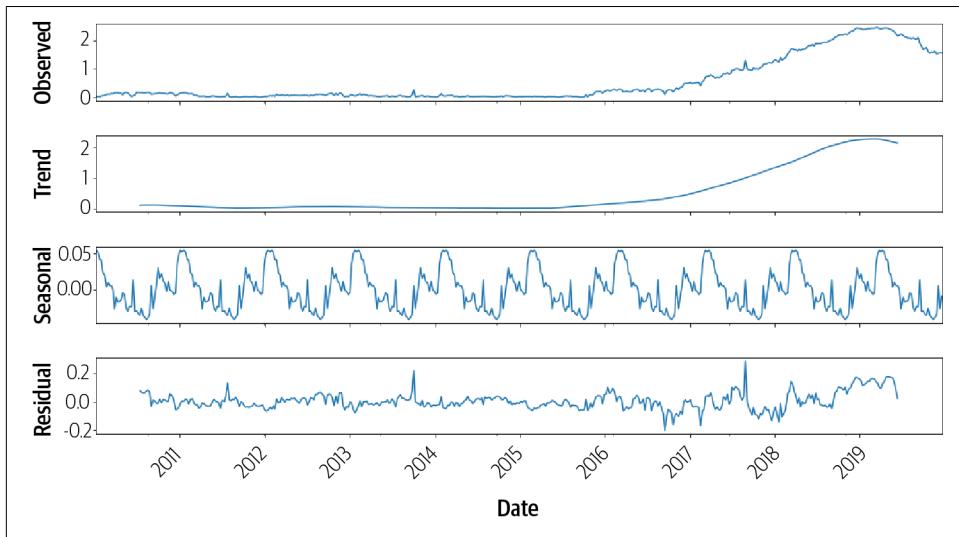


Figure 5-1. Time series components

Autocorrelation and Stationarity

When we are given one or more time series, it is relatively straightforward to decompose the time series into trend, seasonality, and residual components. However, there are other aspects that come into play when dealing with time series data, particularly in finance.

Autocorrelation

There are many situations in which consecutive elements of a time series exhibit correlation. That is, the behavior of sequential points in the series affect each other in a dependent manner. *Autocorrelation* is the similarity between observations as a function of the time lag between them. Such relationships can be modeled using an autoregression model. The term *autoregression* indicates that it is a regression of the variable against itself.

In an autoregression model, we forecast the variable of interest using a linear combination of past values of the variable.

Thus, an autoregressive model of order p can be written as

$$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \dots + \phi_p y_{t-p} + \epsilon$$

where ϵ_t is white noise.⁵ An autoregressive model is like a multiple regression but with lagged values of y_t as predictors. We refer to this as an AR(p) model, an autoregressive model of order p . Autoregressive models are remarkably flexible at handling a wide range of different time series patterns.

Stationarity

A time series is said to be stationary if its statistical properties do not change over time. Thus a time series with trend or with seasonality is not stationary, as the trend and seasonality will affect the value of the time series at different times. On the other hand, a white noise series is stationary, as it does not matter when you observe it; it should look similar at any point in time.

Figure 5-2 shows some examples of nonstationary series.

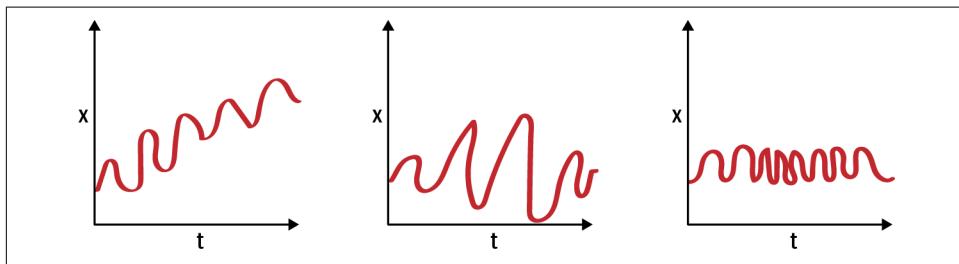


Figure 5-2. Nonstationary plots

In the first plot, we can clearly see that the mean varies (increases) with time, resulting in an upward trend. Thus this is a nonstationary series. For a series to be classified as stationary, it should not exhibit a trend. Moving on to the second plot, we certainly do not see a trend in the series, but the variance of the series is a function of time. A stationary series must have a constant variance; hence this series is a nonstationary series as well. In the third plot, the spread becomes closer as the time increases, which implies that the covariance is a function of time. The three examples shown in Figure 5-2 represent nonstationary time series. Now look at a fourth plot, as shown in Figure 5-3.

⁵ A white noise process is a random process of random variables that are uncorrelated and have a mean of zero and a finite variance.

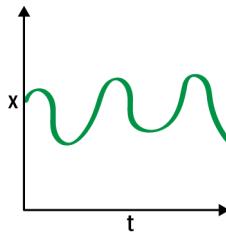


Figure 5-3. Stationary plot

In this case, the mean, variance, and covariance are constant with time. This is what a stationary time series looks like. Predicting future values using this fourth plot would be easier. Most statistical models require the series to be stationary to make effective and precise predictions.

The two major reasons behind nonstationarity of a time series are trend and seasonality, as shown in [Figure 5-2](#). In order to use time series forecasting models, we generally convert any nonstationary series to a stationary series, making it easier to model since statistical properties don't change over time.

Differencing

Differencing is one of the methods used to make a time series stationary. In this method, we compute the difference of consecutive terms in the series. Differencing is typically performed to get rid of the varying mean. Mathematically, differencing can be written as:

$$y'_t = y_t - y_{t-1}$$

where y_t is the value at a time t .

When the differenced series is white noise, the original series is referred to as a non-stationary series of degree one.

Traditional Time Series Models (Including the ARIMA Model)

There are many ways to model a time series in order to make predictions. Most of the time series models aim at incorporating the trend, seasonality, and remainder components while addressing the autocorrelation and stationarity embedded in the time series. For example, the autoregressive (AR) model discussed in the previous section addresses the autocorrelation in the time series.

One of the most widely used models in time series forecasting is the ARIMA model.

ARIMA

If we combine stationarity with autoregression and a moving average model (discussed further on in this section), we obtain an ARIMA model. ARIMA is an acronym for AutoRegressive Integrated Moving Average, and it has the following components:

$AR(p)$

It represents autoregression, i.e., regression of the time series onto itself, as discussed in the previous section, with an assumption that current series values depend on its previous values with some lag (or several lags). The maximum lag in the model is referred to as p .

$I(d)$

It represents order of integration. It is simply the number of differences needed to make the series stationary.

$MA(q)$

It represents moving average. Without going into detail, it models the error of the time series; again, the assumption is that current error depends on the previous with some lag, which is referred to as q .

The moving average equation is written as:

$$y_t = c + \epsilon_t + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2}$$

where, ϵ_t is white noise. We refer to this as an $MA(q)$ model of order q .

Combining all the components, the full ARIMA model can be written as:

$$y'_t = c + \phi_1 y'_{t-1} + \dots + \phi_p y'_{t-p} + \theta_1 \epsilon_{t-1} + \dots + \theta_q \epsilon_{t-q} + \epsilon_t$$

where y'_t is the differenced series (it may have been differenced more than once). The predictors on the right-hand side include both lagged values of y'_t and lagged errors. We call this an ARIMA(p,d,q) model, where p is the order of the autoregressive part, d is the degree of first differencing involved, and q is the order of the moving average part. The same stationarity and invertibility conditions that are used for autoregressive and moving average models also apply to an ARIMA model.

The Python code to fit the ARIMA model of the order (1,0,0) is shown in the following:

```
from statsmodels.tsa.arima_model import ARIMA  
model=ARIMA(endog=Y_train,order=[1,0,0])
```

The family of ARIMA models has several variants, and some of them are as follows:

ARIMAX

ARIMA models with exogenous variables included. We will be using this model in case study 1.

SARIMA

“S” in this model stands for seasonal, and this model is targeted at modeling the seasonality component embedded in the time series, along with other components.

VARMA

This is the extension of the model to multivariate case, when there are many variables to be predicted simultaneously. We predict many variables simultaneously in “[Case Study 4: Yield Curve Prediction](#)” on page 141.

Deep Learning Approach to Time Series Modeling

The traditional time series models such as ARIMA are well understood and effective on many problems. However, these traditional methods also suffer from several limitations. Traditional time series models are linear functions, or simple transformations of linear functions, and they require manually diagnosed parameters, such as time dependence, and don’t perform well with corrupt or missing data.

If we look at the advancements in the field of deep learning for time series prediction, we see that *recurrent neural network* (RNN) has gained increasing attention in recent years. These methods can identify structure and patterns such as nonlinearity, can seamlessly model problems with multiple input variables, and are relatively robust to missing data. The RNN models can retain state from one iteration to the next by using their own output as input for the next step. These deep learning models can be referred to as time series models, as they can make future predictions using the data points in the past, similar to traditional time series models such as ARIMA. Therefore, there are a wide range of applications in finance where these deep learning models can be leveraged. Let us look at the deep learning models for time series forecasting.

RNNs

Recurrent neural networks (RNNs) are called “recurrent” because they perform the same task for every element of a sequence, with the output being dependent on the previous computations. RNN models have a memory, which captures information about what has been calculated so far. As shown in [Figure 5-4](#), a recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor.

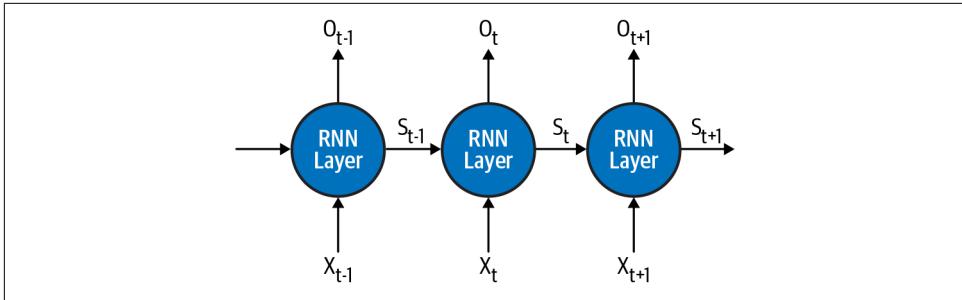


Figure 5-4. Recurrent Neural Network

In Figure 5-4:

- X_t is the input at time step t .
- O_t is the output at time step t .
- S_t is the hidden state at time step t . It's the memory of the network. It is calculated based on the previous hidden state and the input at the current step.

The main feature of an RNN is this hidden state, which captures some information about a sequence and uses it accordingly whenever needed.

Long short-term memory

Long short-term memory (LSTM) is a special kind of RNN explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically default behavior for an LSTM model.⁶ These models are composed of a set of cells with features to memorize the sequence of data. These cells capture and store the data streams. Further, the cells interconnect one module of the past to another module of the present to convey information from several past time instants to the present one. Due to the use of gates in each cell, data in each cell can be disposed, filtered, or added for the next cells.

The *gates*, based on artificial neural network layers, enable the cells to optionally let data pass through or be disposed. Each layer yields numbers in the range of zero to one, depicting the amount of every segment of data that ought to be let through in each cell. More precisely, an estimation of zero value implies “let nothing pass through.” An estimation of one indicates “let everything pass through.” Three types of gates are involved in each LSTM, with the goal of controlling the state of each cell:

⁶ A detailed explanation of LSTM models can be found in this [blog post by Christopher Olah](#).

Forget Gate

Outputs a number between zero and one, where one shows “completely keep this” and zero implies “completely ignore this.” This gate conditionally decides whether the past should be forgotten or preserved.

Input Gate

Chooses which new data needs to be stored in the cell.

Output Gate

Decides what will yield out of each cell. The yielded value will be based on the cell state along with the filtered and newly added data.

Keras wraps the efficient numerical computation libraries and functions and allows us to define and train LSTM neural network models in a few short lines of code. In the following code, LSTM module from `keras.layers` is used for implementing LSTM network. The network is trained with the variable `X_train_LSTM`. The network has a hidden layer with 50 LSTM blocks or neurons and an output layer that makes a single value prediction. Also refer to [Chapter 3](#) for a more detailed description of all the terms (i.e., sequential, learning rate, momentum, epoch, and batch size).

A sample Python code for implementing an LSTM model in Keras is shown below:

```
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from keras.layers import LSTM

def create_LSTMmodel(learn_rate = 0.01, momentum=0):
    # create model
    model = Sequential()
    model.add(LSTM(50, input_shape=(X_train_LSTM.shape[1],\
        X_train_LSTM.shape[2])))
    #More number of cells can be added if needed
    model.add(Dense(1))
    optimizer = SGD(lr=learn_rate, momentum=momentum)
    model.compile(loss='mse', optimizer='adam')
    return model
LSTMModel = create_LSTMmodel(learn_rate = 0.01, momentum=0)
LSTMModel_fit = LSTMModel.fit(X_train_LSTM, Y_train_LSTM, validation_data=\n    (X_test_LSTM, Y_test_LSTM), epochs=330, batch_size=72, verbose=0, shuffle=False)
```

In terms of both learning and implementation, LSTM provides considerably more options for fine-tuning compared to ARIMA models. Although deep learning models have several advantages over traditional time series models, deep learning models are more complicated and difficult to train.⁷

⁷ An ARIMA model and a Keras-based LSTM model will be demonstrated in one of the case studies.

Modifying Time Series Data for Supervised Learning Models

A time series is a sequence of numbers that are ordered by a time index. Supervised learning is where we have input variables (X) and an output variable (Y). Given a sequence of numbers for a time series dataset, we can restructure the data into a set of predictor and predicted variables, just like in a supervised learning problem. We can do this by using previous time steps as input variables and using the next time step as the output variable. Let's make this concrete with an example.

We can restructure a time series shown in the left table in [Figure 5-5](#) as a supervised learning problem by using the value at the previous time step to predict the value at the next time step. Once we've reorganized the time series dataset this way, the data would look like the table on the right.

Time step	Value	X	Y
1	10	?	10
2	11	10	11
3	18	11	18
4	15	18	15
5	20	15	20
<hr/>			<hr/>
		20	?
<hr/>			

Figure 5-5. Modifying time series for supervised learning models

We can see that the previous time step is the input (X) and the next time step is the output (Y) in our supervised learning problem. The order between the observations is preserved and must continue to be preserved when using this dataset to train a supervised model. We will delete the first and last row while training our supervised model as we don't have values for either X or Y .

In Python, the main function to help transform time series data into a supervised learning problem is the `shift()` function from the Pandas library. We will demonstrate this approach in the case studies. The use of prior time steps to predict the next time step is called the *sliding window*, *time delay*, or *lag* method.

Having discussed all the concepts of supervised learning and time series models, let us move to the case studies.

Case Study 1: Stock Price Prediction

One of the biggest challenges in finance is predicting stock prices. However, with the onset of recent advancements in machine learning applications, the field has been evolving to utilize nondeterministic solutions that learn what is going on in order to make more accurate predictions. Machine learning techniques naturally lend

themselves to stock price prediction based on historical data. Predictions can be made for a single time point ahead or for a set of future time points.

As a high-level overview, other than the historical price of the stock itself, the features that are generally useful for stock price prediction are as follows:

Correlated assets

An organization depends on and interacts with many external factors, including its competitors, clients, the global economy, the geopolitical situation, fiscal and monetary policies, access to capital, and so on. Hence, its stock price may be correlated not only with the stock price of other companies but also with other assets such as commodities, FX, broad-based indices, or even fixed income securities.

Technical indicators

A lot of investors follow technical indicators. Moving average, exponential moving average, and momentum are the most popular indicators.

Fundamental analysis

Two primary data sources to glean features that can be used in fundamental analysis include:

Performance reports

Annual and quarterly reports of companies can be used to extract or determine key metrics, such as ROE (Return on Equity) and P/E (Price-to-Earnings).

News

News can indicate upcoming events that can potentially move the stock price in a certain direction.

In this case study, we will use various supervised learning–based models to predict the stock price of Microsoft using correlated assets and its own historical data. By the end of this case study, readers will be familiar with a general machine learning approach to stock prediction modeling, from gathering and cleaning data to building and tuning different models.

In this case study, we will focus on:

- Looking at various machine learning and time series models, ranging in complexity, that can be used to predict stock returns.
- Visualization of the data using different kinds of charts (i.e., density, correlation, scatterplot, etc.)
- Using deep learning (LSTM) models for time series forecasting.

- Implementation of the grid search for time series models (i.e., ARIMA model).
- Interpretation of the results and examining potential overfitting and underfitting of the data across the models.



Blueprint for Using Supervised Learning Models to Predict a Stock Price

1. Problem definition

In the supervised regression framework used for this case study, the weekly return of Microsoft stock is the predicted variable. We need to understand what affects Microsoft stock price and incorporate as much information into the model. Out of correlated assets, technical indicators, and fundamental analysis (discussed in the section before), we will focus on correlated assets as features in this case study.⁸

For this case study, other than the historical data of Microsoft, the independent variables used are the following potentially correlated assets:

Stocks

IBM (IBM) and Alphabet (GOOGL)

Currency⁹

USD/JPY and GBP/USD

Indices

S&P 500, Dow Jones, and VIX

The dataset used for this case study is extracted from Yahoo Finance and [the FRED website](#). In addition to predicting the stock price accurately, this case study will also demonstrate the infrastructure and framework for each step of time series and supervised regression-based modeling for stock price prediction. We will use the daily closing price of the last 10 years, from 2010 onward.

⁸ Refer to “Case Study 3: Bitcoin Trading Strategy” on page 179 presented in Chapter 6 and “Case Study 1: NLP and Sentiment Analysis-Based Trading Strategies” on page 362 presented in Chapter 10 to understand the usage of technical indicators and news-based fundamental analysis as features in the price prediction.

⁹ Equity markets have trading holidays, while currency markets do not. However, the alignment of the dates across all the time series is ensured before any modeling or analysis.

2. Getting started—loading the data and Python packages

2.1. Loading the Python packages. The list of the libraries used for data loading, data analysis, data preparation, model evaluation, and model tuning are shown below. The packages used for different purposes have been segregated in the Python code that follows. The details of most of these packages and functions were provided in Chapters 2 and 4. The use of these packages will be demonstrated in different steps of the model development process.

Function and modules for the supervised regression models

```
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Lasso
from sklearn.linear_model import ElasticNet
from sklearn.tree import DecisionTreeRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.svm import SVR
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.ensemble import ExtraTreesRegressor
from sklearn.ensemble import AdaBoostRegressor
from sklearn.neural_network import MLPRegressor
```

Function and modules for data analysis and model evaluation

```
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2, f_regression
```

Function and modules for deep learning models

```
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from keras.layers import LSTM
from keras.wrappers.scikit_learn import KerasRegressor
```

Function and modules for time series models

```
from statsmodels.tsa.arima_model import ARIMA
import statsmodels.api as sm
```

Function and modules for data preparation and visualization

```
# pandas, pandas_datareader, numpy and matplotlib
import numpy as np
import pandas as pd
import pandas_datareader.data as web
from matplotlib import pyplot
```

```

from pandas.plotting import scatter_matrix
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from pandas.plotting import scatter_matrix
from statsmodels.graphics.tsaplots import plot_acf

```

2.2. Loading the data. One of the most important steps in machine learning and predictive modeling is gathering good data. The following steps demonstrate the loading of data from the Yahoo Finance and FRED websites using the Pandas DataReader function:¹⁰

```

stk_tickers = ['MSFT', 'IBM', 'GOOGL']
ccy_tickers = ['DEXJPUS', 'DEXUSUK']
idx_tickers = ['SP500', 'DJIA', 'VIXCLS']

stk_data = web.DataReader(stk_tickers, 'yahoo')
ccy_data = web.DataReader(ccy_tickers, 'fred')
idx_data = web.DataReader(idx_tickers, 'fred')

```

Next, we define our dependent (Y) and independent (X) variables. The predicted variable is the weekly return of Microsoft (MSFT). The number of trading days in a week is assumed to be five, and we compute the return using five trading days. For independent variables we use the correlated assets and the historical return of MSFT at different frequencies.

The variables used as independent variables are lagged five-day return of stocks (IBM and GOOG), currencies (USD/JPY and GBP/USD), and indices (S&P 500, Dow Jones, and VIX), along with lagged 5-day, 15-day, 30-day and 60-day return of MSFT.

The lagged five-day variables embed the time series component by using a time-delay approach, where the lagged variable is included as one of the independent variables. This step is reframing the time series data into a supervised regression-based model framework.

```

return_period = 5
Y = np.log(stk_data.loc[:, ('Adj Close', 'MSFT')]).diff(return_period).\
shift(-return_period)
Y.name = Y.name[-1]+'_pred'

X1 = np.log(stk_data.loc[:, ('Adj Close', ('GOOGL', 'IBM'))]).diff(return_period)
X1.columns = X1.columns.droplevel()
X2 = np.log(ccy_data).diff(return_period)
X3 = np.log(idx_data).diff(return_period)

X4 = pd.concat([np.log(stk_data.loc[:, ('Adj Close', 'MSFT')]).diff(i) \

```

¹⁰ In different case studies across the book we will demonstrate loading the data through different sources (e.g., CSV, and external websites like quandl).

```

for i in [return_period, return_period*3,\n
          return_period*6, return_period*12]], axis=1).dropna()\nX4.columns = ['MSFT_DT', 'MSFT_3DT', 'MSFT_6DT', 'MSFT_12DT']\n\nX = pd.concat([X1, X2, X3, X4], axis=1)\n\ndataset = pd.concat([Y, X], axis=1).dropna().iloc[:return_period, :]\nY = dataset.loc[:, Y.name]\nX = dataset.loc[:, X.columns]

```

3. Exploratory data analysis

We will look at descriptive statistics, data visualization, and time series analysis in this section.

3.1. Descriptive statistics. Let's have a look at the dataset we have:

```
dataset.head()
```

Output

	MSFT_pred	GOOGL	IBM	DEXJPUS	DEXUSUK	SP500	DJIA	VIXCLS	MSFT_DT	MSFT_3DT	MSFT_6DT	MSFT_12DT
2010-03-31	0.021	1.741e-02	-0.002	1.630e-02	0.018	0.001	0.002	0.002	-0.012	0.011	0.024	-0.050
2010-04-08	0.031	6.522e-04	-0.005	-7.166e-03	-0.001	0.007	0.000	-0.058	0.021	0.010	0.044	-0.007
2010-04-16	0.009	-2.879e-02	0.014	-1.349e-02	0.002	-0.002	0.002	0.129	0.011	0.022	0.069	0.007
2010-04-23	-0.014	-9.424e-03	-0.005	2.309e-02	-0.002	0.021	0.017	-0.100	0.009	0.060	0.059	0.047
2010-04-30	-0.079	-3.604e-02	-0.008	6.369e-04	-0.004	-0.025	-0.018	0.283	-0.014	0.007	0.031	0.069

The variable MSFT_pred is the return of Microsoft stock and is the predicted variable. The dataset contains the lagged series of other correlated stocks, currencies, and indices. Additionally, it also consists of the lagged historical returns of MSFT.

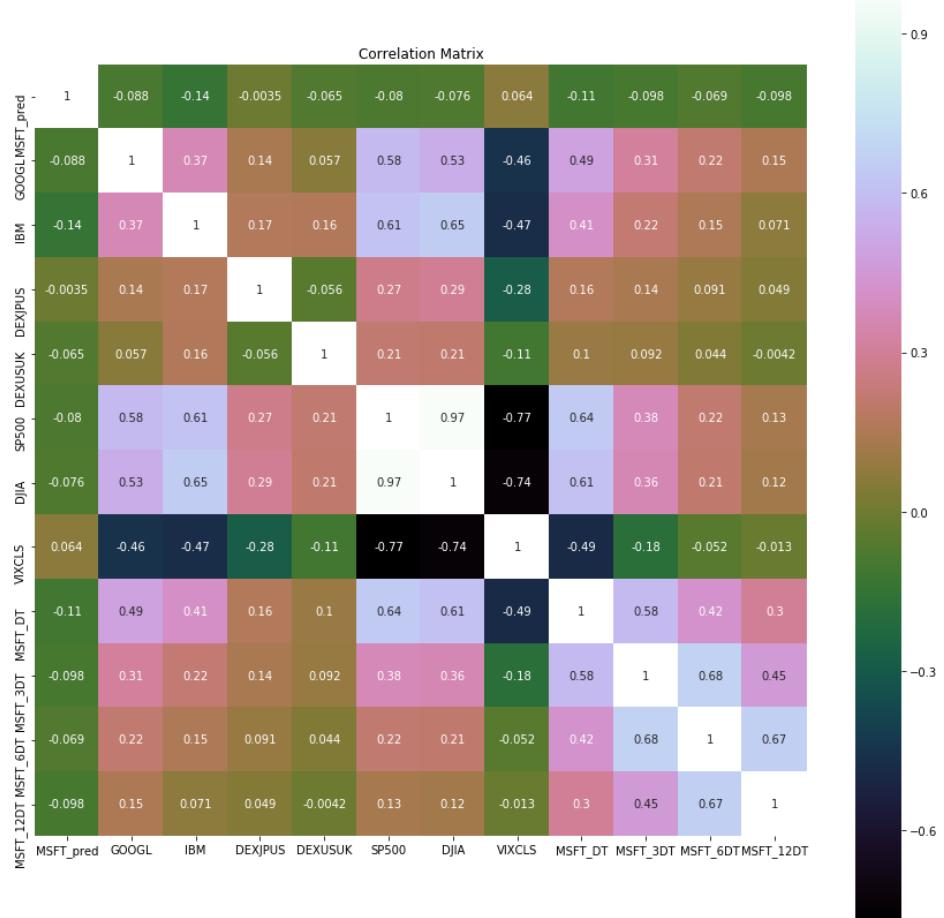
3.2. Data visualization. The fastest way to learn more about the data is to visualize it. The visualization involves independently understanding each attribute of the dataset. We will look at the scatterplot and the correlation matrix. These plots give us a sense of the interdependence of the data. Correlation can be calculated and displayed for each pair of the variables by creating a correlation matrix. Hence, besides the relationship between independent and dependent variables, it also shows the correlation among the independent variables. This is useful to know because some machine learning algorithms like linear and logistic regression can have poor performance if there are highly correlated input variables in the data:

```

correlation = dataset.corr()\npyplot.figure(figsize=(15,15))\npyplot.title('Correlation Matrix')\nsns.heatmap(correlation, vmax=1, square=True, annot=True, cmap='cubehelix')

```

Output

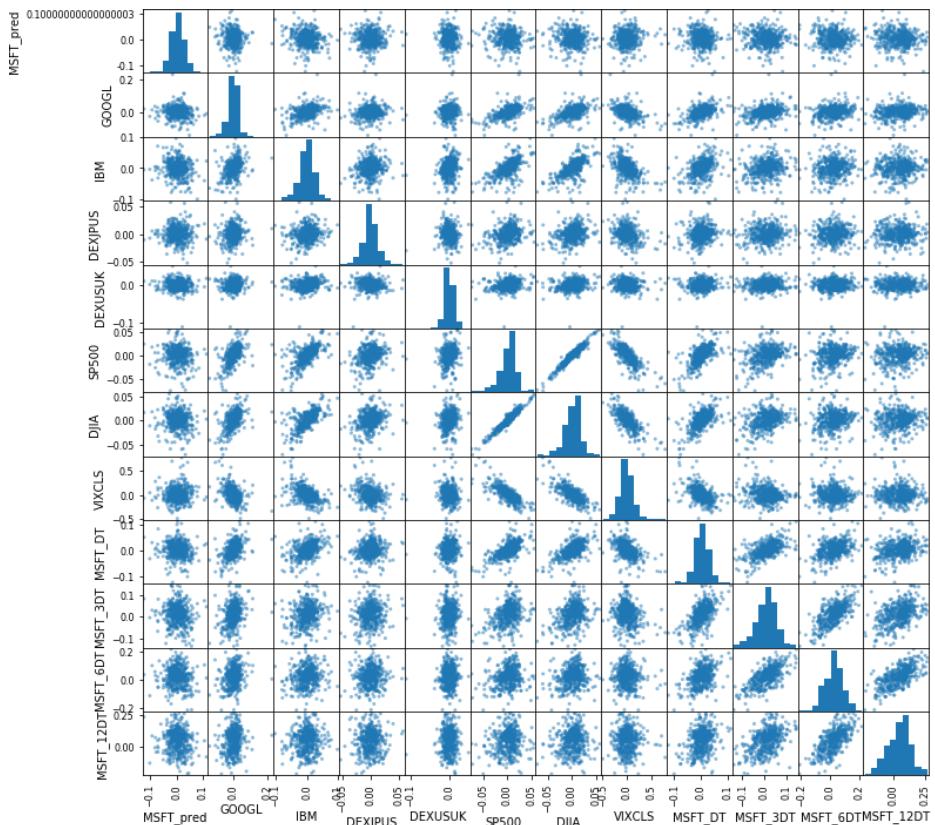


Looking at the correlation plot (full-size version available on [GitHub](#)), we see some correlation of the predicted variable with the lagged 5-day, 15-day, 30-day, and 60-day returns of MSFT. Also, we see a higher negative correlation of many asset returns versus VIX, which is intuitive.

Next, we can visualize the relationship between all the variables in the regression using the scatterplot matrix shown below:

```
pyplot.figure(figsize=(15,15))
scatter_matrix(dataset, figsize=(12,12))
pyplot.show()
```

Output

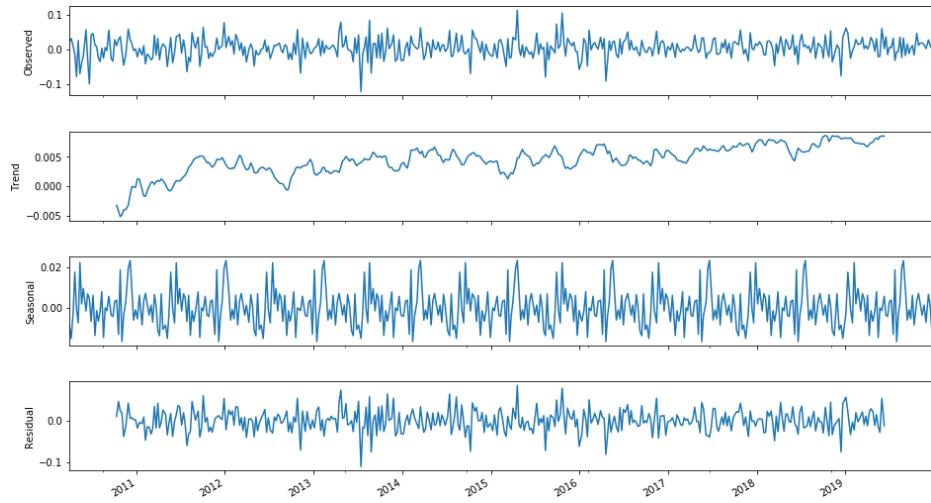


Looking at the scatterplot (full-size version available on [GitHub](#)), we see some linear relationship of the predicted variable with the lagged 15-day, 30-day, and 60-day returns of MSFT. Otherwise, we do not see any special relationship between our predicted variable and the features.

3.3. Time series analysis. Next, we delve into the time series analysis and look at the decomposition of the time series of the predicted variable into trend and seasonality components:

```
res = sm.tsa.seasonal_decompose(Y,freq=52)
fig = res.plot()
fig.set_figheight(8)
fig.set_figwidth(15)
pyplot.show()
```

Output



We can see that for MSFT there has been a general upward trend in the return series. This may be due to the large run-up of MSFT in the recent years, causing more positive weekly return data points than negative.¹¹ The trend may show up in the constant/bias terms in our models. The residual (or white noise) term is relatively small over the entire time series.

4. Data preparation

This step typically involves data processing, data cleaning, looking at feature importance, and performing feature reduction. The data obtained for this case study is relatively clean and doesn't require further processing. Feature reduction might be useful here, but given the relatively small number of variables considered, we will keep all of them as is. We will demonstrate data preparation in some of the subsequent case studies in detail.

5. Evaluate models

5.1. Train-test split and evaluation metrics. As described in [Chapter 2](#), it is a good idea to partition the original dataset into a *training set* and a *test set*. The test set is a sample of the data that we hold back from our analysis and modeling. We use it right at the end of our project to confirm the performance of our final model. It is the final test that gives us confidence on our estimates of accuracy on unseen data. We will use

¹¹ The time series is not the stock price but stock return, so the trend is mild compared to the stock price series.

80% of the dataset for modeling and use 20% for testing. With time series data, the sequence of values is important. So we do not distribute the dataset into training and test sets in random fashion, but we select an arbitrary split point in the ordered list of observations and create two new datasets:

```
validation_size = 0.2
train_size = int(len(X) * (1-validation_size))
X_train, X_test = X[0:train_size], X[train_size:len(X)]
Y_train, Y_test = Y[0:train_size], Y[train_size:len(Y)]
```

5.2. Test options and evaluation metrics. To optimize the various hyperparameters of the models, we use ten-fold cross validation (CV) and recalculate the results ten times to account for the inherent randomness in some of the models and the CV process. We will evaluate algorithms using the mean squared error metric. This metric gives an idea of the performance of the supervised regression models. All these concepts, including cross validation and evaluation metrics, have been described in [Chapter 4](#):

```
num_folds = 10
scoring = 'neg_mean_squared_error'
```

5.3. Compare models and algorithms. Now that we have completed the data loading and designed the test harness, we need to choose a model.

5.3.1. Machine learning models from Scikit-learn. In this step, the supervised regression models are implemented using the sklearn package:

Regression and tree regression algorithms

```
models = []
models.append(('LR', LinearRegression()))
models.append(('LASSO', Lasso()))
models.append(('EN', ElasticNet()))
models.append(('KNN', KNeighborsRegressor()))
models.append(('CART', DecisionTreeRegressor()))
models.append(('SVR', SVR()))
```

Neural network algorithms

```
models.append(('MLP', MLPRegressor()))
```

Ensemble models

```
# Boosting methods
models.append(('ABR', AdaBoostRegressor()))
models.append(('GBR', GradientBoostingRegressor()))
# Bagging methods
models.append(('RFR', RandomForestRegressor()))
models.append(('ETR', ExtraTreesRegressor()))
```

Once we have selected all the models, we loop over each of them. First, we run the k -fold analysis. Next, we run the model on the entire training and testing dataset.

All the algorithms use default tuning parameters. We will calculate the mean and standard deviation of the evaluation metric for each algorithm and collect the results for model comparison later:

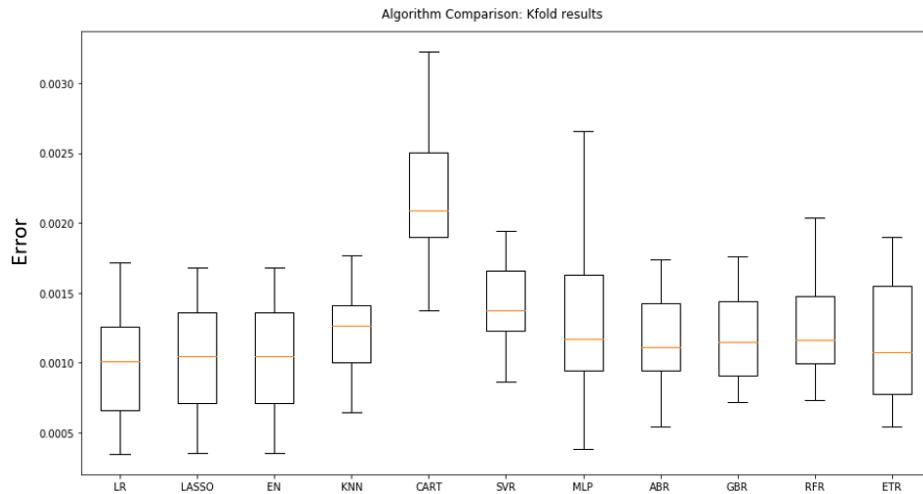
```
names = []
kfold_results = []
test_results = []
train_results = []
for name, model in models:
    names.append(name)
    ## k-fold analysis:
    kfold = KFold(n_splits=num_folds, random_state=seed)
    #converted mean squared error to positive. The lower the better
    cv_results = -1* cross_val_score(model, X_train, Y_train, cv=kfold, \
        scoring=scoring)
    kfold_results.append(cv_results)
    # Full Training period
    res = model.fit(X_train, Y_train)
    train_result = mean_squared_error(res.predict(X_train), Y_train)
    train_results.append(train_result)
    # Test results
    test_result = mean_squared_error(res.predict(X_test), Y_test)
    test_results.append(test_result)
```

Let's compare the algorithms by looking at the cross validation results:

Cross validation results

```
fig = pyplot.figure()
fig.suptitle('Algorithm Comparison: Kfold results')
ax = fig.add_subplot(111)
pyplot.boxplot(kfold_results)
ax.set_xticklabels(names)
fig.set_size_inches(15,8)
pyplot.show()
```

Output



Although the results of a couple of the models look good, we see that the linear regression and the regularized regression including the lasso regression (LASSO) and elastic net (EN) seem to perform best. This indicates a strong linear relationship between the dependent and independent variables. Going back to the exploratory analysis, we saw a good correlation and linear relationship of the target variables with the different lagged MSFT variables.

Let us look at the errors of the test set as well:

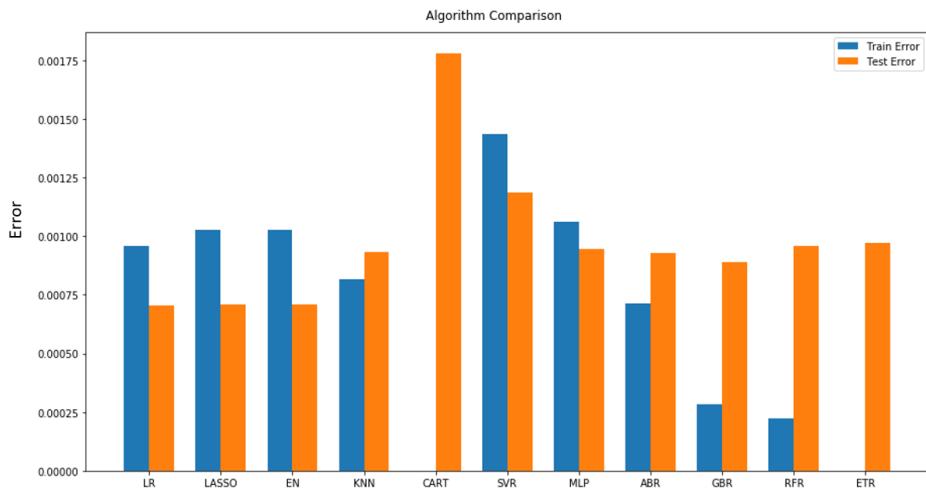
Training and test error

```
# compare algorithms
fig = pyplot.figure()

ind = np.arange(len(names)) # the x locations for the groups
width = 0.35 # the width of the bars

fig.suptitle('Algorithm Comparison')
ax = fig.add_subplot(111)
pyplot.bar(ind - width/2, train_results, width=width, label='Train Error')
pyplot.bar(ind + width/2, test_results, width=width, label='Test Error')
fig.set_size_inches(15,8)
pyplot.legend()
ax.set_xticks(ind)
ax.set_xticklabels(names)
pyplot.show()
```

Output



Examining the training and test error, we still see a stronger performance from the linear models. Some of the algorithms, such as the decision tree regressor (CART), overfit on the training data and produced very high error on the test set. Ensemble models such as gradient boosting regression (GBR) and random forest regression (RFR) have low bias but high variance. We also see that the artificial neural network algorithm (shown as MLP in the chart) shows higher errors in both the training and test sets. This is perhaps due to the linear relationship of the variables not captured accurately by ANN, improper hyperparameters, or insufficient training of the model. Our original intuition from the cross validation results and the scatterplots also seem to demonstrate a better performance of linear models.

We now look at some of the time series and deep learning models that can be used. Once we are done creating these, we will compare their performance against that of the supervised regression-based models. Due to the nature of time series models, we are not able to run a k -fold analysis. We can still compare our results to the other models based on the full training and testing results.

5.3.2. Time series-based models: ARIMA and LSTM. The models used so far already embed the time series component by using a time-delay approach, where the lagged variable is included as one of the independent variables. However, for the time series-based models we do not need the lagged variables of MSFT as the independent variables. Hence, as a first step we remove MSFT's previous returns for these models. We use all other variables as the exogenous variables in these models.

Let us first prepare the dataset for ARIMA models by having only the correlated variables as exogenous variables:

```

X_train_ARIMA=X_train.loc[:, ['GOOGL', 'IBM', 'DEXJPUS', 'SP500', 'DJIA', \
'VIXCLS']]
X_test_ARIMA=X_test.loc[:, ['GOOGL', 'IBM', 'DEXJPUS', 'SP500', 'DJIA', \
'VIXCLS']]
tr_len = len(X_train_ARIMA)
te_len = len(X_test_ARIMA)
to_len = len (X)

```

We now configure the ARIMA model with the order (1,0,0) and use the independent variables as the exogenous variables in the model. The version of the ARIMA model where the exogenous variables are also used is known as the *ARIMAX* model, where "X" represents exogenous variables:

```

modelARIMA=ARIMA(endog=Y_train,exog=X_train_ARIMA,order=[1,0,0])
model_fit = modelARIMA.fit()

```

Now we fit the ARIMA model:

```

error_Training_ARIMA = mean_squared_error(Y_train, model_fit.fittedvalues)
predicted = model_fit.predict(start = tr_len -1 ,end = to_len -1, \
    exog = X_test_ARIMA)[1:]
error_Test_ARIMA = mean_squared_error(Y_test,predicted)
error_Test_ARIMA

```

Output

```
0.0005931919240399084
```

Error of this ARIMA model is reasonable.

Now let's prepare the dataset for the LSTM model. We need the data in the form of arrays of all the input variables and the output variables.

The logic behind the LSTM is that data is taken from the previous day (the data of all the other features for that day—correlated assets and the lagged variables of MSFT) and we try to predict the next day. Then we move the one-day window with one day and again predict the next day. We iterate like this over the whole dataset (of course in batches). The code below will create a dataset in which *X* is the set of independent variables at a given time (*t*) and *Y* is the target variable at the next time (*t + 1*):

```

seq_len = 2 #Length of the seq for the LSTM

Y_train_LSTM, Y_test_LSTM = np.array(Y_train)[seq_len-1:], np.array(Y_test)
X_train_LSTM = np.zeros((X_train.shape[0]+1-seq_len, seq_len, X_train.shape[1]))
X_test_LSTM = np.zeros((X_test.shape[0], seq_len, X.shape[1]))
for i in range(seq_len):
    X_train_LSTM[:, i, :] = np.array(X_train)[i:X_train.shape[0]+i+1-seq_len, :]
    X_test_LSTM[:, i, :] = np.array(X)\n    [X_train.shape[0]+i-1:X.shape[0]+i+1-seq_len, :]

```

In the next step, we create the LSTM architecture. As we can see, the input of the LSTM is in *X_train_LSTM*, which goes into 50 hidden units in the LSTM layer and then is transformed to a single output—the stock return value. The hyperparameters

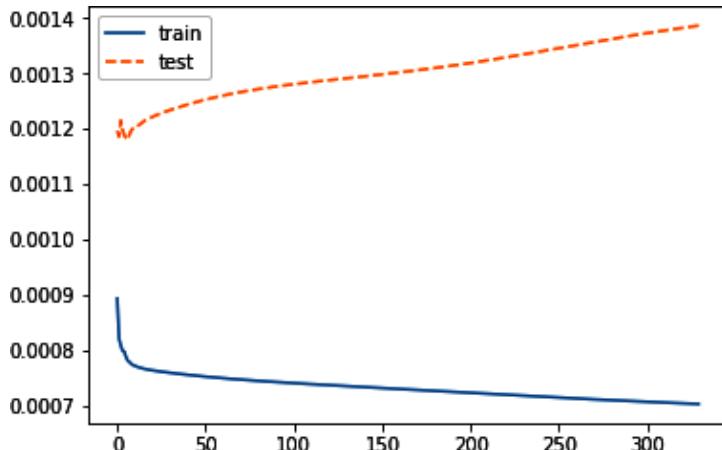
(i.e., learning rate, optimizer, activation function, etc.) were discussed in Chapter 3 of the book:

```
# LSTM Network
def create_LSTMmodel(learn_rate = 0.01, momentum=0):
    # create model
    model = Sequential()
    model.add(LSTM(50, input_shape=(X_train_LSTM.shape[1], \
        X_train_LSTM.shape[2])))
    #More cells can be added if needed
    model.add(Dense(1))
    optimizer = SGD(lr=learn_rate, momentum=momentum)
    model.compile(loss='mse', optimizer='adam')
    return model
LSTMModel = create_LSTMmodel(learn_rate = 0.01, momentum=0)
LSTMModel_fit = LSTMModel.fit(X_train_LSTM, Y_train_LSTM, \
    validation_data=(X_test_LSTM, Y_test_LSTM), \
    epochs=330, batch_size=72, verbose=0, shuffle=False)
```

Now we fit the LSTM model with the data and look at the change in the model performance metric over time simultaneously in the training set and the test set:

```
pyplot.plot(LSTMModel_fit.history['loss'], label='train', )
pyplot.plot(LSTMModel_fit.history['val_loss'], '--', label='test', )
pyplot.legend()
pyplot.show()
```

Output



```
error_Training_LSTM = mean_squared_error(Y_train_LSTM,\n    LSTMModel.predict(X_train_LSTM))\npredicted = LSTMModel.predict(X_test_LSTM)\nerror_Test_LSTM = mean_squared_error(Y_test,predicted)
```

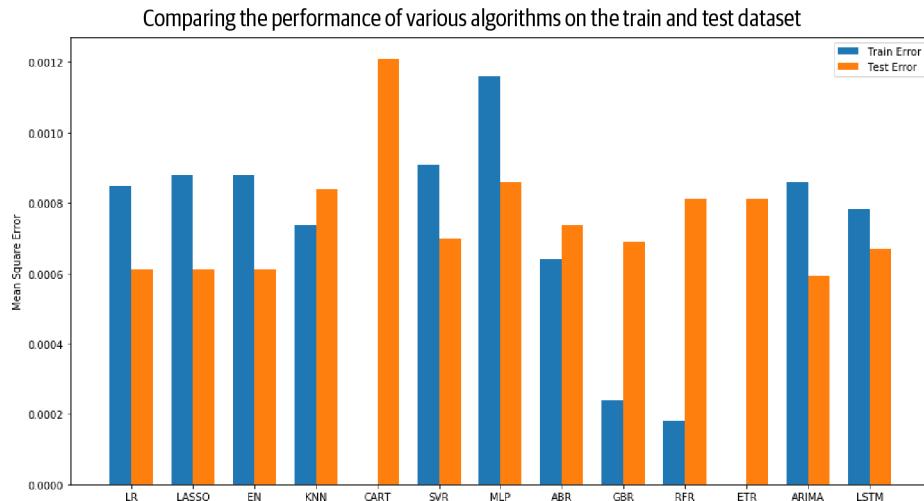
Now, in order to compare the time series and the deep learning models, we append the result of these models to the results of the supervised regression-based models:

```
test_results.append(error_Test_ARIMA)
test_results.append(error_Test_LSTM)

train_results.append(error_Training_ARIMA)
train_results.append(error_Training_LSTM)

names.append("ARIMA")
names.append("LSTM")
```

Output



Looking at the chart, we find the time series-based ARIMA model comparable to the linear supervised regression models: linear regression (LR), lasso regression (LASSO), and elastic net (EN). This can primarily be due to the strong linear relationship as discussed before. The LSTM model performs decently; however, the ARIMA model outperforms the LSTM model in the test set. Hence, we select the ARIMA model for model tuning.

6. Model tuning and grid search

Let us perform the model tuning of the ARIMA model.



Model Tuning for the Supervised Learning or Time Series Models

The detailed implementation of grid search for all the supervised learning-based models, along with the ARIMA and LSTM models, is provided in the Regression-Master template under the [GitHub repository for this book](#). For the grid search of the ARIMA and LSTM models, refer to the “ARIMA and LSTM Grid Search” section of the Regression-Master template.

The ARIMA model is generally represented as ARIMA(p,d,q) model, where p is the order of the autoregressive part, d is the degree of first differencing involved, and q is the order of the moving average part. The order of the ARIMA model was set to (1,0,0). So we perform a grid search with different p , d , and q combinations in the ARIMA model’s order and select the combination that minimizes the fitting error:

```
def evaluate_arima_model(arima_order):
    #predicted = list()
    modelARIMA=ARIMA(endog=Y_train,exog=X_train_ARIMA,order=arima_order)
    model_fit = modelARIMA.fit()
    error = mean_squared_error(Y_train, model_fit.fittedvalues)
    return error

# evaluate combinations of p, d and q values for an ARIMA model
def evaluate_models(p_values, d_values, q_values):
    best_score, best_cfg = float("inf"), None
    for p in p_values:
        for d in d_values:
            for q in q_values:
                order = (p,d,q)
                try:
                    mse = evaluate_arima_model(order)
                    if mse < best_score:
                        best_score, best_cfg = mse, order
                    print('ARIMA%s MSE=% .7f' % (order,mse))
                except:
                    continue
    print('Best ARIMA%s MSE=% .7f' % (best_cfg, best_score))

# evaluate parameters
p_values = [0, 1, 2]
d_values = range(0, 2)
q_values = range(0, 2)
warnings.filterwarnings("ignore")
evaluate_models(p_values, d_values, q_values)
```

Output

```
ARIMA(0, 0, 0) MSE=0.0009879
ARIMA(0, 0, 1) MSE=0.0009721
ARIMA(1, 0, 0) MSE=0.0009696
ARIMA(1, 0, 1) MSE=0.0009685
```

```
ARIMA(2, 0, 0) MSE=0.0009684
ARIMA(2, 0, 1) MSE=0.0009683
Best ARIMA(2, 0, 1) MSE=0.0009683
```

We see that the ARIMA model with the order $(2,0,1)$ is the best performer out of all the combinations tested in the grid search, although there isn't a significant difference in the mean squared error (MSE) with other combinations. This means that the model with the autoregressive lag of two and moving average of one yields the best result. We should not forget the fact that there are other exogenous variables in the model that influence the order of the best ARIMA model as well.

7. Finalize the model

In the last step we will check the finalized model on the test set.

7.1. Results on the test dataset.

```
# prepare model
modelARIMA_tuned=ARIMA(endog=Y_train,exog=X_train_ARIMA,order=[2,0,1])
model_fit_tuned = modelARIMA_tuned.fit()

# estimate accuracy on validation set
predicted_tuned = model_fit_tuned.predict(start = tr_len -1 ,\
    end = to_len -1, exog = X_test_ARIMA)[1:]
print(mean_squared_error(Y_test,predicted_tuned))
```

Output

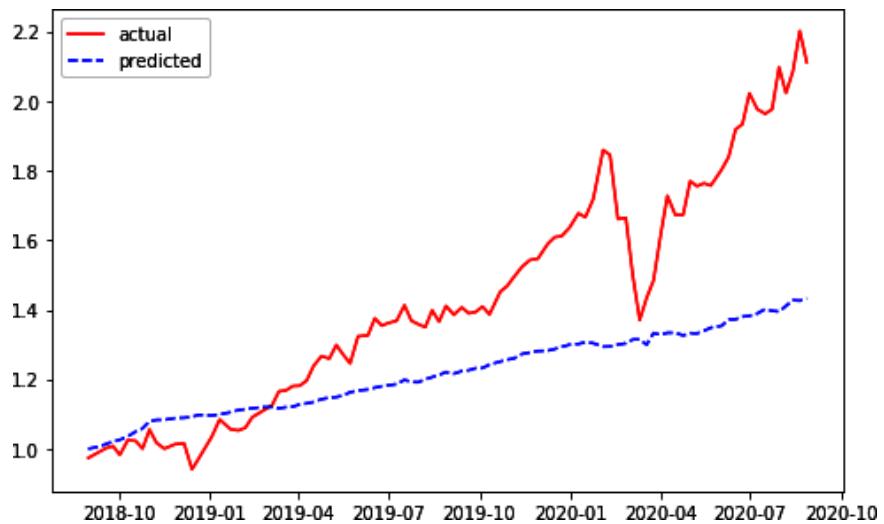
```
0.0005970582461404503
```

The MSE of the model on the test set looks good and is actually less than that of the training set.

In the last step, we will visualize the output of the selected model and compare the modeled data against the actual data. In order to visualize the chart, we convert the return time series to a price time series. We also assume the price at the beginning of the test set as one for the sake of simplicity. Let us look at the plot of actual versus predicted data:

```
# plotting the actual data versus predicted data
predicted_tuned.index = Y_test.index
pyplot.plot(np.exp(Y_test).cumprod(), 'r', label='actual',)

# plotting t, a separately
pyplot.plot(np.exp(predicted_tuned).cumprod(), 'b--', label='predicted')
pyplot.legend()
pyplot.rcParams["figure.figsize"] = (8,5)
pyplot.show()
```



Looking at the chart, we clearly see the trend has been captured perfectly by the model. The predicted series is less volatile compared to the actual time series, and it aligns with the actual data for the first few months of the test set. A point to note is that the purpose of the model is to compute the next day's return given the data observed up to the present day, and not to predict the stock price several days in the future given the current data. Hence, a deviation from the actual data is expected as we move away from the beginning of the test set. The model seems to perform well for the first few months, with deviation from the actual data increasing six to seven months after the beginning of the test set.

Conclusion

We can conclude that simple models—linear regression, regularized regression (i.e., Lasso and elastic net)—along with the time series models, such as ARIMA, are promising modeling approaches for stock price prediction problems. This approach helps us deal with overfitting and underfitting, which are some of the key challenges in predicting problems in finance.

We should also note that we can use a wider set of indicators, such as P/E ratio, trading volume, technical indicators, or news data, which might lead to better results. We will demonstrate this in some of the future case studies in the book.

Overall, we created a supervised-regression and time series modeling framework that allows us to perform stock price prediction using historical data. This framework generates results to analyze risk and profitability before risking any capital.

Case Study 2: Derivative Pricing

In computational finance and risk management, several numerical methods (e.g., finite differences, fourier methods, and Monte Carlo simulation) are commonly used for the valuation of financial derivatives.

The *Black-Scholes formula* is probably one of the most widely cited and used models in derivative pricing. Numerous variations and extensions of this formula are used to price many kinds of financial derivatives. However, the model is based on several assumptions. It assumes a specific form of movement for the derivative price, namely a *Geometric Brownian Motion* (GBM). It also assumes a conditional payment at maturity of the option and economic constraints, such as no-arbitrage. Several other derivative pricing models have similarly impractical model assumptions. Finance practitioners are well aware that these assumptions are violated in practice, and prices from these models are further adjusted using practitioner judgment.

Another aspect of the many traditional derivative pricing models is model calibration, which is typically done not by historical asset prices but by means of derivative prices (i.e., by matching the market prices of heavily traded options to the derivative prices from the mathematical model). In the process of model calibration, thousands of derivative prices need to be determined in order to fit the parameters of the model, and the overall process is time consuming. Efficient numerical computation is increasingly important in financial risk management, especially when we deal with real-time risk management (e.g., high frequency trading). However, due to the requirement of a highly efficient computation, certain high-quality asset models and methodologies are discarded during model calibration of traditional derivative pricing models.

Machine learning can potentially be used to tackle these drawbacks related to impractical model assumptions and inefficient model calibration. Machine learning algorithms have the ability to tackle more nuances with very few theoretical assumptions and can be effectively used for derivative pricing, even in a world with frictions. With the advancements in hardware, we can train machine learning models on high performance CPUs, GPUs, and other specialized hardware to achieve a speed increase of several orders of magnitude as compared to the traditional derivative pricing models.

Additionally, market data is plentiful, so it is possible to train a machine learning algorithm to learn the function that is collectively generating derivative prices in the market. Machine learning models can capture subtle nonlinearities in the data that are not obtainable through other statistical approaches.

In this case study, we look at derivative pricing from a machine learning standpoint and use a supervised regression-based model to price an option from simulated data. The main idea here is to come up with a machine learning framework for derivative pricing. Achieving a machine learning model with high accuracy would mean that we

can leverage the efficient numerical calculation of machine learning for derivative pricing with fewer underlying model assumptions.

In this case study, we will focus on:

- Developing a machine learning-based framework for derivative pricing.
- Comparison of linear and nonlinear supervised regression models in the context of derivative pricing.



Blueprint for Developing a Machine Learning Model for Derivative Pricing

1. Problem definition

In the supervised regression framework we used for this case study, the predicted variable is the price of the option, and the predictor variables are the market data used as inputs to the Black-Scholes option pricing model.

The variables selected to estimate the market price of the option are stock price, strike price, time to expiration, volatility, interest rate, and dividend yield. The predicted variable for this case study was generated using random inputs and feeding them into the well-known Black-Scholes model.¹²

The price of a call option per the Black-Scholes option pricing model is defined in [Equation 5-1](#).

Equation 5-1. Black-Scholes equation for call option

$$Se^{-q\tau}\Phi(d_1) - e^{-r\tau}K\Phi(d_2)$$

with

$$d_1 = \frac{\ln(S/K) + (r - q + \sigma^2/2)\tau}{\sigma\sqrt{\tau}}$$

and

¹² The predicted variable, which is the option price, should ideally be directly obtained for the market. Given this case study is more for demonstration purposes, we use model-generated option price for the sake of convenience.

$$d_2 = \frac{\ln(S/K) + (r - q - \sigma^2/2)\tau}{\sigma\sqrt{\tau}} = d_1 - \sigma\sqrt{\tau}$$

where we have stock price S ; strike price K ; risk-free rate r ; annual dividend yield q ; time to maturity $\tau = T - t$ (represented as a unitless fraction of one year); and volatility σ .

To make the logic simpler, we define moneyness as $M = K/S$ and look at the prices in terms of per unit of current stock price. We also set q as 0.

This simplifies the formula to the following:

$$e^{-q\tau}\Phi\left(\frac{-\ln(M) + (r + \sigma^2/2)\tau}{\sigma\sqrt{\tau}}\right) - e^{-r\tau}M\Phi\left(\frac{-\ln(M) + (r - \sigma^2/2)\tau}{\sigma\sqrt{\tau}}\right)$$

Looking at the equation above, the parameters that feed into the Black-Scholes option pricing model are moneyness, risk-free rate, volatility, and time to maturity.

The parameter that plays the central role in derivative market is volatility, as it is directly related to the movement of the stock prices. With the increase in the volatility, the range of share price movements becomes much wider than that of a low volatility stock.

In the options market, there isn't a single volatility used to price all the options. This volatility depends on the option moneyness and time to maturity. In general, the volatility increases with higher time to maturity and with moneyness. This behavior is referred to as volatility smile/skew. We often derive the volatility from the price of the options existing in the market, and this volatility is referred to as "implied" volatility. In this exercise, we assume the structure of the volatility surface and use function in [Equation 5-2](#), where volatility depends on the option moneyness and time to maturity to generate the option volatility surface.

Equation 5-2. Equation for vloatility

$$\sigma(M, \tau) = \sigma_0 + \alpha\tau + \beta(M - 1)^2$$

2. Getting started—loading the data and Python packages

2.1. Loading the Python packages. The loading of Python packages is similar to case study 1 in this chapter. Please refer to the Jupyter notebook of this case study for more details.

2.2. Defining functions and parameters. To generate the dataset, we need to simulate the input parameters and then create the predicted variable.

As a first step we define the constant parameters. The constant parameters required for the volatility surface are defined below. These parameters are not expected to have a significant impact on the option price; therefore, these parameters are set to some meaningful values:

```
true_alpha = 0.1
true_beta = 0.1
true_sigma0 = 0.2
```

The risk-free rate, which is an input to the Black-Scholes option pricing model, is defined as follows:

```
risk_free_rate = 0.05
```

Volatility and option pricing functions. In this step we define the function to compute the volatility and price of a call option as per Equations 5-1 and 5-2:

```
def option_vol_from_surface(moneyness, time_to_maturity):
    return true_sigma0 + true_alpha * time_to_maturity +\
        true_beta * np.square(moneyness - 1)

def call_option_price(moneyness, time_to_maturity, option_vol):
    d1=(np.log(1/moneyness)+(risk_free_rate+np.square(option_vol))*\
        time_to_maturity)/ (option_vol*np.sqrt(time_to_maturity))
    d2=(np.log(1/moneyness)+(risk_free_rate-np.square(option_vol))*\
        time_to_maturity)/(option_vol*np.sqrt(time_to_maturity))
    N_d1 = norm.cdf(d1)
    N_d2 = norm.cdf(d2)

    return N_d1 - moneyness * np.exp(-risk_free_rate*time_to_maturity) * N_d2
```

2.3. Data generation. We generate the input and output variables in the following steps:

- Time to maturity (T_s) is generated using the `np.random.random` function, which generates a uniform random variable between zero and one.
- Moneyness (K_s) is generated using the `np.random.randn` function, which generates a normally distributed random variable. The random number multiplied by 0.25 generates the deviation of strike from spot price,¹³ and the overall equation ensures that the moneyness is greater than zero.
- Volatility (σ) is generated as a function of time to maturity and moneyness using [Equation 5-2](#).
- The option price is generated using [Equation 5-1](#) for the Black-Scholes option price.

¹³ When the spot price is equal to the strike price, at-the-money option.

In total we generate 10,000 data points (N):

```
N = 10000

Ks = 1+0.25*np.random.randn(N)
Ts = np.random.random(N)
Sigmas = np.array([option_vol_from_surface(k,t) for k,t in zip(Ks,Ts)])
Ps = np.array([call_option_price(k,t,sig) for k,t,sig in zip(Ks,Ts,Sigmas)])
```

Now we create the variables for predicted and predictor variables:

```
Y = Ps
X = np.concatenate([Ks.reshape(-1,1), Ts.reshape(-1,1), Sigmas.reshape(-1,1)], \
axis=1)

dataset = pd.DataFrame(np.concatenate([Y.reshape(-1,1), X], axis=1),
columns=['Price', 'Moneyness', 'Time', 'Vol'])
```

3. Exploratory data analysis

Let's have a look at the dataset we have.

3.1. Descriptive statistics.

```
dataset.head()
```

Output

	Price	Moneyness	Time	Vol
0	1.390e-01	0.898	0.221	0.223
1	3.814e-06	1.223	0.052	0.210
2	1.409e-01	0.969	0.391	0.239
3	1.984e-01	0.950	0.628	0.263
4	2.495e-01	0.914	0.810	0.282

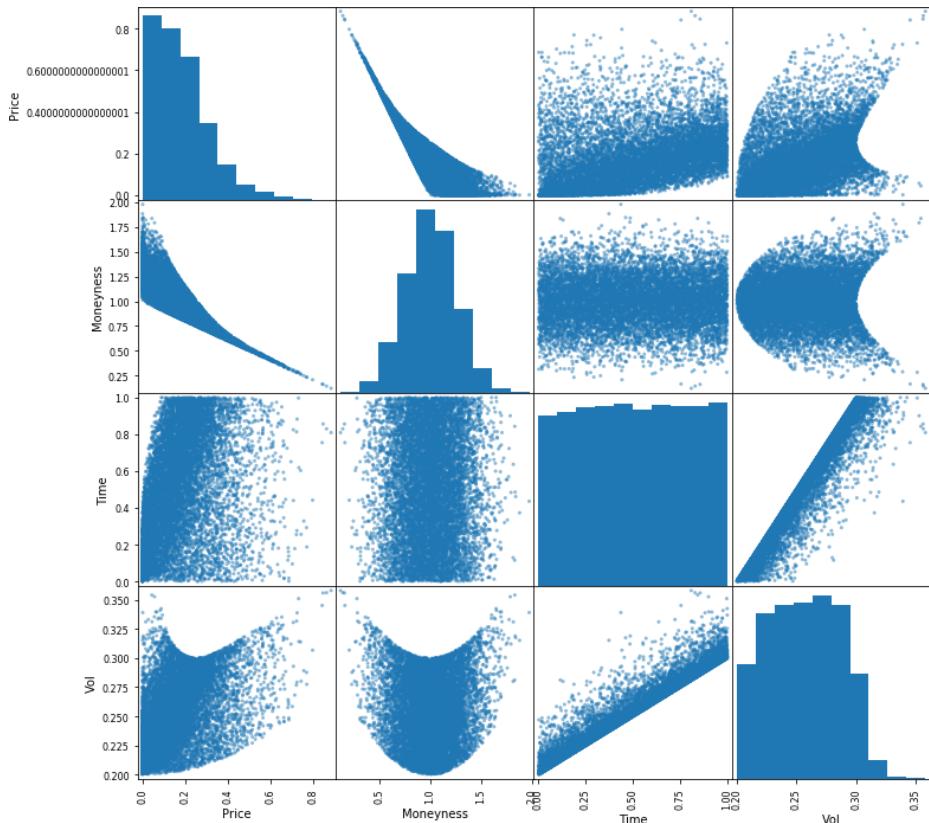
The dataset contains *price*—which is the price of the option and is the predicted variable—along with *moneyness* (the ratio of strike and spot price), *time to maturity*, and *volatility*, which are the features in the model.

3.2. Data visualization. In this step we look at scatterplot to understand the interaction between different variables:¹⁴

```
pyplot.figure(figsize=(15,15))
scatter_matrix(dataset,figsize=(12,12))
pyplot.show()
```

¹⁴ Refer to the Jupyter notebook of this case study to go through other charts such as histogram plot and correlation plot.

Output



The scatterplot reveals very interesting dependencies and relationships between the variables. Let us look at the first row of the chart to see the relationship of price to different variables. We observe that as moneyness decreases (i.e., strike price decreases as compared to the stock price), there is an increase in the price, which is in line with the rationale described in the previous section. Looking at the price versus time to maturity, we see an increase in the option price. The price versus volatility chart also shows an increase in the price with the volatility. However, option price seems to exhibit a nonlinear relationship with most of the variables. This means that we expect our nonlinear models to do a better job than our linear models.

Another interesting relationship is between volatility and strike. The more we deviate from the moneyness of one, the higher the volatility we observe. This behavior is shown due to the volatility function we defined before and illustrates the volatility smile/skew.

4. Data preparation and analysis

We performed most of the data preparation steps (i.e., getting the dependent and independent variables) in the preceding sections. In this step we look at the feature importance.

4.1. Univariate feature selection. We start by looking at each feature individually and, using the single variable regression fit as the criteria, look at the most important variables:

```
bestfeatures = SelectKBest(k='all', score_func=f_regression)
fit = bestfeatures.fit(X,Y)
dfscores = pd.DataFrame(fit.scores_)
dfcolumns = pd.DataFrame(['Moneyness', 'Time', 'Vol'])
#concat two dataframes for better visualization
featureScores = pd.concat([dfcolumns,dfscores],axis=1)
featureScores.columns = ['Specs','Score'] #naming the dataframe columns
featureScores.nlargest(10,'Score').set_index('Specs')
```

Output

```
Moneyness : 30282.309
Vol : 2407.757
Time : 1597.452
```

We observe that the moneyness is the most important variable for the option price, followed by volatility and time to maturity. Given there are only three predictor variables, we retain all the variables for modeling.

5. Evaluate models

5.1. Train-test split and evaluation metrics. First, we separate the training set and test set:

```
validation_size = 0.2

train_size = int(len(X) * (1-validation_size))
X_train, X_test = X[0:train_size], X[train_size:len(X)]
Y_train, Y_test = Y[0:train_size], Y[train_size:len(X)]
```

We use the prebuilt sklearn models to run a k -fold analysis on our training data. We then train the model on the full training data and use it for prediction of the test data. We will evaluate algorithms using the mean squared error metric. The parameters for the k -fold analysis and evaluation metrics are defined as follows:

```
num_folds = 10
seed = 7
scoring = 'neg_mean_squared_error'
```

5.2. Compare models and algorithms. Now that we have completed the data loading and have designed the test harness, we need to choose a model out of the suite of the supervised regression models.

Linear models and regression trees

```
models = []
models.append(('LR', LinearRegression()))
models.append(('KNN', KNeighborsRegressor()))
models.append(('CART', DecisionTreeRegressor()))
models.append(('SVR', SVR()))
```

Artificial neural network

```
models.append(('MLP', MLPRegressor()))
```

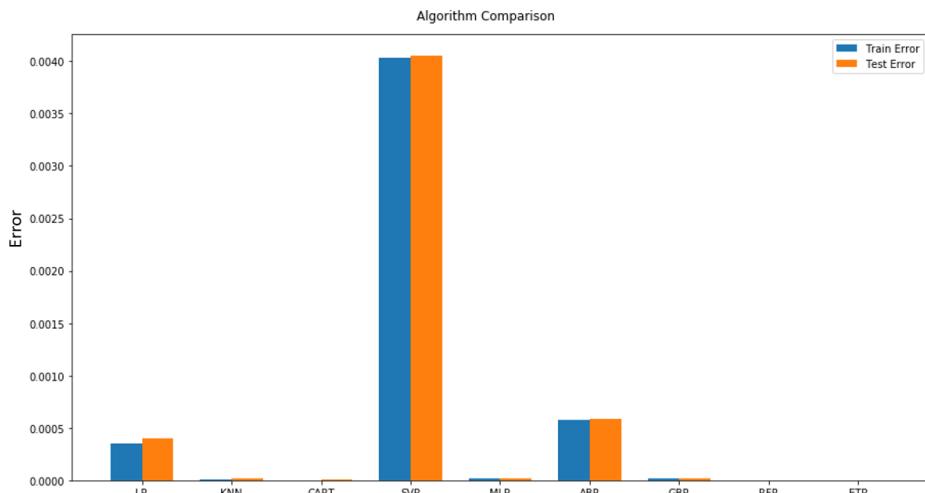
Boosting and bagging methods

```
# Boosting methods
models.append(('ABR', AdaBoostRegressor()))
models.append(('GBR', GradientBoostingRegressor()))
# Bagging methods
models.append(('RFR', RandomForestRegressor()))
models.append(('ETR', ExtraTreesRegressor()))
```

Once we have selected all the models, we loop over each of them. First, we run the k -fold analysis. Next, we run the model on the entire training and testing dataset.

The algorithms use default tuning parameters. We will calculate the mean and standard deviation of error metric and save the results for use later.

Output



The Python code for the k -fold analysis step is similar to that used in case study 1. Readers can also refer to the Jupyter notebook of this case study in the code repository for more details. Let us look at the performance of the models in the training set.

We see clearly that the nonlinear models, including classification and regression tree (CART), ensemble models, and artificial neural network (represented by MLP in the chart above), perform a lot better than the linear algorithms. This is intuitive given the nonlinear relationships we observed in the scatterplot.

Artificial neural networks (ANN) have the natural ability to model any function with fast experimentation and deployment times (definition, training, testing, inference). ANN can effectively be used in complex derivative pricing situations. Hence, out of all the models with good performance, we choose ANN for further analysis.

6. Model tuning and finalizing the model

Determining the proper number of nodes for the middle layer of an ANN is more of an art than a science, as discussed in [Chapter 3](#). Too many nodes in the middle layer, and thus too many connections, produce a neural network that memorizes the input data and lacks the ability to generalize. Therefore, increasing the number of nodes in the middle layer will improve performance on the training set, while decreasing the number of nodes in the middle layer will improve performance on a new dataset.

As discussed in [Chapter 3](#), the ANN model has several other hyperparameters such as learning rate, momentum, activation function, number of epochs, and batch size. All these hyperparameters can be tuned during the grid search process. However, in this step, we stick to performing grid search on the number of hidden layers for the purpose of simplicity. The approach to perform grid search on other hyperparameters is the same as described in the following code snippet:

```
...
hidden_layer_sizes : tuple, length = n_layers - 2, default (100,)
    The ith element represents the number of neurons in the ith
    hidden layer.
...
param_grid={'hidden_layer_sizes': [(20,), (50,), (20,20), (20, 30, 20)]}
model = MLPRegressor()
kfold = KFold(n_splits=num_folds, random_state=seed)
grid = GridSearchCV(estimator=model, param_grid=param_grid, scoring=scoring, \
    cv=kfold)
grid_result = grid.fit(X_train, Y_train)
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

Output

```
Best: -0.000024 using {'hidden_layer_sizes': (20, 30, 20)}
-0.000580 (0.000601) with: {'hidden_layer_sizes': (20,)}
-0.000078 (0.000041) with: {'hidden_layer_sizes': (50,)}
-0.000090 (0.000140) with: {'hidden_layer_sizes': (20, 20)}
-0.000024 (0.000011) with: {'hidden_layer_sizes': (20, 30, 20)}
```

The best model has three layers, with 20, 30, and 20 nodes in each hidden layer, respectively. Hence, we prepare a model with this configuration and check its performance on the test set. This is a crucial step, because a greater number of layers may lead to overfitting and have poor performance in the test set.

```
# prepare model
model_tuned = MLPRegressor(hidden_layer_sizes=(20, 30, 20))
model_tuned.fit(X_train, Y_train)

# estimate accuracy on validation set
# transform the validation dataset
predictions = model_tuned.predict(X_test)
print(mean_squared_error(Y_test, predictions))
```

Output

```
3.08127276609567e-05
```

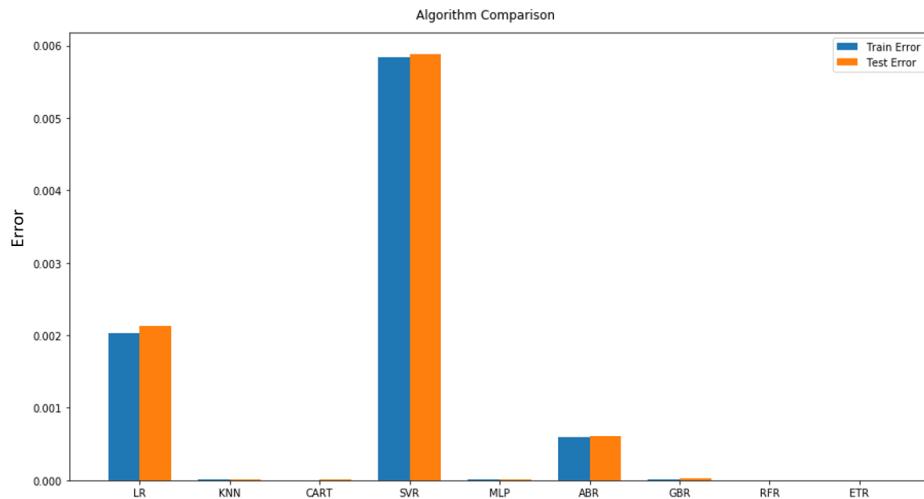
We see that the root mean squared error (RMSE) is 3.08×10^{-5} , which is less than one cent. Hence, the ANN model does an excellent job of fitting the Black-Scholes option pricing model. A greater number of layers and tuning of other hyperparameters may enable the ANN model to capture the complex relationship and nonlinearity in the data even better. Overall, the results suggest that ANN may be used to train an option pricing model that matches market prices.

7. Additional analysis: removing the volatility data

As an additional analysis, we make the process harder by trying to predict the price without the volatility data. If the model performance is good, we will eliminate the need to have a volatility function as described before. In this step, we further compare the performance of the linear and nonlinear models. In the following code snippet, we remove the volatility variable from the dataset of the predictor variable and define the training set and test set again:

```
X = X[:, :2]
validation_size = 0.2
train_size = int(len(X) * (1-validation_size))
X_train, X_test = X[0:train_size], X[train_size:len(X)]
Y_train, Y_test = Y[0:train_size], Y[train_size:len(X)]
```

Next, we run the suite of the models (except the regularized regression model) with the new dataset, with the same parameters and similar Python code as before. The performance of all the models after removing the volatility data is as follows:



Looking at the result, we have a similar conclusion as before and see a poor performance of the linear regression and good performance of the ensemble and ANN models. The linear regression now does even a worse job than before. However, the performance of ANN and other ensemble models does not deviate much from their previous performance. This implies the information of the volatility is likely captured in other variables, such as moneyness and time to maturity. Overall, it is good news as it means that fewer variables might be needed to achieve the same performance.

Conclusion

We know that derivative pricing is a nonlinear problem. As expected, our linear regression model did not do as well as our nonlinear models, and the non-linear models have a very good overall performance. We also observed that removing the volatility increases the difficulty of the prediction problem for the linear regression. However, the nonlinear models such as ensemble models and ANN are still able to do well at the prediction process. This does indicate that one might be able to sidestep the development of an option volatility surface and achieve a good prediction with a smaller number of variables.

We saw that an artificial neural network (ANN) can reproduce the Black-Scholes option pricing formula for a call option to a high degree of accuracy, meaning we can leverage efficient numerical calculation of machine learning in derivative pricing without relying on the impractical assumptions made in the traditional derivative pricing models. The ANN and the related machine learning architecture can easily be extended to pricing derivatives in the real world, with no knowledge of the theory of derivative pricing. The use of machine learning techniques can lead to much faster derivative pricing compared to traditional derivative pricing models. The price we

might have to pay for this extra speed is some loss of accuracy. However, this reduced accuracy is often well within reasonable limits and acceptable from a practical point of view. New technology has commoditized the use of ANN, so it might be worthwhile for banks, hedge funds, and financial institutions to explore these models for derivative pricing.

Case Study 3: Investor Risk Tolerance and Robo-Advisors

The risk tolerance of an investor is one of the most important inputs to the portfolio allocation and rebalancing steps of the portfolio management process. There is a wide variety of risk profiling tools that take varied approaches to understanding the risk tolerance of an investor. Most of these approaches include qualitative judgment and involve significant manual effort. In most of the cases, the risk tolerance of an investor is decided based on a risk tolerance questionnaire.

Several studies have shown that these risk tolerance questionnaires are prone to error, as investors suffer from behavioral biases and are poor judges of their own risk perception, especially during stressed markets. Also, given that these questionnaires must be manually completed by investors, they eliminate the possibility of automating the entire investment management process.

So can machine learning provide a better understanding of an investor's risk profile than a risk tolerance questionnaire can? Can machine learning contribute to automating the entire portfolio management process by cutting the client out of the loop? Could an algorithm be written to develop a personality profile for the client that would be a better representation of how they would deal with different market scenarios?

The goal of this case study is to answer these questions. We first build a supervised regression-based model to predict the risk tolerance of an investor. We then build a robo-advisor dashboard in Python and implement the risk tolerance prediction model in the dashboard. The overall purpose is to demonstrate the automation of the manual steps in the portfolio management process with the help of machine learning. This can prove to be immensely useful, specifically for robo-advisors.

A *dashboard* is one of the key features of a robo-advisor as it provides access to important information and allows users to interact with their accounts free of any human dependency, making the portfolio management process highly efficient.

Figure 5-6 provides a quick glance at the robo-advisor dashboard built for this case study. The dashboard performs end-to-end asset allocation for an investor, embedding the machine learning-based risk tolerance model constructed in this case study.

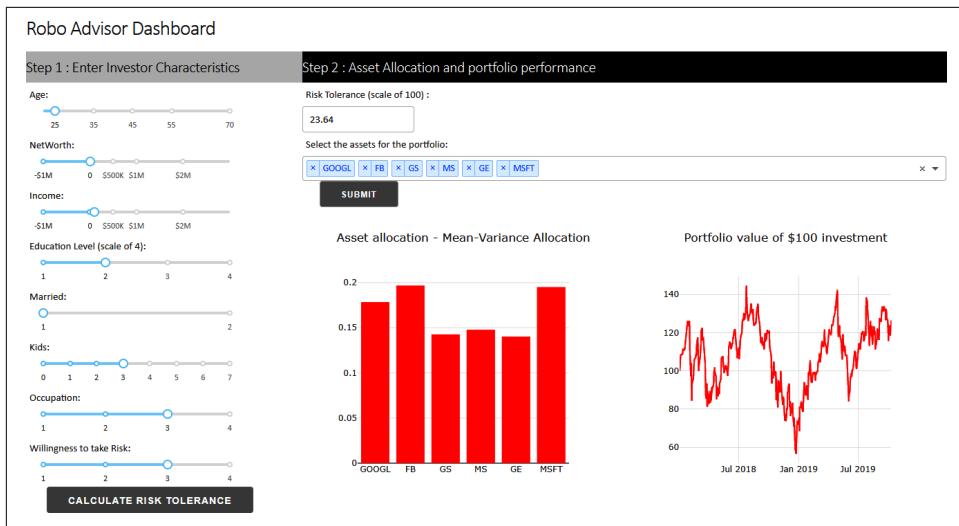


Figure 5-6. Robo-advisor dashboard

This dashboard has been built in Python and is described in detail in an additional step in this case study. Although it has been built in the context of robo-advisors, it can be extended to other areas in finance and can embed the machine learning models discussed in other case studies, providing finance decision makers with a graphical interface for analyzing and interpreting model results.

In this case study, we will focus on:

- Feature elimination and feature importance/intuition.
- Using machine learning to automate manual processes involved in portfolio management process.
- Using machine learning to quantify and model the behavioral bias of investors/individuals.
- Embedding machine learning models into user interfaces or dashboards using Python.



Blueprint for Modeling Investor Risk Tolerance and Enabling a Machine Learning-Based Robo-Advisor

1. Problem definition

In the supervised regression framework used for this case study, the predicted variable is the “true” risk tolerance of an individual,¹⁵ and the predictor variables are demographic, financial, and behavioral attributes of an individual.

The data used for this case study is from the [Survey of Consumer Finances \(SCF\)](#), which is conducted by the Federal Reserve Board. The survey includes responses about household demographics, net worth, financial, and nonfinancial assets for the same set of individuals in 2007 (precrisis) and 2009 (postcrisis). This enables us to see how each household’s allocation changed after the 2008 global financial crisis. Refer to the [data dictionary](#) for more information on this survey.

2. Getting started—loading the data and Python packages

2.1. Loading the Python packages. The details on loading the standard Python packages were presented in the previous case studies. Refer to the Jupyter notebook for this case study for more details.

2.2. Loading the data. In this step we load the data from the Survey of Consumer Finances and look at the data shape:

```
# load dataset
dataset = pd.read_excel('SCFP2009panel.xlsx')
```

Let us look at the size of the data:

```
dataset.shape
```

Output

```
(19285, 515)
```

As we can see, the dataset has a total of 19,285 observations with 515 columns. The number of columns represents the number of features.

¹⁵ Given that the primary purpose of the model is to be used in the portfolio management context, the individual is also referred to as investor in the case study.

3. Data preparation and feature selection

In this step we prepare the predicted and predictor variables to be used for modeling.

3.1. Preparing the predicted variable. In the first step, we prepare the predicted variable, which is the true risk tolerance.

The steps to compute the true risk tolerance are as follows:

1. Compute the risky assets and the risk-free assets for all the individuals in the survey data. Risky and risk-free assets are defined as follows:

Risky assets

Investments in mutual funds, stocks, and bonds.

Risk-free assets

Checking and savings balances, certificates of deposit, and other cash balances and equivalents.

2. Take the ratio of risky assets to total assets (where total assets is the sum of risky and risk-free assets) of an individual and consider that as a measure of the individual's risk tolerance.¹⁶ From the SCF, we have the data of risky and risk-free assets for the individuals in 2007 and 2009. We use this data and normalize the risky assets with the price of a stock index (S&P500) in 2007 versus 2009 to get risk tolerance.
3. Identify the “intelligent” investors. Some literature describes an intelligent investor as one who does not change their risk tolerance during changes in the market. So we consider the investors who changed their risk tolerance by less than 10% between 2007 and 2009 as the intelligent investors. Of course, this is a qualitative judgment, and there can be several other ways of defining an intelligent investor. However, as mentioned before, beyond coming up with a precise definition of true risk tolerance, the purpose of this case study is to demonstrate the usage of machine learning and provide a machine learning-based framework in portfolio management that can be further leveraged for more detailed analysis.

Let us compute the predicted variable. First, we get the risky and risk-free assets and compute the risk tolerance for 2007 and 2009 in the following code snippet:

```
# Compute the risky assets and risk-free assets for 2007
dataset['RiskFree07']= dataset['LIQ07'] + dataset['CDS07'] + dataset['SAVBND07']\
+ dataset['CASHLI07']
dataset['Risky07'] = dataset['NMMF07'] + dataset['STOCKS07'] + dataset['BOND07']
```

¹⁶ There potentially can be several ways of computing the risk tolerance. In this case study, we use the intuitive ways to measure the risk tolerance of an individual.

```

# Compute the risky assets and risk-free assets for 2009
dataset['RiskFree09']= dataset['LIQ09'] + dataset['CDS09'] + dataset['SAVBND09']\ 
+ dataset['CASHLI09']
dataset['Risky09'] = dataset['NMMF09'] + dataset['STOCKS09'] + dataset['BOND09']

# Compute the risk tolerance for 2007
dataset['RT07'] = dataset['Risky07']/(dataset['Risky07']+dataset['RiskFree07'])

#Average stock index for normalizing the risky assets in 2009
Average_SP500_2007=1478
Average_SP500_2009=948

# Compute the risk tolerance for 2009
dataset['RT09'] = dataset['Risky09']/(dataset['Risky09']+dataset['RiskFree09'])*\
(Average_SP500_2009/Average_SP500_2007)

```

Let us look at the details of the data:

```
dataset.head()
```

Output

	YY1	Y1	WGT09	AGE07	AGECL07	EDUC07	EDCL07	MARRIED07	KIDS07	LIFECL07	...	1
0	1	11	11668.134198	47	3	12	2	1	0	2	...	
1	1	12	11823.456494	47	3	12	2	1	0	2	...	
2	1	13	11913.228354	47	3	12	2	1	0	2	...	
3	1	14	11929.394266	47	3	12	2	1	0	2	...	
4	1	15	11917.722907	47	3	12	2	1	0	2	...	

5 rows × 521 columns

The data above displays some of the columns out of the 521 columns of the dataset.

Let us compute the percentage change in risk tolerance between 2007 and 2009:

```
dataset['PercentageChange'] = np.abs(dataset['RT09']/dataset['RT07']-1)
```

Next, we drop the rows containing “NA” or “NaN”:

```

# Drop the rows containing NA
dataset=dataset.dropna(axis=0)

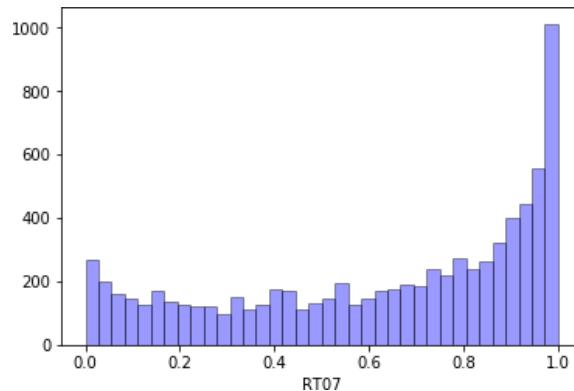
dataset=dataset[~dataset.isin([np.nan, np.inf, -np.inf]).any(1)]

```

Let us investigate the risk tolerance behavior of individuals in 2007 versus 2009. First we look at the risk tolerance in 2007:

```
sns.distplot(dataset['RT07'], hist=True, kde=False,
             bins=int(180/5), color = 'blue',
             hist_kws={'edgecolor':'black'})
```

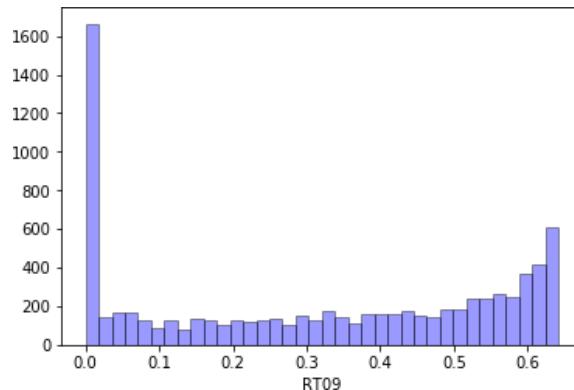
Output



Looking at the risk tolerance in 2007, we see that a significant number of individuals had a risk tolerance close to one, meaning investments were skewed more toward the risky assets. Now let us look at the risk tolerance in 2009:

```
sns.distplot(dataset['RT09'], hist=True, kde=False,  
            bins=int(180/5), color = 'blue',  
            hist_kws={'edgecolor':'black'})
```

Output



Clearly, the behavior of the individuals reversed after the crisis. Overall risk tolerance decreased, which is shown by the outsized proportion of households having risk tolerance close to zero in 2009. Most of the investments of these individuals were in risk-free assets.

In the next step, we pick the intelligent investors whose change in risk tolerance between 2007 and 2009 was less than 10%, as described in “[3.1. Preparing the predicted variable](#)” on page 128:

```
dataset3 = dataset[dataset['PercentageChange'] <= .1]
```

We assign the true risk tolerance as the average risk tolerance of these intelligent investors between 2007 and 2009:

```
dataset3['TrueRiskTolerance'] = (dataset3['RT07'] + dataset3['RT09'])/2
```

This is the predicted variable for this case study.

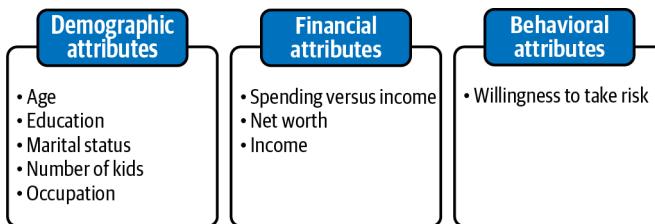
Let us drop other labels that might not be needed for the prediction:

```
dataset3.drop(labels=['RT07', 'RT09'], axis=1, inplace=True)  
dataset3.drop(labels=['PercentageChange'], axis=1, inplace=True)
```

3.2. Feature selection—limit the feature space. In this section, we will explore ways to condense the feature space.

3.2.1. Feature elimination. To filter the features further, we check the description in the [data dictionary](#) and keep only the features that are relevant.

Looking at the entire data, we have more than 500 features in the dataset. However, academic literature and industry practice indicate risk tolerance is heavily influenced by investor demographic, financial, and behavioral attributes, such as age, current income, net worth, and willingness to take risk. All these attributes were available in the dataset and are summarized in the following section. These attributes are used as features to predict investors’ risk tolerance.



In the dataset, each of the columns contains a numeric value corresponding to the value of the attribute. The details are as follows:

AGE

There are six age categories, where 1 represents age less than 35 and 6 represents age more than 75.

EDUC

There are four education categories, where 1 represents no high school and 4 represents college degree.

MARRIED

There are two categories to represent marital status, where 1 represents married and 2 represents unmarried.

OCCU

This represents occupation category. A value of 1 represents managerial status and 4 represents unemployed.

KIDS

Number of children.

WSAVED

This represents the individual's spending versus income, split into three categories. For example, 1 represents spending exceeded income.

NWCAT

This represents net worth category. There are five categories, where 1 represents net worth less than the 25th percentile and 5 represents net worth more than the 90th percentile.

INCCL

This represents income category. There are five categories, where 1 represents income less than \$10,000 and 5 represents income more than \$100,000.

RISK

This represents the willingness to take risk on a scale of 1 to 4, where 1 represents the highest level of willingness to take risk.

We keep only the intuitive features as of 2007 and remove all the intermediate features and features related to 2009, as the variables of 2007 are the only ones required for predicting the risk tolerance:

```
keep_list2 = ['AGE07', 'EDCL07', 'MARRIED07', 'KIDS07', 'OCCAT107', 'INCOME07', \
'RISK07', 'NETWORTH07', 'TrueRiskTolerance']

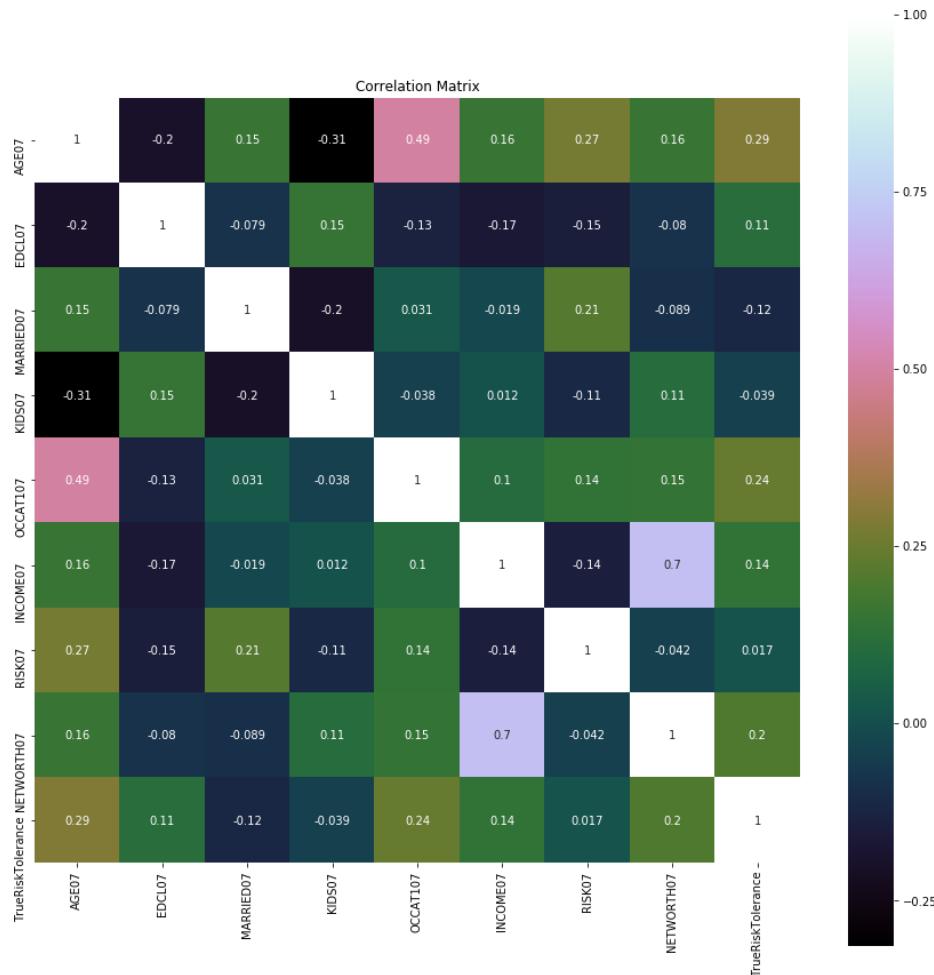
drop_list2 = [col for col in dataset3.columns if col not in keep_list2]

dataset3.drop(labels=drop_list2, axis=1, inplace=True)
```

Now let us look at the correlation among the features:

```
# correlation
correlation = dataset3.corr()
plt.figure(figsize=(15,15))
plt.title('Correlation Matrix')
sns.heatmap(correlation, vmax=1, square=True, annot=True, cmap='cubehelix')
```

Output



Looking at the correlation chart (full-size version available on [GitHub](#)), net worth and income are positively correlated with risk tolerance. With a greater number of kids and marriage, risk tolerance decreases. As the willingness to take risks decreases, the risk tolerance decreases. With age there is a positive relationship of the risk

tolerance. As per Hui Wang and Sherman Hanna's paper "Does Risk Tolerance Decrease with Age?", risk tolerance increases as people age (i.e., the proportion of net wealth invested in risky assets increases as people age) when other variables are held constant.

So in summary, the relationship of these variables with risk tolerance seems intuitive.

4. Evaluate models

4.1. Train-test split. Let us split the data into training and test set:

```
Y= dataset3["TrueRiskTolerance"]
X = dataset3.loc[:, dataset3.columns != 'TrueRiskTolerance']
validation_size = 0.2
seed = 3
X_train, X_validation, Y_train, Y_validation = \
train_test_split(X, Y, test_size=validation_size, random_state=seed)
```

4.2. Test options and evaluation metrics. We use R^2 as the evaluation metric and select 10 as the number of folds for cross validation.¹⁷

```
num_folds = 10
scoring = 'r2'
```

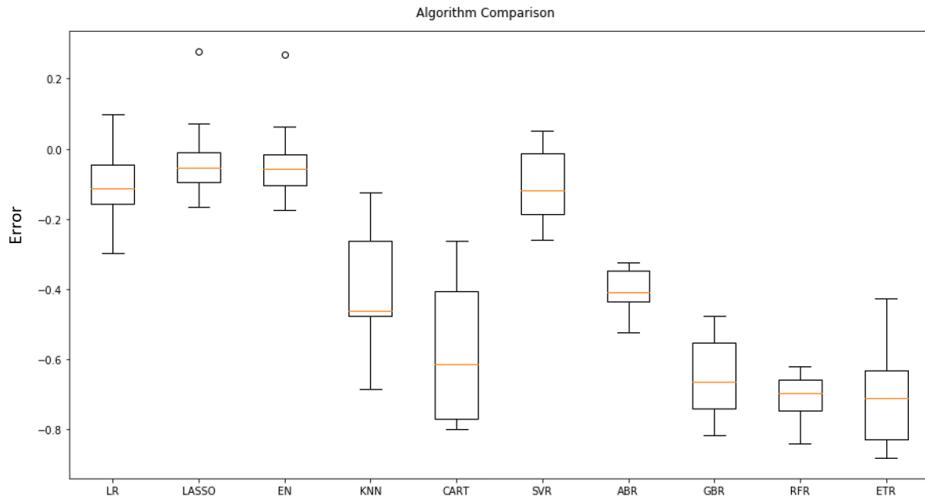
4.3. Compare models and algorithms. Next, we select the suite of the regression model and perform the k -folds cross validation.

Regression Models

```
# spot-check the algorithms
models = []
models.append(('LR', LinearRegression()))
models.append(('LASSO', Lasso()))
models.append(('EN', ElasticNet()))
models.append(('KNN', KNeighborsRegressor()))
models.append(('CART', DecisionTreeRegressor()))
models.append(('SVR', SVR()))
#Ensemble Models
# Boosting methods
models.append(('ABR', AdaBoostRegressor()))
models.append(('GBR', GradientBoostingRegressor()))
# Bagging methods
models.append(('RFR', RandomForestRegressor()))
models.append(('ETR', ExtraTreesRegressor()))
```

¹⁷ We could have chosen RMSE as the evaluation metric; however, R^2 was chosen as the evaluation metric given that we already used RMSE as the evaluation metric in the previous case studies.

The Python code for the k -fold analysis step is similar to that of previous case studies. Readers can also refer to the Jupyter notebook of this case study in the code repository for more details. Let us look at the performance of the models in the training set.



The nonlinear models perform better than the linear models, which means that there is a nonlinear relationship between the risk tolerance and the variables used to predict it. Given random forest regression is one of the best methods, we use it for further grid search.

5. Model tuning and grid search

As discussed in [Chapter 4](#), random forest has many hyperparameters that can be tweaked while performing the grid search. However, we will confine our grid search to number of estimators (`n_estimators`) as it is one of the most important hyperparameters. It represents the number of trees in the random forest model. Ideally, this should be increased until no further improvement is seen in the model:

```
# 8. Grid search : RandomForestRegressor
...
n_estimators : integer, optional (default=10)
    The number of trees in the forest.
...
param_grid = {'n_estimators': [50,100,150,200,250,300,350,400]}
model = RandomForestRegressor()
kfold = KFold(n_splits=num_folds, random_state=seed)
grid = GridSearchCV(estimator=model, param_grid=param_grid, scoring=scoring,
    cv=kfold)
grid_result = grid.fit(X_train, Y_train)
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_[ 'mean_test_score' ]
```

```
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
```

Output

```
Best: 0.738632 using {'n_estimators': 250}
```

Random forest with number of estimators as 250 is the best model after grid search.

6. Finalize the model

Let us look at the results on the test dataset and check the feature importance.

6.1. Results on the test dataset. We prepare the random forest model with the number of estimators as 250:

```
model = RandomForestRegressor(n_estimators = 250)
model.fit(X_train, Y_train)
```

Let us look at the performance in the training set:

```
from sklearn.metrics import r2_score
predictions_train = model.predict(X_train)
print(r2_score(Y_train, predictions_train))
```

Output

```
0.9640632406817223
```

The R^2 of the training set is 96%, which is a good result. Now let us look at the performance in the test set:

```
predictions = model.predict(X_validation)
print(mean_squared_error(Y_validation, predictions))
print(r2_score(Y_validation, predictions))
```

Output

```
0.007781840953471237
0.7614494526639909
```

From the mean squared error and R^2 of 76% shown above for the test set, the random forest model does an excellent job of fitting the risk tolerance.

6.2. Feature importance and features intuition

Let us look into the feature importance of the variables within the random forest model:

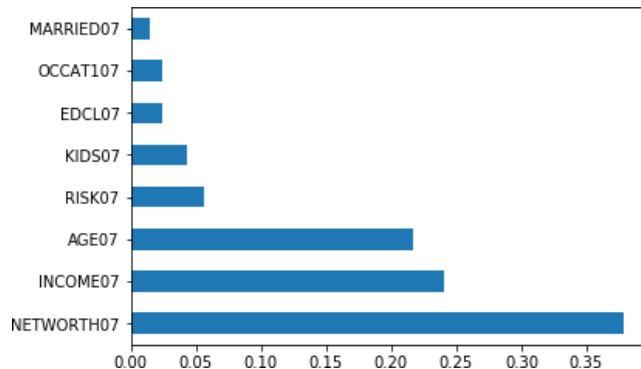
```
import pandas as pd
import numpy as np
model = RandomForestRegressor(n_estimators= 200,n_jobs=-1)
model.fit(X_train,Y_train)
#use inbuilt class feature_importances_ of tree based classifiers
#plot graph of feature importances for better visualization
```

```

feat_importances = pd.Series(model.feature_importances_, index=X.columns)
feat_importances.nlargest(10).plot(kind='barh')
plt.show()

```

Output



In the chart, the x-axis represents the magnitude of the importance of a feature. Hence, income and net worth, followed by age and willingness to take risk, are the key variables in determining risk tolerance.

6.3. Save model for later use. In this step we save the model for later use. The saved model can be used directly for prediction given the set of input variables. The model is saved as *finalized_model.sav* using the *dump* module of the pickle package. This saved model can be loaded using the *load* module.

Let's save the model as the first step:

```

# Save Model Using Pickle
from pickle import dump
from pickle import load

# save the model to disk
filename = 'finalized_model.sav'
dump(model, open(filename, 'wb'))

```

Now let's load the saved model and use it for prediction:

```

# load the model from disk
loaded_model = load(open(filename, 'rb'))
# estimate accuracy on validation set
predictions = loaded_model.predict(X_validation)
result = mean_squared_error(Y_validation, predictions)
print(r2_score(Y_validation, predictions))
print(result)

```

Output

```
0.7683894847939692  
0.007555447734714956
```

7. Additional step: robo-advisor dashboard

We mentioned the robo-advisor dashboard in the beginning of this case study. The robo-advisor dashboard performs an automation of the portfolio management process and aims to overcome the problem of traditional risk tolerance profiling.



Python Code for Robo-Advisor Dashboard

This robo-advisor dashboard is built in Python using the `plotly dash` package. `Dash` is a productive Python framework for building web applications with good user interfaces. The code for the robo-advisor dashboard is added to the [code repository for this book](#). The code is in a Jupyter notebook called “Sample Robo-advisor”. A detailed description of the code is outside the scope of this case study. However, the codebase can be leveraged for creation of any new machine learning–enabled dashboard.

The dashboard has two panels:

- Inputs for investor characteristics
- Asset allocation and portfolio performance

Input for investor characteristics. [Figure 5-7](#) shows the input panel for the investor characteristics. This panel takes all the input regarding the investor’s demographic, financial, and behavioral attributes. These inputs are for the predicted variables we used in the risk tolerance model created in the preceding steps. The interface is designed to input the categorical and continuous variables in the correct format.

Once the inputs are submitted, we leverage the model saved in “[6.3. Save model for later use](#)” on page 137. This model takes all the inputs and produces the risk tolerance of an investor (refer to the `predict_riskTolerance` function of the “Sample Robo-advisor” Jupyter notebook in the code repository for this book for more details). The risk tolerance prediction model is embedded in this dashboard and is triggered once the “Calculate Risk Tolerance” button is pressed after submitting the inputs.

Step 1 : Enter Investor Characteristics

Age:

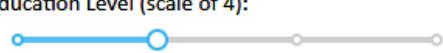
 25 35 45 55 70

NetWorth:

 -\$1M 0 \$500K \$1M \$2M

Income:

 -\$1M 0 \$500K \$1M \$2M

Education Level (scale of 4):

 1 2 3 4

Married:

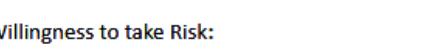
 1 2

Kids:

 0 1 2 3 4 5 6 7

Occupation:

 1 2 3 4

Willingness to take Risk:

 1 2 3 4

CALCULATE RISK TOLERANCE

Figure 5-7. Robo-advisor input panel

7.2 Asset allocation and portfolio performance. Figure 5-8 shows the “Asset Allocation and Portfolio Performance” panel, which performs the following functionalities:

- Once the risk tolerance is computed using the model, it is displayed on the top of this panel.
- In the next step, we pick the assets for our portfolio from the dropdown.
- Once the list of assets are submitted, the traditional mean-variance portfolio allocation model is used to allocate the portfolio among the assets selected. Risk

tolerance is one of the key inputs for this process. (Refer to the `get_asset_allocation` function of the “Sample Robo-advisor” Jupyter notebook in the code repository for this book for more details.)

- The dashboard also shows the historical performance of the allocated portfolio for an initial investment of \$100.

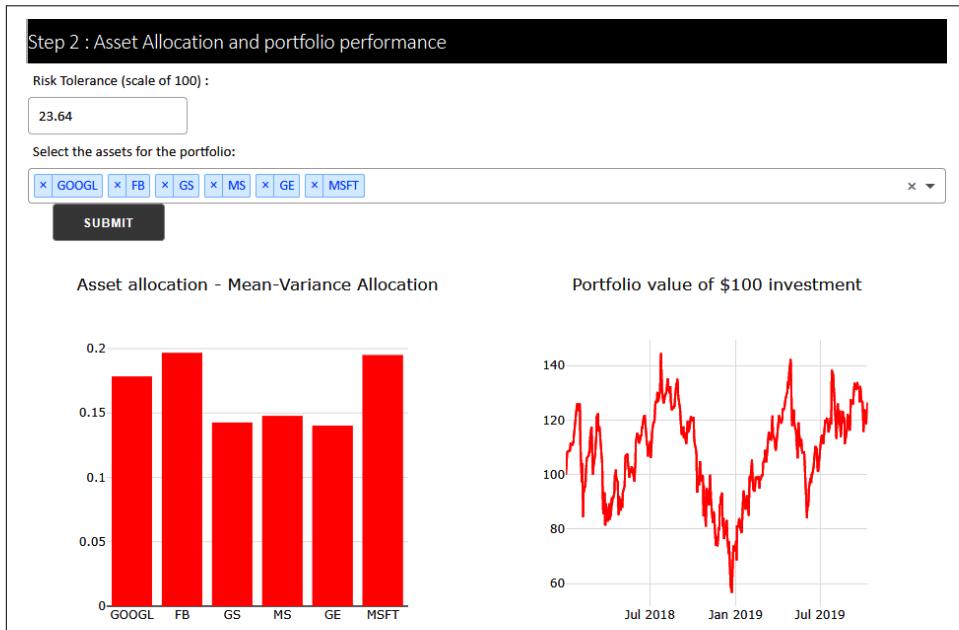


Figure 5-8. Robo-advisor asset allocation and portfolio performance panel

Although the dashboard is a basic version of the robo-advisor dashboard, it performs end-to-end asset allocation for an investor and provides the portfolio view and historical performance of the portfolio over a selected period. There are several potential enhancements to this prototype in terms of the interface and underlying models used. The dashboard can be enhanced to include additional instruments and incorporate additional features such as real-time portfolio monitoring, portfolio rebalancing, and investment advisory. In terms of the underlying models used for asset allocation, we have used the traditional mean-variance optimization method, but it can be further enhanced to use the allocation algorithms based on machine learning techniques such as eigen-portfolio, hierarchical risk parity, or reinforcement learning-based models, described in Chapters 7, 8 and 9, respectively. The risk tolerance model can be further enhanced by using additional features or using the actual data of the investors rather than using data from the Survey of Consumer Finances.

Conclusion

In this case study, we introduced the regression-based algorithm applied to compute an investor's risk tolerance, followed by a demonstration of the model in a robo-advisor setup. We showed that machine learning models might be able to objectively analyze the behavior of different investors in a changing market and attribute these changes to variables involved in determining risk appetite. With an increase in the volume of investors' data and the availability of rich machine learning infrastructure, such models might prove to be more useful than existing manual processes.

We saw that there is a nonlinear relationship between the variables and the risk tolerance. We analyzed the feature importance and found that results of the case study are quite intuitive. Income and net worth, followed by age and willingness to take risk, are the key variables to deciding risk tolerance. These variables have been considered key variables to model risk tolerance across academic and industry literature.

Through the robo-advisor dashboard powered by machine learning, we demonstrated an effective combination of data science and machine learning implementation in wealth management. Robo-advisors and investment managers could leverage such models and platforms to enhance the portfolio management process with the help of machine learning.

Case Study 4: Yield Curve Prediction

A *yield curve* is a line that plots yields (interest rates) of bonds having equal credit quality but differing maturity dates. This yield curve is used as a benchmark for other debt in the market, such as mortgage rates or bank lending rates. The most frequently reported yield curve compares the 3-months, 2-years, 5-years, 10-years, and 30-years U.S. Treasury debt.

The yield curve is the centerpiece in a fixed income market. Fixed income markets are important sources of finance for governments, national and supranational institutions, banks, and private and public corporations. In addition, yield curves are very important to investors in pension funds and insurance companies.

The yield curve is a key representation of the state of the bond market. Investors watch the bond market closely as it is a strong predictor of future economic activity and levels of inflation, which affect prices of goods, financial assets, and real estate. The slope of the yield curve is an important indicator of short-term interest rates and is followed closely by investors.

Hence, an accurate yield curve forecasting is of critical importance in financial applications. Several statistical techniques and tools commonly used in econometrics and finance have been applied to model the yield curve.

In this case study we will use supervised learning-based models to predict the yield curve. This case study is inspired by the paper *Artificial Neural Networks in Fixed Income Markets for Yield Curve Forecasting* by Manuel Nunes et al. (2018).

In this case study, we will focus on:

- Simultaneous modeling (producing multiple outputs at the same time) of the interest rates.
- Comparison of neural network versus linear regression models.
- Modeling a time series in a supervised regression-based framework.
- Understanding the variable intuition and feature selection.

Overall, the case study is similar to the stock price prediction case study presented earlier in this chapter, with the following differences:

- We predict multiple outputs simultaneously, rather than a single output.
- The predicted variable in this case study is not the return variable.
- Given that we already covered time series models in case study 1, we focus on artificial neural networks for prediction in this case study.



Blueprint for Using Supervised Learning Models to Predict the Yield Curve

1. Problem definition

In the supervised regression framework used for this case study, three tenors (1M, 5Y, and 30Y) of the yield curve are the predicted variables. These tenors represent short-term, medium-term, and long-term tenors of the yield curve.

We need to understand what affects the movement of the yield curve and hence incorporate as much information into our model as we can. As a high-level overview, other than the historical price of the yield curve itself, we look at other correlated variables that can influence the yield curve. The independent or predictor variables we consider are:

- *Previous value of the treasury curve for different tenors.* The tenors used are 1-month, 3-month, 1-year, 2-year, 5-year, 7-year, 10-year, and 30-year yields.

- *Percentage of the federal debt* held by the public, foreign governments, and the federal reserve.
- *Corporate spread* on Baa-rated debt relative to the 10-year treasury rate.

The federal debt and corporate spread are correlated variables and can be potentially useful in modeling the yield curve. The dataset used for this case study is extracted from Yahoo Finance and [FRED](#). We will use the daily data of the last 10 years, from 2010 onward.

By the end of this case study, readers will be familiar with a general machine learning approach to yield curve modeling, from gathering and cleaning data to building and tuning different models.

2. Getting started—loading the data and Python packages

2.1. Loading the Python packages. The loading of Python packages is similar to other case studies in this chapter. Refer to the Jupyter notebook of this case study for more details.

2.2. Loading the data. The following steps demonstrate the loading of data using Pandas’s `DataReader` function:

```
# Get the data by webscraping using pandas DataReader
tsy_tickers = ['DGS1MO', 'DGS3MO', 'DGS1', 'DGS2', 'DGS5', 'DGS7', 'DGS10',
               'DGS30',
               'TREAST', # Treasury securities held by the Federal Reserve ($MM)
               'FYGFDPUN', # Federal Debt Held by the Public ($MM)
               'FDHBFIN', # Federal Debt Held by International Investors ($BN)
               'GFDEBTN', # Federal Debt: Total Public Debt ($BN)
               'BAA10Y', # Baa Corporate Bond Yield Relative to Yield on 10-Year
              ]
tsy_data = web.DataReader(tsy_tickers, 'fred').dropna(how='all').ffill()
tsy_data['FDHBFIN'] = tsy_data['FDHBFIN'] * 1000
tsy_data['GOV_PCT'] = tsy_data['TREAST'] / tsy_data['GFDEBTN']
tsy_data['HOM_PCT'] = tsy_data['FYGFDPUN'] / tsy_data['GFDEBTN']
tsy_data['FOR_PCT'] = tsy_data['FDHBFIN'] / tsy_data['GFDEBTN']
```

Next, we define our dependent (Y) and independent (X) variables. The predicted variables are the rate for three tenors of the yield curve (i.e., 1M, 5Y, and 30Y) as mentioned before. The number of trading days in a week is assumed to be five, and we compute the lagged version of the variables mentioned in the problem definition section as independent variables using five trading day lag.

The lagged five-day variables embed the time series component by using a *time-delay approach*, where the lagged variable is included as one of the independent variables. This step reframes the time series data into a supervised regression-based model framework.

3. Exploratory data analysis. We will look at descriptive statistics and data visualization in this section.

3.1. Descriptive statistics. Let us look at the shape and the columns in the dataset:

```
dataset.shape
```

Output

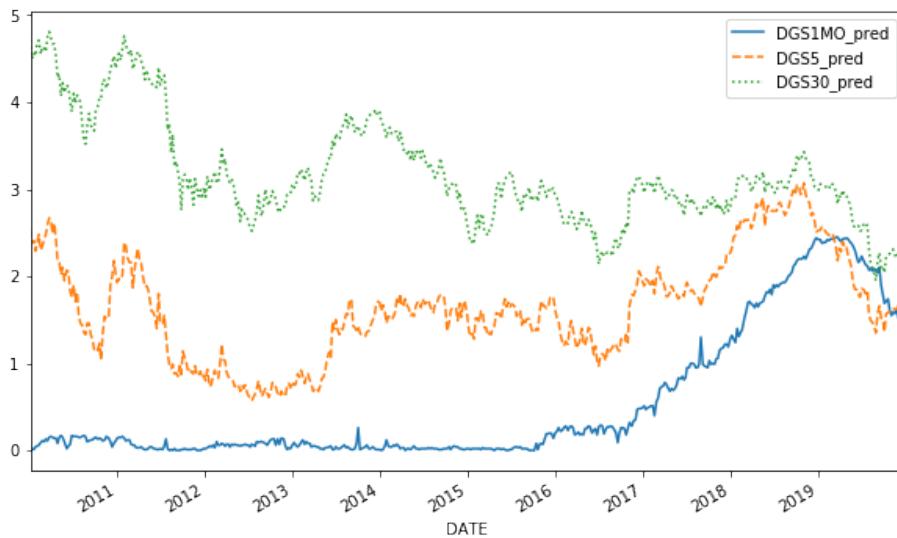
```
(505, 15)
```

The data contains around 500 observations with 15 columns.

3.2. Data visualization. Let us first plot the predicted variables and see their behavior:

```
Y.plot(style=['-', '--', ':'])
```

Output



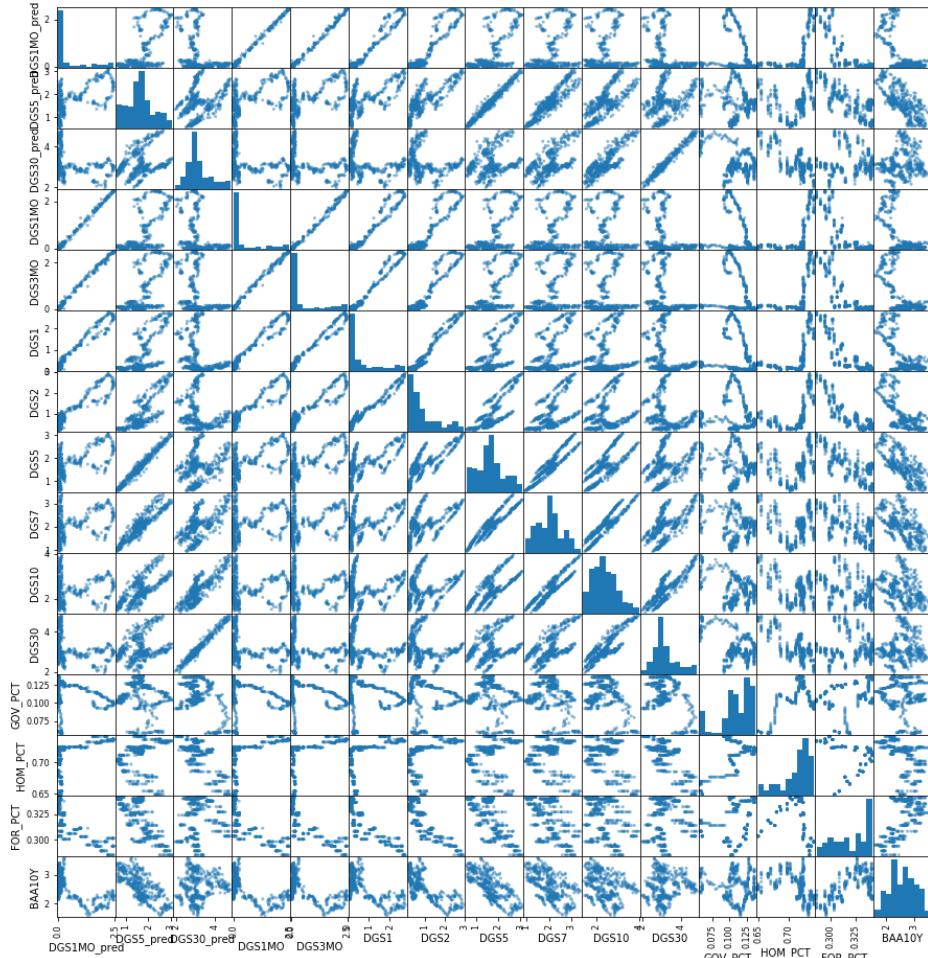
In the plot, we see that the deviation among the short-term, medium-term, and long-term rates was higher in 2010 and has been decreasing since then. There was a drop in the long-term and medium-term rates during 2011, and they also have been declining since then. The order of the rates has been in line with the tenors. However, for a few months in recent years, the 5Y rate has been lower than the 1M rate. In the time series of all the tenors, we can see that the mean varies with time, resulting in an upward trend. Thus these series are nonstationary time series.

In some cases, the linear regression for such nonstationary dependent variables might not be valid. However, we are using the lagged variables, which are also nonstationary as independent variables. So we are effectively modeling a nonstationary time series against another nonstationary time series, which might still be valid.

Next, we look at the scatterplots (a correlation plot is skipped for this case study as it has a similar interpretation to that of a scatterplot). We can visualize the relationship between all the variables in the regression using the scatter matrix shown below:

```
# Scatterplot Matrix
pyplot.figure(figsize=(15,15))
scatter_matrix(dataset,figsize=(15,16))
pyplot.show()
```

Output



Looking at the scatterplot (full-size version available on [GitHub](#)), we see a significant linear relationship of the predicted variables with their lags and other tenors of the yield curve. There is also a linear relationship, with negative slope between 1M, 5Y rates versus corporate spread and changes in foreign government purchases. The 30Y rate shows a linear relationship with these variables, although the slope is negative. Overall, we see a lot of linear relationships, and we expect the linear models to perform well.

4. Data preparation and analysis

We performed most of the data preparation steps (i.e., getting the dependent and independent variables) in the preceding steps, and so we'll skip this step.

5. Evaluate models

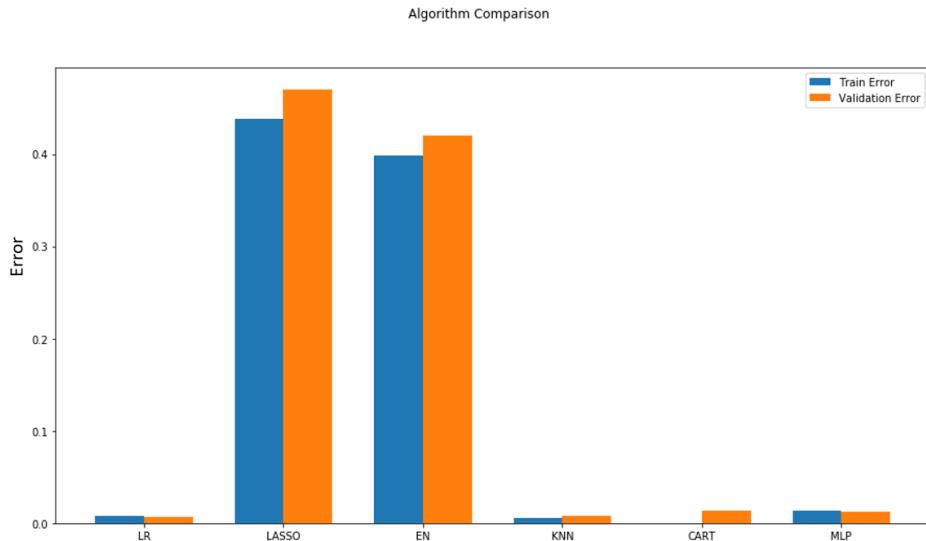
In this step we evaluate the models. The Python code for this step is similar to dthat in case study 1, and some of the repetitive code is skipped. Readers can also refer to the Jupyter notebook of this case study in the code repository for this book for more details.

5.1. Train-test split and evaluation metrics. We will use 80% of the dataset for modeling and use 20% for testing. We will evaluate algorithms using the mean squared error metric. All the algorithms use default tuning parameters.

5.2. Compare models and algorithms. In this case study, the primary purpose is to compare the linear models with the artificial neural network in yield curve modeling. So we stick to the linear regression (LR), regularized regression (LASSO and EN), and artificial neural network (shown as MLP). We also include a few other models such as KNN and CART, as these models are simpler with good interpretation, and if there is a nonlinear relationship between the variables, the CART and KNN models will be able to capture it and provide a good comparison benchmark for ANN.

Looking at the training and test error, we see a good performance of the linear regression model. We see that lasso and elastic net perform poorly. These are regularized regression models, and they reduce the number of variables in case they are not important. A decrease in the number of variables might have caused a loss of information leading to poor model performance. KNN and CART are good, but looking closely, we see that the test errors are higher than the training error. We also see that the performance of the artificial neural network (MLP) algorithm is comparable to the linear regression model. Despite its simplicity, the linear regression is a tough benchmark to beat for one-step-ahead forecasting when there is a significant linear relationship between the variables.

Output



6. Model tuning and grid search.

Similar to case study 2 of this chapter, we perform a grid search of the ANN model with different combinations of hidden layers. Several other hyperparameters such as learning rate, momentum, activation function, number of epochs, and batch size can be tuned during the grid search process, similar to the steps mentioned below.

```
...
hidden_layer_sizes : tuple, length = n_layers - 2, default (100,)
    The ith element represents the number of neurons in the ith
    hidden layer.
...
param_grid={'hidden_layer_sizes': [(20,), (50,), (20,20), (20, 30, 20)]}
model = MLPRegressor()
kfold = KFold(n_splits=num_folds, random_state=seed)
grid = GridSearchCV(estimator=model, param_grid=param_grid, scoring=scoring, \
    cv=kfold)
grid_result = grid.fit(X_train, Y_train)
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

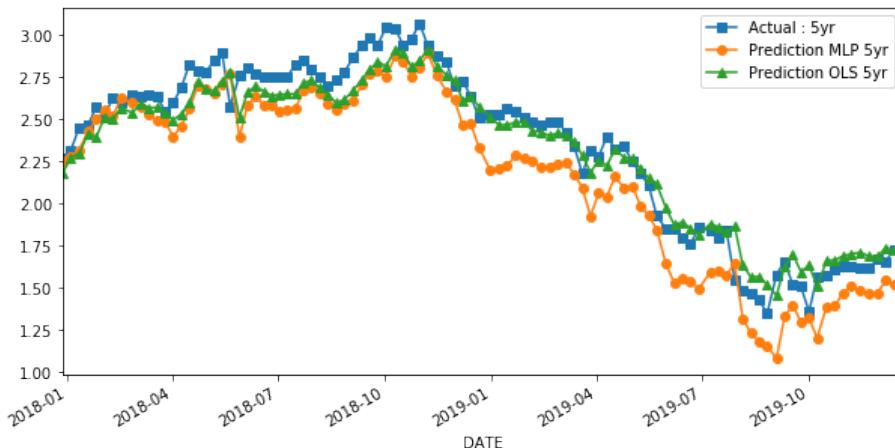
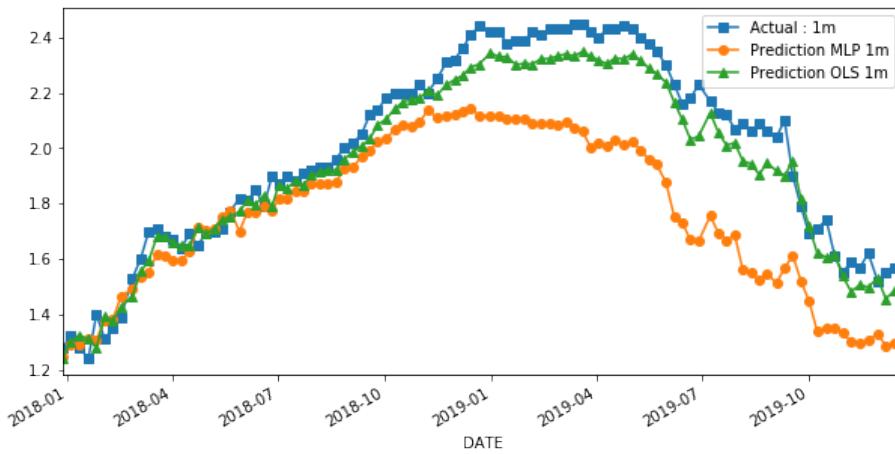
Output

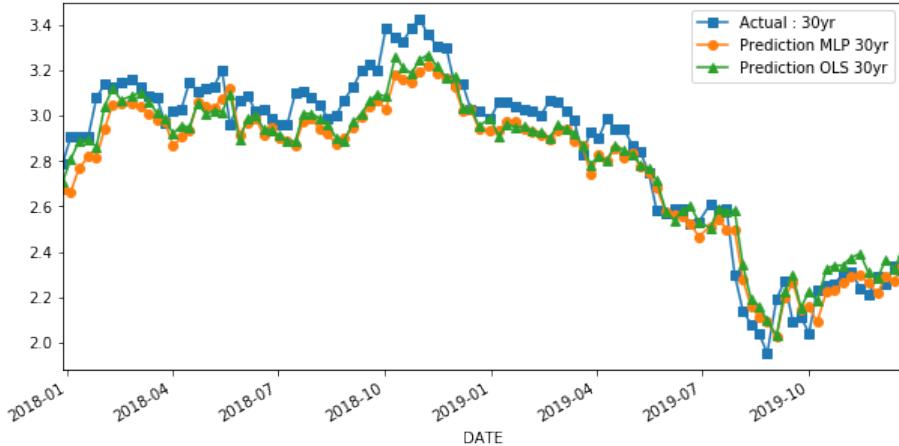
```
Best: -0.018006 using {'hidden_layer_sizes': (20, 30, 20)}
-0.036433 (0.019326) with: {'hidden_layer_sizes': (20,)}
```

```
-0.020793 (0.007075) with: {'hidden_layer_sizes': (50,)}
-0.026638 (0.010154) with: {'hidden_layer_sizes': (20, 20)}
-0.018006 (0.005637) with: {'hidden_layer_sizes': (20, 30, 20)}
```

The best model is the model with three layers, with 20, 30, and 20 nodes in each hidden layer, respectively. Hence, we prepare a model with this configuration and check its performance on the test set. This is a crucial step, as a greater number of layers may lead to overfitting and have poor performance in the test set.

Prediction comparison. In the last step we look at the prediction plot of actual data versus the prediction from both linear regression and ANN models. Refer to the Jupyter notebook of this case study for the Python code of this section.





Looking at the charts above, we see that the predictions of the linear regression and ANN are comparable. For 1M tenor, the fitting with ANN is slightly poor compared to the regression. However, for 5Y and 30Y tenors the ANN performs as well as the regression model.

Conclusion

In this case study, we applied supervised regression to the prediction of several tenors of yield curve. The linear regression model, despite its simplicity, is a tough benchmark to beat for such one-step-ahead forecasting, given the dominant characteristic of the last available value of the variable to predict. The ANN results in this case study are comparable to the linear regression models. An additional benefit of ANN is that it is more flexible to changing market conditions. Also, ANN models can be enhanced by performing grid search on several other hyperparameters and the option of incorporating recurrent neural networks, such as LSTM.

Overall, we built a machine learning-based model using ANN with an encouraging outcome, in the context of fixed income instruments. This allows us to perform predictions using historical data to generate results and analyze risk and profitability before risking any actual capital in the fixed income market.

Chapter Summary

In “[Case Study 1: Stock Price Prediction](#)” on page 95, we covered a machine learning and time series-based framework for stock price prediction. We demonstrated the significance of visualization and compared time series against the machine learning

models. In “[Case Study 2: Derivative Pricing](#)” on page 114, we explored the use of machine learning for a traditional derivative pricing problem and demonstrated a high model performance. In “[Case Study 3: Investor Risk Tolerance and Robo-Advisors](#)” on page 125, we demonstrated how supervised learning models can be used to model the risk tolerance of investors, which can lead to automation of the portfolio management process. “[Case Study 4: Yield Curve Prediction](#)” on page 141 was similar to the stock price prediction case study, providing another example of comparison of linear and nonlinear models in the context of fixed income markets.

We saw that time series and linear supervised learning models worked well for asset price prediction problems (i.e., case studies 1 and 4), where the predicted variable had a significant linear relationship with its lagged component. However, in derivative pricing and risk tolerance prediction, where there are nonlinear relationships, ensemble and ANN models performed better. Readers who are interested in implementing a case study using supervised regression or time series models are encouraged to understand the nuances in the variable relationships and model intuition before proceeding to model selection.

Overall, the concepts in Python, machine learning, time series, and finance presented in this chapter through the case studies can be used as a blueprint for any other supervised regression-based problem in finance.

Exercises

- Using the concepts and framework of machine learning and time series models specified in case study 1, develop a predictive model for another asset class—currency pair (EUR/USD, for example) or bitcoin.
- In case study 1, add some technical indicators, such as trend or momentum, and check the enhancement in the model performance. Some of the ideas of the technical indicators can be borrowed from “[Case Study 3: Bitcoin Trading Strategy](#)” on page 179 in Chapter 6.
- Using the concepts in “[Case Study 2: Derivative Pricing](#)” on page 114, develop a machine learning-based model to price [American options](#).
- Incorporate multivariate time series modeling using a variant of the ARIMA model, such as [VARMAX](#), for rates prediction in the yield curve prediction case study and compare the performance against the machine learning-based models.
- Enhance the robo-advisor dashboard presented in “[Case Study 3: Investor Risk Tolerance and Robo-Advisors](#)” on page 125 to incorporate instruments other than equities.

Supervised Learning: Classification

Here are some of the key questions that financial analysts attempt to solve:

- Is a borrower going to repay their loan or default on it?
- Will the instrument price go up or down?
- Is this credit card transaction a fraud or not?

All of these problem statements, in which the goal is to predict the categorical class labels, are inherently suitable for classification-based machine learning.

Classification-based algorithms have been used across many areas within finance that require predicting a qualitative response. These include fraud detection, default prediction, credit scoring, directional forecasting of asset price movement, and buy/sell recommendations. There are many other use cases of classification-based supervised learning in portfolio management and algorithmic trading.

In this chapter we cover three such classification-based case studies that span a diverse set of areas, including fraud detection, loan default probability, and formulating a trading strategy.

In “[Case Study 1: Fraud Detection](#)” on page 153, we use a classification-based algorithm to predict whether a transaction is fraudulent. The focus of this case study is also to deal with an unbalanced dataset, given that the fraud dataset is highly unbalanced with a small number of fraudulent observations.

In “[Case Study 2: Loan Default Probability](#)” on page 166, we use a classification-based algorithm to predict whether a loan will default. The case study focuses on various techniques and concepts of data processing, feature selection, and exploratory analysis.

In “[Case Study 3: Bitcoin Trading Strategy](#)” on page 179, we use classification-based algorithms to predict whether the current trading signal of bitcoin is to buy or sell depending on the relationship between the short-term and long-term price. We predict the trend of bitcoin’s price using technical indicators. The prediction model can easily be transformed into a trading bot that can perform buy, sell, or hold actions without human intervention.

In addition to focusing on different problem statements in finance, these case studies will help you understand:

- How to develop new features such as technical indicators for an investment strategy using feature engineering, and how to improve model performance.
- How to use data preparation and data transformation, and how to perform feature reduction and use feature importance.
- How to use data visualization and exploratory data analysis for feature reduction and to improve model performance.
- How to use algorithm tuning and grid search across various classification-based models to improve model performance.
- How to handle unbalanced data.
- How to use the appropriate evaluation metrics for classification.



This Chapter’s Code Repository

A Python-based master template for supervised classification model, along with the Jupyter notebook for the case studies presented in this chapter, is included in the folder [*Chapter 6 - Sup. Learning - Classification models*](#) in the code repository for this book. All of the case studies presented in this chapter use the standardized seven-step model development process presented in [Chapter 2](#).¹

For any new classification-based problem, the master template from the code repository can be modified with the elements specific to the problem. The templates are designed to run on cloud infrastructure (e.g., Kaggle, Google Colab, or AWS). In order to run the template on the local machine, all the packages used within the template must be installed successfully.

¹ There may be reordering or renaming of the steps or substeps based on the appropriateness and intuitiveness of the steps/substeps.

Case Study 1: Fraud Detection

Fraud is one of the most significant issues the finance sector faces. It is incredibly costly. According to one study, it is estimated that the typical organization loses 5% of its annual revenue to fraud each year. When applied to the 2017 estimated Gross World Product of \$79.6 trillion, this translates to potential global losses of up to \$4 trillion.

Fraud detection is a task inherently suitable for machine learning, as machine learning-based models can scan through huge transactional datasets, detect unusual activity, and identify all cases that might be prone to fraud. Also, the computations of these models are faster compared to traditional rule-based approaches. By collecting data from various sources and then mapping them to trigger points, machine learning solutions are able to discover the rate of defaulting or fraud propensity for each potential customer and transaction, providing key alerts and insights for the financial institutions.

In this case study, we will use various classification-based models to detect whether a transaction is a normal payment or a fraud.

The focuses of this case study are:

- Handling unbalanced data by downsampling/upsampling the data.
- Selecting the right evaluation metric, given that one of the main goals is to reduce false negatives (cases in which fraudulent transactions incorrectly go unnoticed).



Blueprint for Using Classification Models to Determine Whether a Transaction Is Fraudulent

1. Problem definition

In the classification framework defined for this case study, the response (or target) variable has the column name “Class.” This column has a value of 1 in the case of fraud and a value of 0 otherwise.

The dataset used is obtained from [Kaggle](#). This dataset holds transactions by European cardholders that occurred over two days in September 2013, with 492 cases of fraud out of 284,807 transactions.

The dataset has been anonymized for privacy reasons. Given that certain feature names are not provided (i.e., they are called V1, V2, V3, etc.), the visualization and feature importance will not give much insight into the behavior of the model.

By the end of this case study, readers will be familiar with a general approach to fraud modeling, from gathering and cleaning data to building and tuning a classifier.

2. Getting started—loading the data and Python packages

2.1. Loading the Python packages. The list of the libraries used for data loading, data analysis, data preparation, model evaluation, and model tuning are shown below. The packages used for different purposes have been separated in the Python code below. The details of most of these packages and functions have been provided in [Chapter 2](#) and [Chapter 4](#):

Packages for data loading, data analysis, and data preparation

```
import numpy as np
import pandas as pd
import seaborn as sns
from matplotlib import pyplot

from pandas import read_csv, set_option
from pandas.plotting import scatter_matrix
from sklearn.preprocessing import StandardScaler
```

Packages for model evaluation and classification models

```
from sklearn.model_selection import train_test_split, KFold,\ 
cross_val_score, GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from sklearn.pipeline import Pipeline
from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier,
from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier
from sklearn.metrics import classification_report, confusion_matrix,\ 
accuracy_score
```

Packages for deep learning models

```
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
```

Packages for saving the model

```
from pickle import dump  
from pickle import load
```

3. Exploratory data analysis

The following sections walk through some high-level data inspection.

3.1. Descriptive statistics. The first thing we must do is gather a basic sense of our data. Remember, except for the transaction and amount, we do not know the names of other columns. The only thing we know is that the values of those columns have been scaled. Let's look at the shape and columns of the data:

```
# shape  
dataset.shape
```

Output

```
(284807, 31)
```

```
#peek at data  
set_option('display.width', 100)  
dataset.head(5)
```

Output

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26	V27	V28	Amount	Class
0	0.0	-1.360	-0.073	2.536	1.378	-0.338	0.462	0.240	0.099	0.364	...	-0.018	0.278	-0.110	0.067	0.129	-0.189	0.134	-0.021	149.62	0
1	0.0	1.192	0.266	0.166	0.448	0.060	-0.082	-0.079	0.085	-0.255	...	-0.226	-0.639	0.101	-0.340	0.167	0.126	-0.009	0.015	2.69	0
2	1.0	-1.358	-1.340	1.773	0.380	-0.503	1.800	0.791	0.248	-1.515	...	0.248	0.772	0.909	-0.689	-0.328	-0.139	-0.055	-0.060	378.66	0
3	1.0	-0.966	-0.185	1.793	-0.863	-0.010	1.247	0.238	0.377	-1.387	...	-0.108	0.005	-0.190	-1.176	0.647	-0.222	0.063	0.061	123.50	0
4	2.0	-1.158	0.878	1.549	0.403	-0.407	0.096	0.593	-0.271	0.818	...	-0.009	0.798	-0.137	0.141	-0.206	0.502	0.219	0.215	69.99	0

5 rows × 31 columns

As shown, the variable names are nondescript (*V1*, *V2*, etc.). Also, the data type for the entire dataset is `float`, except `Class`, which is of type `integer`.

How many are fraud and how many are not fraud? Let us check:

```
class_names = {0:'Not Fraud', 1:'Fraud'}  
print(dataset.Class.value_counts().rename(index = class_names))
```

Output

```
Not Fraud    284315  
Fraud        492  
Name: Class, dtype: int64
```

Notice the stark imbalance of the data labels. Most of the transactions are nonfraud. If we use this dataset as the base for our modeling, most models will not place enough emphasis on the fraud signals; the nonfraud data points will drown out any weight

the fraud signals provide. As is, we may encounter difficulties modeling the prediction of fraud, with this imbalance leading the models to simply assume *all* transactions are nonfraud. This would be an unacceptable result. We will explore some ways of dealing with this issue in the subsequent sections.

3.2. Data visualization. Since the feature descriptions are not provided, visualizing the data will not lead to much insight. This step will be skipped in this case study.

4. Data preparation

This data is from Kaggle and is already in a cleaned format without any empty rows or columns. Data cleaning or categorization is unnecessary.

5. Evaluate models

Now we are ready to split the data and evaluate the models.

5.1. Train-test split and evaluation metrics. As described in [Chapter 2](#), it is a good idea to partition the original dataset into training and test sets. The test set is a sample of the data that we hold back from our analysis and modeling. We use it at the end of our project to confirm the accuracy of our final model. It is the final test that gives us confidence in our estimates of accuracy on unseen data. We will use 80% of the dataset for model training and 20% for testing:

```
Y= dataset["Class"]
X = dataset.loc[:, dataset.columns != 'Class']
validation_size = 0.2
seed = 7
X_train, X_validation, Y_train, Y_validation =\
train_test_split(X, Y, test_size=validation_size, random_state=seed)
```

5.2. Checking models. In this step, we will evaluate different machine learning models. To optimize the various hyperparameters of the models, we use ten-fold cross validation and recalculate the results ten times to account for the inherent randomness in some of the models and the CV process. All of these models, including cross validation, are described in [Chapter 4](#).

Let us design our test harness. We will evaluate algorithms using the *accuracy metric*. This is a gross metric that will give us a quick idea of how correct a given model is. It is useful on binary classification problems.

```
# test options for classification
num_folds = 10
scoring = 'accuracy'
```

Let's create a baseline of performance for this problem and spot-check a number of different algorithms. The selected algorithms include:

Linear algorithms

Logistic regression (LR) and linear discriminant analysis (LDA).

Nonlinear algorithms

Classification and regression trees (CART) and K -nearest neighbors (KNN).

There are good reasons for selecting these models. These models are simpler and faster models with good interpretation for problems with large datasets. CART and KNN will be able to discern any nonlinear relationships between the variables. The key problem here is using an unbalanced sample. Unless we resolve that, more complex models, such as ensemble and ANNs, will have poor prediction. We will focus on addressing this later in the case study and then will evaluate the performance of these types of models.

```
# spot-check basic Classification algorithms
models = []
models.append(('LR', LogisticRegression()))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
```

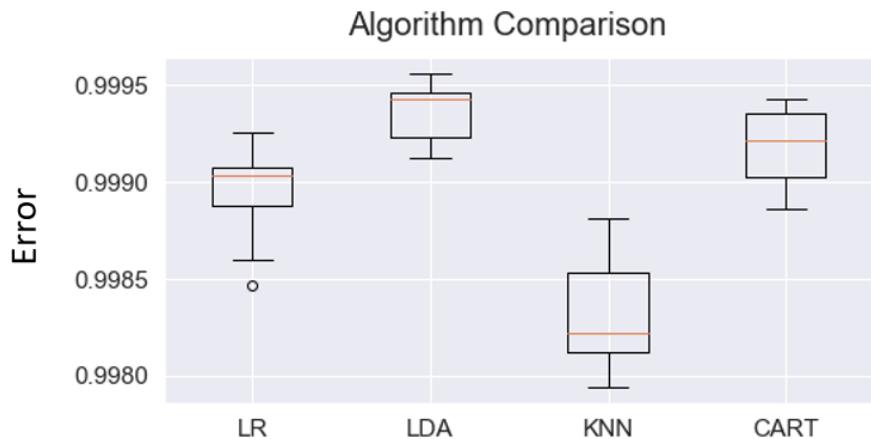
All the algorithms use default tuning parameters. We will display the mean and standard deviation of accuracy for each algorithm as we calculate and collect the results for use later.

```
results = []
names = []
for name, model in models:
    kfold = KFold(n_splits=num_folds, random_state=seed)
    cv_results = cross_val_score(model, X_train, Y_train, cv=kfold, \
        scoring=scoring)
    results.append(cv_results)
    names.append(name)
    msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
    print(msg)
```

Output

```
LR: 0.998942 (0.000229)
LDA: 0.999364 (0.000136)
KNN: 0.998310 (0.000290)
CART: 0.999175 (0.000193)
```

```
# compare algorithms
fig = pyplot.figure()
fig.suptitle('Algorithm Comparison')
ax = fig.add_subplot(111)
pyplot.boxplot(results)
ax.set_xticklabels(names)
fig.set_size_inches(8,4)
pyplot.show()
```



The accuracy of the overall result is quite high. But let us check how well it predicts the fraud cases. Choosing one of the model CART from the results above and looking at the result on the test set:

```
# prepare model
model = DecisionTreeClassifier()
model.fit(X_train, Y_train)

# estimate accuracy on validation set
predictions = model.predict(X_validation)
print(accuracy_score(Y_validation, predictions))
print(classification_report(Y_validation, predictions))
```

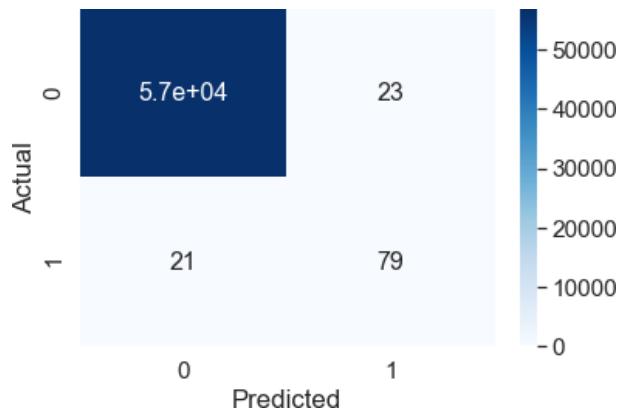
Output

```
0.9992275552122467
          precision    recall  f1-score   support
          0       1.00     1.00     1.00     56862
          1       0.77     0.79     0.78      100

    accuracy                           1.00     56962
   macro avg       0.89     0.89     0.89     56962
weighted avg       1.00     1.00     1.00     56962
```

And producing the confusion matrix yields:

```
df_cm = pd.DataFrame(confusion_matrix(Y_validation, predictions), \
columns=np.unique(Y_validation), index = np.unique(Y_validation))
df_cm.index.name = 'Actual'
df_cm.columns.name = 'Predicted'
sns.heatmap(df_cm, cmap="Blues", annot=True, annot_kws={"size": 16})
```



Overall accuracy is strong, but the confusion metrics tell a different story. Despite the high accuracy level, 21 out of 100 instances of fraud are missed and incorrectly predicted as nonfraud. The *false negative* rate is substantial.

The intention of a fraud detection model is to minimize these false negatives. To do so, the first step would be to choose the right evaluation metric.

In [Chapter 4](#), we covered the evaluation metrics, such as accuracy, precision, and recall, for a classification-based problem. Accuracy is the number of correct predictions made as a ratio of all predictions made. Precision is the number of items correctly identified as positive out of total items identified as positive by the model. Recall is the total number of items correctly identified as positive out of total true positives.

For this type of problem, we should focus on recall, the ratio of true positives to the sum of true positives and false negatives. So if false negatives are high, then the value of recall will be low.

In the next step, we perform model tuning, select the model using the recall metric, and perform under-sampling.

6. Model tuning

The purpose of the model tuning step is to perform the grid search on the model selected in the previous step. However, since we encountered poor model performance in the previous section due to the unbalanced dataset, we will focus our attention on that. We will analyze the impact of choosing the correct evaluation metric and see the impact of using an adjusted, balanced sample.

6.1. Model tuning by choosing the correct evaluation metric. As mentioned in the preceding step, if false negatives are high, then the value of recall will be low. Models are ranked according to this metric:

```
scoring = 'recall'
```

Let us spot-check some basic classification algorithms for recall:

```
models = []
models.append(('LR', LogisticRegression()))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
```

Running cross validation:

```
results = []
names = []
for name, model in models:
    kfold = KFold(n_splits=num_folds, random_state=seed)
    cv_results = cross_val_score(model, X_train, Y_train, cv=kfold, \
        scoring=scoring)
    results.append(cv_results)
    names.append(name)
    msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
    print(msg)
```

Output

```
LR: 0.595470 (0.089743)
LDA: 0.758283 (0.045450)
KNN: 0.023882 (0.019671)
CART: 0.735192 (0.073650)
```

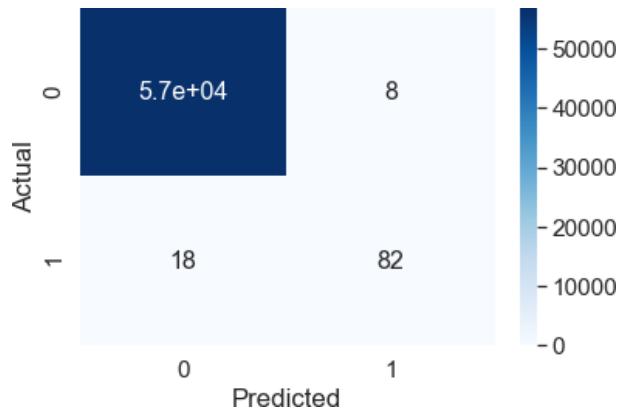
We see that the LDA model has the best recall of the four models. We continue by evaluating the test set using the trained LDA:

```
# prepare model
model = LinearDiscriminantAnalysis()
model.fit(X_train, Y_train)
# estimate accuracy on validation set

predictions = model.predict(X_validation)
print(accuracy_score(Y_validation, predictions))
```

Output

```
0.9995435553526912
```



LDA performs better, missing only 18 out of 100 cases of fraud. Additionally, we find fewer false positives as well. However, there is still improvement to be made.

6.2. Model tuning—balancing the sample by random under-sampling. The current data exhibits a significant class imbalance, where there are very few data points labeled “fraud.” The issue of such class imbalance can result in a serious bias toward the majority class, reducing the classification performance and increasing the number of false negatives.

One of the remedies to handle such situations is to *under-sample* the data. A simple technique is to under-sample the majority class randomly and uniformly. This might lead to a loss of information, but it may yield strong results by modeling the minority class well.

Next, we will implement random under-sampling, which consists of removing data to have a more balanced dataset. This will help ensure that our models avoid overfitting.

The steps to implement random under-sampling are:

1. First, we determine the severity of the class imbalance by using `value_counts()` on the class column. We determine how many instances are considered fraud transactions (`fraud = 1`).
2. We bring the nonfraud transaction observation count to the same amount as fraud transactions. Assuming we want a 50/50 ratio, this will be equivalent to 492 cases of fraud and 492 cases of nonfraud transactions.
3. We now have a subsample of our dataframe with a 50/50 ratio with regards to our classes. We train the models on this subsample. Then we perform this iteration again to shuffle the nonfraud observations in the training sample. We keep

track of the model performance to see whether our models can maintain a certain accuracy every time we repeat this process:

```
df = pd.concat([X_train, Y_train], axis=1)
# amount of fraud classes 492 rows.
fraud_df = df.loc[df['Class'] == 1]
non_fraud_df = df.loc[df['Class'] == 0][:492]

normal_distributed_df = pd.concat([fraud_df, non_fraud_df])

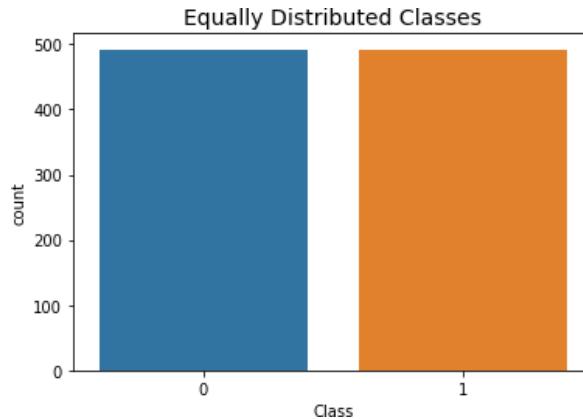
# Shuffle dataframe rows
df_new = normal_distributed_df.sample(frac=1, random_state=42)
# split out validation dataset for the end
Y_train_new= df_new["Class"]
X_train_new = df_new.loc[:, dataset.columns != 'Class']
```

Let us look at the distribution of the classes in the dataset:

```
print('Distribution of the Classes in the subsample dataset')
print(df_new['Class'].value_counts()/len(df_new))
sns.countplot('Class', data=df_new)
pyplot.title('Equally Distributed Classes', fontsize=14)
pyplot.show()
```

Output

```
Distribution of the Classes in the subsample dataset
1    0.5
0    0.5
Name: Class, dtype: float64
```



The data is now balanced, with close to 1,000 observations. We will train all the models again, including an ANN. Now that the data is balanced, we will focus on accuracy as our main evaluation metric, since it considers both false positives and false negatives. Recall can always be produced if needed:

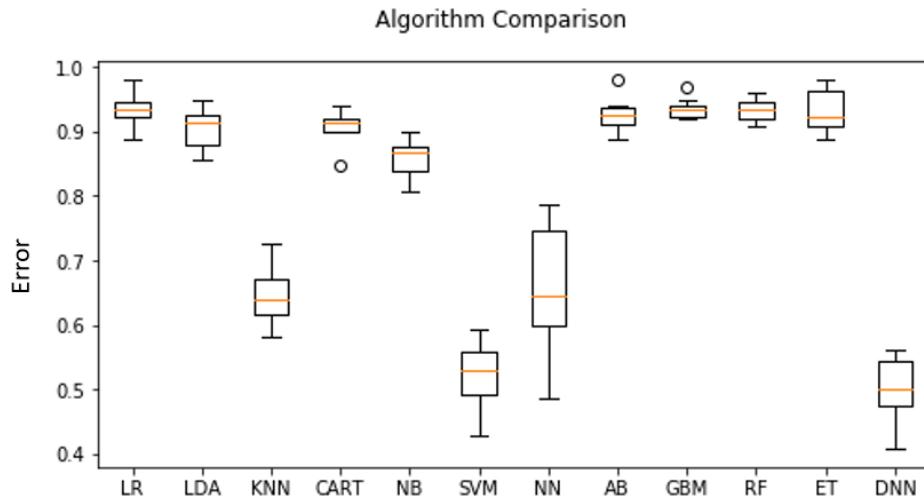
```
#setting the evaluation metric
scoring='accuracy'
# spot-check the algorithms
models = []
models.append(('LR', LogisticRegression()))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
models.append(('SVM', SVC()))
#Neural Network
models.append(('NN', MLPClassifier()))
# Ensemble Models
# Boosting methods
models.append(('AB', AdaBoostClassifier()))
models.append(('GBM', GradientBoostingClassifier()))
# Bagging methods
models.append(('RF', RandomForestClassifier()))
models.append(('ET', ExtraTreesClassifier()))
```

The steps to define and compile an ANN-based deep learning model in Keras, along with all the terms (neurons, activation, momentum, etc.) mentioned in the following code, have been described in [Chapter 3](#). This code can be leveraged for any deep learning-based classification model.

Keras-based deep learning model:

```
# Function to create model, required for KerasClassifier
def create_model(neurons=12, activation='relu', learn_rate = 0.01, momentum=0):
    # create model
    model = Sequential()
    model.add(Dense(X_train.shape[1], input_dim=X_train.shape[1], \
                    activation=activation))
    model.add(Dense(32,activation=activation))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    optimizer = SGD(lr=learn_rate, momentum=momentum)
    model.compile(loss='binary_crossentropy', optimizer='adam', \
                  metrics=['accuracy'])
    return model
models.append(('DNN', KerasClassifier(build_fn=create_model,\n                                     epochs=50, batch_size=10, verbose=0)))
```

Running the cross validation on the new set of models results in the following:



Although a couple of models, including random forest (RF) and logistic regression (LR), perform well, GBM slightly edges out the other models. We select this for further analysis. Note that the result of the deep learning model using Keras (i.e., “DNN”) is poor.

A grid search is performed for the GBM model by varying the number of estimators and maximum depth. The details of the GBM model and the parameters to tune for this model are described in [Chapter 4](#).

```
# Grid Search: GradientBoosting Tuning
n_estimators = [20, 180, 1000]
max_depth= [2, 3, 5]
param_grid = dict(n_estimators=n_estimators, max_depth=max_depth)
model = GradientBoostingClassifier()
kfold = KFold(n_splits=num_folds, random_state=seed)
grid = GridSearchCV(estimator=model, param_grid=param_grid, scoring=scoring, \
cv=kfold)
grid_result = grid.fit(X_train_new, Y_train_new)
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
```

Output

```
Best: 0.936992 using {'max_depth': 5, 'n_estimators': 1000}
```

In the next step, the final model is prepared, and the result on the test set is checked:

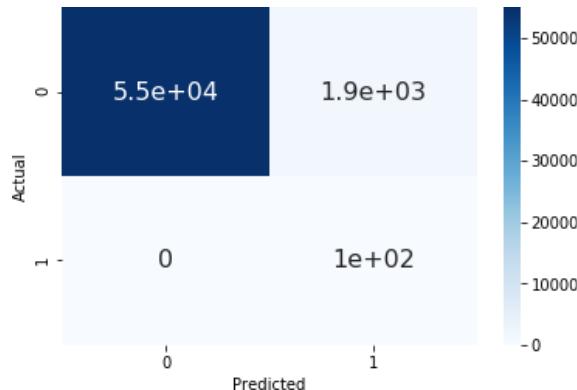
```
# prepare model
model = GradientBoostingClassifier(max_depth= 5, n_estimators = 1000)
model.fit(X_train_new, Y_train_new)
# estimate accuracy on Original validation set
predictions = model.predict(X_validation)
print(accuracy_score(Y_validation, predictions))
```

Output

```
0.9668199852533268
```

The accuracy of the model is high. Let's look at the confusion matrix:

Output



The results on the test set are impressive, with a high accuracy and, importantly, no false negatives. However, we see that an outcome of using our under-sampled data is a propensity for false positives—cases in which nonfraud transactions are misclassified as fraudulent. This is a trade-off the financial institution would have to consider. There is an inherent cost balance between the operational overhead, and possible customer experience impact, from processing false positives and the financial loss resulting from missing fraud cases through false negatives.

Conclusion

In this case study, we performed fraud detection on credit card transactions. We illustrated how different classification machine learning models stack up against each other and demonstrated that choosing the right metric can make an important difference in model evaluation. Under-sampling was shown to lead to a significant improvement, as all fraud cases in the test set were correctly identified after applying under-sampling. This came with a trade-off, though. The reduction in false negatives came with an increase in false positives.

Overall, by using different machine learning models, choosing the right evaluation metrics, and handling unbalanced data, we demonstrated how the implementation of a simple classification-based model can produce robust results for fraud detection.

Case Study 2: Loan Default Probability

Lending is one of the most important activities of the finance industry. Lenders provide loans to borrowers in exchange for the promise of repayment with interest. That means the lender makes a profit only if the borrower pays off the loan. Hence, the two most critical questions in the lending industry are:

1. How risky is the borrower?
2. Given the borrower's risk, should we lend to them?

Default prediction could be described as a perfect job for machine learning, as the algorithms can be trained on millions of examples of consumer data. Algorithms can perform automated tasks such as matching data records, identifying exceptions, and calculating whether an applicant qualifies for a loan. The underlying trends can be assessed with algorithms and continuously analyzed to detect trends that might influence lending and underwriting risk in the future.

The goal of this case study is to build a machine learning model to predict the probability that a loan will default.

In most real-life cases, including loan default modeling, we are unable to work with clean, complete data. Some of the potential problems we are bound to encounter are missing values, incomplete categorical data, and irrelevant features. Although data cleaning may not be mentioned often, it is critical for the success of machine learning applications. The algorithms that we use can be powerful, but without the relevant or appropriate data, the system may fail to yield ideal results. So one of the focus areas of this case study will be data preparation and cleaning. Various techniques and concepts of data processing, feature selection, and exploratory analysis are used for data cleaning and organizing the feature space.

In this case study, we will focus on:

- Data preparation, data cleaning, and handling a large number of features.
- Data discretization and handling categorical data.
- Feature selection and data transformation.



Blueprint for Creating a Machine Learning Model for Predicting Loan Default Probability

1. Problem definition

In the classification framework for this case study, the predicted variable is *charge-off*, a debt that a creditor has given up trying to collect on after a borrower has missed payments for several months. The predicted variable takes a value of 1 in case of charge-off and a value of 0 otherwise.

We will analyze data for loans from 2007 to 2017Q3 from Lending Club, [available on Kaggle](#). Lending Club is a US peer-to-peer lending company. It operates an online lending platform that enables borrowers to obtain a loan and investors to purchase notes backed by payments made on these loans. The dataset contains more than 887,000 observations with 150 variables containing complete loan data for all loans issued over the specified time period. The features include income, age, credit scores, home ownership, borrower location, collections, and many others. We will investigate these 150 predictor variables for feature selection.

By the end of this case study, readers will be familiar with a general approach to loan default modeling, from gathering and cleaning data to building and tuning a classifier.

2. Getting started—loading the data and Python packages

2.1. Loading the Python packages. The standard Python packages are loaded in this step. The details have been presented in the previous case studies. Please refer to the Jupyter notebook for this case study for more details.

2.2. Loading the data. The loan data for the time period from 2007 to 2017Q3 is loaded:

```
# load dataset
dataset = pd.read_csv('LoansData.csv.gz', compression='gzip', \
low_memory=True)
```

3. Data preparation and feature selection

In the first step, let us look at the size of the data:

```
dataset.shape
```

Output

```
(1646801, 150)
```

Given that there are 150 features for each loan, we will first focus on limiting the feature space, followed by the exploratory analysis.

3.1. Preparing the predicted variable. Here, we look at the details of the predicted variable and prepare it. The predicted variable will be derived from the `loan_status` column. Let's check the value distributions:²

```
dataset['loan_status'].value_counts(dropna=False)
```

Output

Current	788950
Fully Paid	646902
Charged Off	168084
Late (31-120 days)	23763
In Grace Period	10474
Late (16-30 days)	5786
Does not meet the credit policy. Status:Fully Paid	1988
Does not meet the credit policy. Status:Charged Off	761
Default	70
NaN	23

Name: loan_status, dtype: int64

From the data definition documentation:

Fully Paid

Loans that have been fully repaid.

Default

Loans that have not been current for 121 days or more.

Charged Off

Loans for which there is no longer a reasonable expectation of further payments.

A large proportion of observations show a status of `Current`, and we do not know whether those will be `Charged Off`, `Fully Paid`, or `Default` in the future. The observations for `Default` are tiny in number compared to `Fully Paid` or `Charged Off` and are not considered. The remaining categories of `loan_status` are not of prime importance for this analysis. So, in order to convert this to a binary classification problem and to analyze in detail the effect of important variables on the loan status, we will consider only two major categories—`Charged Off` and `Fully Paid`:

² The predicted variable is further used for correlation-based feature reduction.

```
dataset = dataset.loc[dataset['loan_status'].isin(['Fully Paid', 'Charged Off'])]
dataset['loan_status'].value_counts(normalize=True, dropna=False)
```

Output

```
Fully Paid      0.793758
Charged Off    0.206242
Name: loan_status, dtype: float64
```

About 79% of the remaining loans have been fully paid and 21% have been charged off, so we have a somewhat unbalanced classification problem, but it is not as unbalanced as the dataset of fraud detection we saw in the previous case study.

In the next step, we create a new binary column in the dataset, where we categorize Fully Paid as 0 and Charged Off as 1. This column represents the predicted variable for this classification problem. A value of 1 in this column indicates the borrower has defaulted:

```
dataset['charged_off'] = (dataset['loan_status'] == 'Charged Off').apply(np.uint8)
dataset.drop('loan_status', axis=1, inplace=True)
```

3.2. Feature selection—limit the feature space. The full dataset has 150 features for each loan, but not all features contribute to the prediction variable. Removing features of low importance can improve accuracy and reduce both model complexity and overfitting. Training time can also be reduced for very large datasets. We'll eliminate features in the following steps using three different approaches:

- Eliminating features that have more than 30% missing values.
- Eliminating features that are unintuitive based on subjective judgment.
- Eliminating features with low correlation with the predicted variable.

3.2.1. Feature elimination based on significant missing values. First, we calculate the percentage of missing data for each feature:

```
missing_fractions = dataset.isnull().mean().sort_values(ascending=False)

#Drop the missing fraction
drop_list = sorted(list(missing_fractions[missing_fractions > 0.3].index))
dataset.drop(labels=drop_list, axis=1, inplace=True)
dataset.shape
```

Output

```
(814986, 92)
```

This dataset has 92 columns remaining once some of the columns with a significant number of missing values are dropped.

3.2.2. Feature elimination based on intuitiveness. To filter the features further we check the description in the data dictionary and keep the features that intuitively contribute

to the prediction of default. We keep features that contain the relevant credit detail of the borrower, including annual income, FICO score, and debt-to-income ratio. We also keep those features that are available to investors when considering an investment in the loan. These include features in the loan application and any features added by Lending Club when the loan listing was accepted, such as loan grade and interest rate.

The list of the features retained are shown in the following code snippet:

```
keep_list = ['charged_off', 'funded_amnt', 'addr_state', 'annual_inc', \
'application_type', 'dti', 'earliest_cr_line', 'emp_length', \
'emp_title', 'fico_range_high', \
'fico_range_low', 'grade', 'home_ownership', 'id', 'initial_list_status', \
'installment', 'int_rate', 'loan_amnt', 'loan_status', \
'mort_acc', 'open_acc', 'pub_rec', 'pub_rec_bankruptcies', \
'purpose', 'revol_bal', 'revol_util', \
'sub_grade', 'term', 'title', 'total_acc', \
'verification_status', 'zip_code', 'last_pymnt_amnt', \
'num_actv_rev_tl', 'mo_sin_rcnt_rev_tl_op', \
'mo_sin_old_rev_tl_op', "bc_util", "bc_open_to_buy", \
"avg_cur_bal", "acc_open_past_24mths" ]\n\ndrop_list = [col for col in dataset.columns if col not in keep_list]\ndataset.drop(labels=drop_list, axis=1, inplace=True)\ndataset.shape
```

Output

```
(814986, 39)
```

After removing the features in this step, 39 columns remain.

3.2.3. Feature elimination based on the correlation. The next step is to check the correlation with the predicted variable. Correlation gives us the interdependence between the predicted variable and the feature. We select features with a moderate-to-strong relationship with the target variable and drop those that have a correlation of less than 3% with the predicted variable:

```
correlation = dataset.corr()\ncorrelation_chargeOff = abs(correlation['charged_off'])\ndrop_list_corr = sorted(list(correlation_chargeOff \
[correlation_chargeOff < 0.03].index))\nprint(drop_list_corr)
```

Output

```
['pub_rec', 'pub_rec_bankruptcies', 'revol_bal', 'total_acc']
```

The columns with low correlation are dropped from the dataset, and we are left with only 35 columns:

```
dataset.drop(labels=drop_list_corr, axis=1, inplace=True)
```

4. Feature selection and exploratory analysis

In this step, we perform the exploratory data analysis of the feature selection. Given that many features had to be eliminated, it is preferable that we perform the exploratory data analysis after feature selection to better visualize the relevant features. We will also continue the feature elimination by visually screening and dropping those features deemed irrelevant.

4.1. Feature analysis and exploration. In the following sections, we take a deeper dive into the dataset features.

4.1.1. Analyzing the categorical features. Let us look at some of the categorical features in the dataset.

First, let's look at the `id`, `emp_title`, `title`, and `zip_code` features:

```
dataset[['id', 'emp_title', 'title', 'zip_code']].describe()
```

Output

	<code>id</code>	<code>emp_title</code>	<code>title</code>	<code>zip_code</code>
count	814986	766415	807068	814986
unique	814986	280473	60298	925
top	14680062	Teacher	Debt consolidation	945xx
freq	1	11351	371874	9517

IDs are all unique and irrelevant for modeling. There are too many unique values for employment titles and titles. Occupation and job title may provide some information for default modeling; however, we assume much of this information is embedded in the verified income of the customer. Moreover, additional cleaning steps on these features, such as standardizing or grouping the titles, would be necessary to extract any marginal information. This work is outside the scope of this case study but could be explored in subsequent iterations of the model.

Geography could play a role in credit determination, and zip codes provide a granular view of this dimension. Again, additional work would be necessary to prepare this feature for modeling and was deemed outside the scope of this case study.

```
dataset.drop(['id', 'emp_title', 'title', 'zip_code'], axis=1, inplace=True)
```

Let's look at the `term` feature.

Term refers to the number of payments on the loan. Values are in months and can be either 36 or 60. The 60-month loans are more likely to charge off.

Let's convert term to integers and group by the term for further analysis:

```
dataset['term'] = dataset['term'].apply(lambda s: np.int8(s.split()[0]))
dataset.groupby('term')[['charged_off']].value_counts(normalize=True).loc[:,1]
```

Output

```
term
36    0.165710
60    0.333793
Name: charged_off, dtype: float64
```

Loans with five-year periods are more than twice as likely to charge-off as loans with three-year periods. This feature seems to be important for the prediction.

Let's look at the emp_length feature:

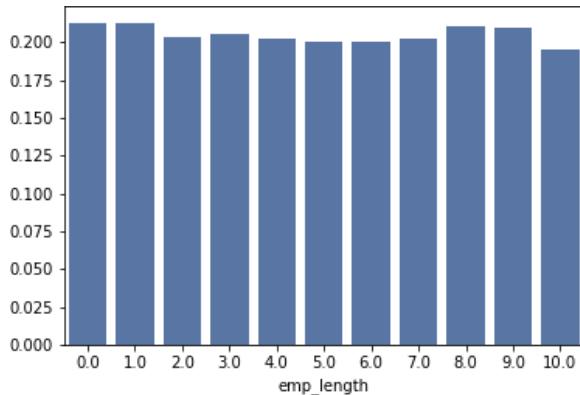
```
dataset['emp_length'].replace(to_replace='10+ years', value='10 years', \
                             inplace=True)

dataset['emp_length'].replace('< 1 year', '0 years', inplace=True)

def emp_length_to_int(s):
    if pd.isnull(s):
        return s
    else:
        return np.int8(s.split()[0])

dataset['emp_length'] = dataset['emp_length'].apply(emp_length_to_int)
charge_off_rates = dataset.groupby('emp_length')[['charged_off']].value_counts\
    (normalize=True).loc[:,1]
sns.barplot(x=charge_off_rates.index, y=charge_off_rates.values)
```

Output



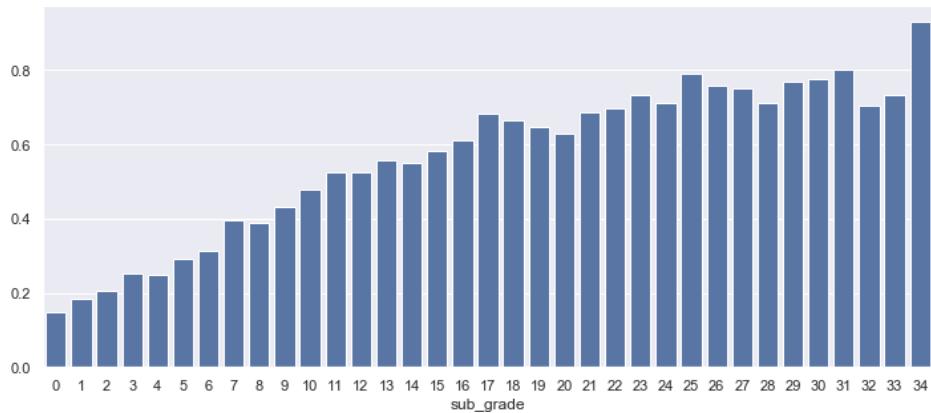
Loan status does not appear to vary much with employment length (on average); hence this feature is dropped:

```
dataset.drop(['emp_length'], axis=1, inplace=True)
```

Let's look at the `sub_grade` feature:

```
charge_off_rates = dataset.groupby('sub_grade')['charged_off'].value_counts\
(normalize=True).loc[:,1]
sns.barplot(x=charge_off_rates.index, y=charge_off_rates.values)
```

Output



As shown in the chart, there's a clear trend of higher probability of charge-off as the sub-grade worsens, and so it is considered to be a key feature.

4.1.2. Analyzing the continuous features.

Let's look at the `annual_inc` feature:

```
dataset[['annual_inc']].describe()
```

Output

annual_inc
count 8.149860e+05
mean 7.523039e+04
std 6.524373e+04
min 0.000000e+00
25% 4.500000e+04
50% 6.500000e+04
75% 9.000000e+04
max 9.550000e+06

Annual income ranges from \$0 to \$9,550,000, with a median of \$65,000. Because of the large range of incomes, we use a log transform of the annual income variable:

```
dataset['log_annual_inc'] = dataset['annual_inc'].apply(lambda x: np.log10(x+1))
dataset.drop('annual_inc', axis=1, inplace=True)
```

Let's look at the FICO score (fico_range_low, fico_range_high) feature:

```
dataset[['fico_range_low', 'fico_range_high']].corr()
```

Output

	fico_range_low	fico_range_high
fico_range_low	1.0	
fico_range_high	1.0	

Given that the correlation between FICO low and high is 1, it is preferred that we keep only one feature, which we take as the average of FICO scores:

```
dataset['fico_score'] = 0.5*dataset['fico_range_low'] +\
0.5*dataset['fico_range_high']

dataset.drop(['fico_range_high', 'fico_range_low'], axis=1, inplace=True)
```

4.2. Encoding categorical data. In order to use a feature in the classification models, we need to convert the categorical data (i.e., text features) to its numeric representation. This process is called encoding. There can be different ways of encoding. However, for this case study we will use a *label encoder*, which encodes labels with a value between 0 and n , where n is the number of distinct labels. The `LabelEncoder` function from `sklearn` is used in the following step, and all the categorical columns are encoded at once:

```
from sklearn.preprocessing import LabelEncoder
# Categorical boolean mask
categorical_feature_mask = dataset.dtypes==object
# filter categorical columns using mask and turn it into a list
categorical_cols = dataset.columns[categorical_feature_mask].tolist()
```

Let us look at the categorical columns:

```
categorical_cols
```

Output

```
['grade',
 'sub_grade',
 'home_ownership',
 'verification_status',
 'purpose',
 'addr_state',
 'initial_list_status',
 'application_type']
```

4.3. Sampling data. Given that the loan data is skewed, it is sampled to have an equal number of charge-off and no charge-off observations. Sampling leads to a more balanced dataset and avoids overfitting:³

```
loanstatus_0 = dataset[dataset["charged_off"]==0]
loanstatus_1 = dataset[dataset["charged_off"]==1]
subset_of_loanstatus_0 = loanstatus_0.sample(n=5500)
subset_of_loanstatus_1 = loanstatus_1.sample(n=5500)
dataset = pd.concat([subset_of_loanstatus_1, subset_of_loanstatus_0])
dataset = dataset.sample(frac=1).reset_index(drop=True)
print("Current shape of dataset :",dataset.shape)
```

Although sampling may have its advantages, there might be some disadvantages as well. Sampling may exclude some data that might not be homogeneous to the data that is taken. This affects the level of accuracy in the results. Also, selection of the proper size of samples is a difficult job. Hence, sampling should be performed with caution and should generally be avoided in the case of a relatively balanced dataset.

5. Evaluate algorithms and models

5.1. Train-test split. Splitting out the validation dataset for the model evaluation is the next step:

```
Y= dataset["charged_off"]
X = dataset.loc[:, dataset.columns != 'charged_off']
validation_size = 0.2
seed = 7
X_train, X_validation, Y_train, Y_validation = \
train_test_split(X, Y, test_size=validation_size, random_state=seed)
```

5.2. Test options and evaluation metrics. In this step, the test options and evaluation metrics are selected. The `roc_auc` evaluation metric is selected for this classification. The details of this metric were provided in [Chapter 4](#). This metric represents a model's ability to discriminate between positive and negative classes. An `roc_auc` of 1.0 represents a model that made all predictions perfectly, and a value of 0.5 represents a model that is as good as random.

```
num_folds = 10
scoring = 'roc_auc'
```

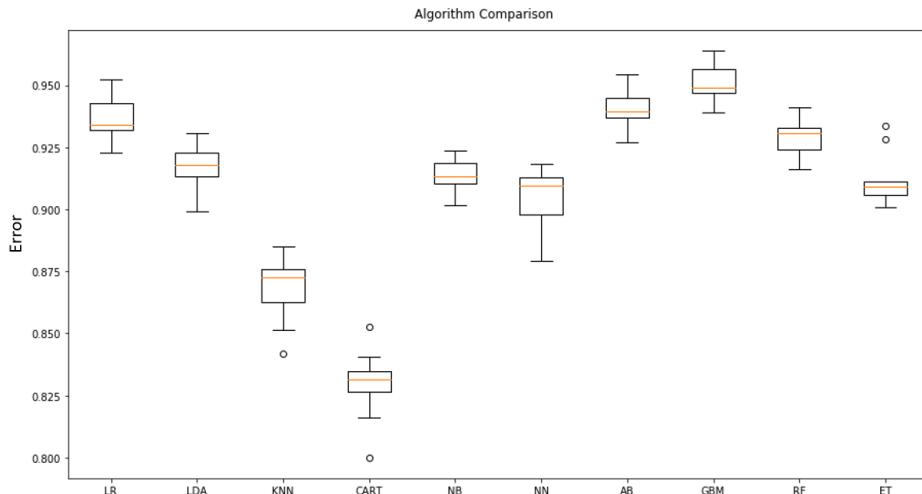
The model cannot afford to have a high amount of false negatives as that leads to a negative impact on the investors and the credibility of the company. So we can use recall as we did in the fraud detection use case.

³ Sampling is covered in detail in “[Case Study 1: Fraud Detection](#)” on page 153.

5.3. Compare models and algorithms. Let us spot-check the classification algorithms. We include ANN and ensemble models in the list of models to be checked:

```
models = []
models.append(('LR', LogisticRegression()))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
# Neural Network
models.append(('NN', MLPClassifier()))
# Ensemble Models
# Boosting methods
models.append(('AB', AdaBoostClassifier()))
models.append(('GBM', GradientBoostingClassifier()))
# Bagging methods
models.append(('RF', RandomForestClassifier()))
models.append(('ET', ExtraTreesClassifier()))
```

After performing the k -fold cross validation on the models shown above, the overall performance is as follows:



The gradient boosting method (GBM) model performs best, and we select it for grid search in the next step. The details of GBM along with the model parameters are described in [Chapter 4](#).

6. Model tuning and grid search

We tune the number of estimator and maximum depth hyperparameters, which were discussed in Chapter 4:

```
# Grid Search: GradientBoosting Tuning
n_estimators = [20,180]
max_depth= [3,5]
param_grid = dict(n_estimators=n_estimators, max_depth=max_depth)
model = GradientBoostingClassifier()
kfold = KFold(n_splits=num_folds, random_state=seed)
grid = GridSearchCV(estimator=model, param_grid=param_grid, scoring=scoring, \
cv=kfold)
grid_result = grid.fit(X_train, Y_train)
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
```

Output

```
Best: 0.952950 using {'max_depth': 5, 'n_estimators': 180}
```

A GBM model with `max_depth` of 5 and number of estimators of 150 results in the best model.

7. Finalize the model

Now, we perform the final steps for selecting a model.

7.1. Results on the test dataset. Let us prepare the GBM model with the parameters found during the grid search step and check the results on the test dataset:

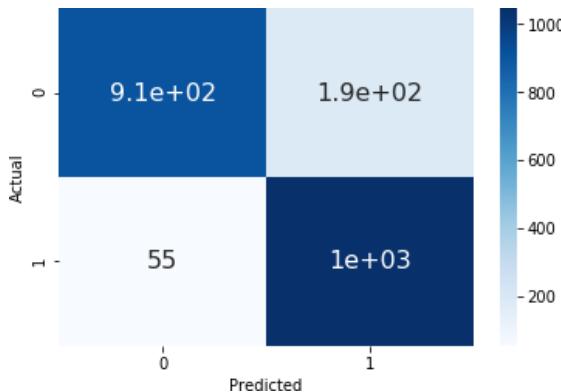
```
model = GradientBoostingClassifier(max_depth= 5, n_estimators= 180)
model.fit(X_train, Y_train)

# estimate accuracy on validation set
predictions = model.predict(X_validation)
print(accuracy_score(Y_validation, predictions))
```

Output

```
0.8890909090909090
```

The accuracy of the model is a reasonable 89% on the test set. Let us examine the confusion matrix:

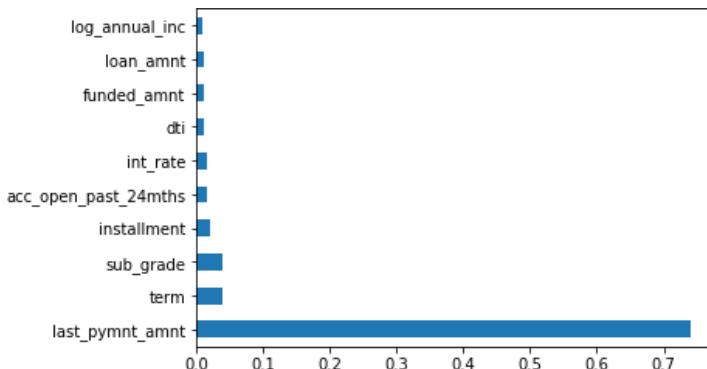


Looking at the confusion matrix and the overall result of the test set, both the rate of false positives and the rate of false negatives are lower; the overall model performance looks good and is in line with the training set results.

7.2. Variable intuition/feature importance. In this step, we compute and display the variable importance of our trained model:

```
print(model.feature_importances_) #use inbuilt class feature_importances_
feat_importances = pd.Series(model.feature_importances_, index=X.columns)
#plot graph of feature importances for better visualization
feat_importances.nlargest(10).plot(kind='barh')
pyplot.show()
```

Output



The results of the model importance are intuitive. The last payment amount seems to be the most important feature, followed by loan term and sub-grade.

Conclusion

In this case study, we introduced the classification-based tree algorithm applied to loan default prediction. We showed that data preparation is one of the most important steps. We addressed this by performing feature elimination using different techniques, such as feature intuition, correlation analysis, visualization, and data quality checks of the features. We illustrated that there can be different ways of handling and analyzing the categorical data and converting categorical data into a usable format for the models.

We emphasized that performing data processing and establishing an understanding of variable importance is key in the model development process. A focus on these steps led to the implementation of a simple classification-based model that produced robust results for default prediction.

Case Study 3: Bitcoin Trading Strategy

First released as open source in 2009 by the pseudonymous Satoshi Nakamoto, bitcoin is the longest-running and most well-known cryptocurrency.

A major drawback of cryptocurrency trading is the volatility of the market. Since cryptocurrency markets trade 24/7, tracking cryptocurrency positions against quickly changing market dynamics can rapidly become an impossible task to manage. This is where automated trading algorithms and trading bots can assist.

Various machine learning algorithms can be used to generate trading signals in an attempt to predict the market's movement. One could use machine learning algorithms to classify the next day's movement into three categories: market will rise (take a long position), market will fall (take a short position), or market will move sideways (take no position). Since we know the market direction, we can decide the optimum entry and exit points.

Machine learning has one key aspect called *feature engineering*. It means that we can create new, intuitive features and feed them to a machine learning algorithm in order to achieve better results. We can introduce different technical indicators as features to help predict future prices of an asset. These technical indicators are derived from market variables such as price or volume and have additional information or signals embedded in them. There are many different categories of technical indicators, including trend, volume, volatility, and momentum indicators.

In this case study, we will use various classification-based models to predict whether the current position signal is buy or sell. We will create additional trend and momentum indicators from market prices to leverage as additional features in the prediction.

In this case study, we will focus on:

- Building a trading strategy using classification (classification of long/short signals).
- Feature engineering and constructing technical indicators of trend, momentum, and mean reversion.
- Building a framework for backtesting results of a trading strategy.
- Choosing the right evaluation metric to assess a trading strategy.



Blueprint for Using Classification-Based Models to Predict Whether to Buy or Sell in the Bitcoin Market

1. Problem definition

The problem of predicting a buy or sell signal for a trading strategy is defined in the classification framework, where the predicted variable has a value of 1 for buy and 0 for sell. This signal is decided through the comparison of the short-term and long-term price trends.

The data used is from one of the largest bitcoin exchanges in terms of average daily volume, [Bitstamp](#). The data covers prices from January 2012 to May 2017. Different trend and momentum indicators are created from the data and are added as features to enhance the performance of the prediction model.

By the end of this case study, readers will be familiar with a general approach to building a trading strategy, from cleaning data and feature engineering to model tuning and developing a backtesting framework.

2. Getting started—loading the data and Python packages

Let's load the packages and the data.

2.1. Loading the Python packages. The standard Python packages are loaded in this step. The details have been presented in the previous case studies. Refer to the Jupyter notebook for this case study for more details.

2.2. Loading the data. The bitcoin data fetched from the Bitstamp website is loaded in this step:

```
# load dataset
dataset = pd.read_csv('BitstampData.csv')
```

3. Exploratory data analysis

In this step, we will take a detailed look at this data.

3.1. Descriptive statistics.

First, let us look at the shape of the data:

```
dataset.shape
```

Output

```
(2841377, 8)
```

```
# peek at data
set_option('display.width', 100)
dataset.tail(2)
```

Output

	Timestamp	Open	High	Low	Close	Volume_(BTC)	Volume_(Currency)	Weighted_Price
2841372	1496188560	2190.49	2190.49	2181.37	2181.37	1.700166	3723.784755	2190.247337
2841373	1496188620	2190.50	2197.52	2186.17	2195.63	6.561029	14402.811961	2195.206304

The dataset has minute-by-minute data of OHLC (Open, High, Low, Close) and traded volume of bitcoin. The dataset is large, with approximately 2.8 million total observations.

4. Data preparation

In this part, we will clean the data to prepare for modeling.

4.1. Data cleaning.

We clean the data by filling the NaNs with the last available values:

```
dataset[dataset.columns.values] = dataset[dataset.columns.values].ffill()
```

The `Timestamp` column is not useful for modeling and is dropped from the dataset:

```
dataset=dataset.drop(columns=['Timestamp'])
```

4.2. Preparing the data for classification.

As a first step, we will create the target variable for our model. This is the column that will indicate whether the trading signal is buy or sell. We define the short-term price as the 10-day rolling average and the long-term price as the 60-day rolling average. We attach a label of 1 (0) if the short-term price is higher (lower) than the long-term price:

```
# Create short simple moving average over the short window
dataset['short_mavg'] = dataset['Close'].rolling(window=10, min_periods=1,\ncenter=False).mean()
```

```

# Create long simple moving average over the long window
dataset['long_mavg'] = dataset['Close'].rolling(window=60, min_periods=1,\\
center=False).mean()

# Create signals
dataset['signal'] = np.where(dataset['short_mavg'] >
dataset['long_mavg'], 1.0, 0.0)

```

4.3. Feature engineering. We begin feature engineering by analyzing the features we expect may influence the performance of the prediction model. Based on a conceptual understanding of key factors that drive investment strategies, the task at hand is to identify and construct new features that may capture the risks or characteristics embodied by these return drivers. For this case study, we will explore the efficacy of specific momentum technical indicators.

The current data of the bitcoin consists of date, open, high, low, close, and volume. Using this data, we calculate the following momentum indicators:

Moving average

A moving average provides an indication of a price trend by cutting down the amount of noise in the series.

Stochastic oscillator %K

A stochastic oscillator is a momentum indicator that compares the closing price of a security to a range of its previous prices over a certain period of time. $\%K$ and $\%D$ are slow and fast indicators. The fast indicator is more sensitive than the slow indicator to changes in the price of the underlying security and will likely result in many transaction signals.

Relative strength index (RSI)

This is a momentum indicator that measures the magnitude of recent price changes to evaluate overbought or oversold conditions in the price of a stock or other asset. The RSI ranges from 0 to 100. An asset is deemed to be overbought once the RSI approaches 70, meaning that the asset may be getting overvalued and is a good candidate for a pullback. Likewise, if the RSI approaches 30, it is an indication that the asset may be getting oversold and is therefore likely to become undervalued.

Rate of change (ROC)

This is a momentum oscillator that measures the percentage change between the current price and the n period past prices. Assets with higher ROC values are considered more likely to be overbought; those with lower ROC, more likely to be oversold.

Momentum (MOM)

This is the rate of acceleration of a security's price or volume—that is, the speed at which the price is changing.

The following steps show how to generate some useful features for prediction. The features for trend and momentum can be leveraged for other trading strategy models:

```
#calculation of exponential moving average
def EMA(df, n):
    EMA = pd.Series(df['Close'].ewm(span=n, min_periods=n).mean(), name='EMA_'\ 
        + str(n))
    return EMA
dataset['EMA10'] = EMA(dataset, 10)
dataset['EMA30'] = EMA(dataset, 30)
dataset['EMA200'] = EMA(dataset, 200)
dataset.head()

#calculation of rate of change
def ROC(df, n):
    M = df.diff(n - 1)
    N = df.shift(n - 1)
    ROC = pd.Series(((M / N) * 100), name = 'ROC_' + str(n))
    return ROC
dataset['ROC10'] = ROC(dataset['Close'], 10)
dataset['ROC30'] = ROC(dataset['Close'], 30)

#calculation of price momentum
def MOM(df, n):
    MOM = pd.Series(df.diff(n), name='Momentum_' + str(n))
    return MOM
dataset['MOM10'] = MOM(dataset['Close'], 10)
dataset['MOM30'] = MOM(dataset['Close'], 30)

#calculation of relative strength index
def RSI(series, period):
    delta = series.diff().dropna()
    u = delta * 0
    d = u.copy()
    u[delta > 0] = delta[delta > 0]
    d[delta < 0] = -delta[delta < 0]
    u[u.index[period-1]] = np.mean( u[:period] ) #first value is sum of avg gains
    u = u.drop(u.index[:-(period-1)])
    d[d.index[period-1]] = np.mean( d[:period] ) #first value is sum of avg losses
    d = d.drop(d.index[:-(period-1)])
    rs = u.ewm(com=period-1, adjust=False).mean() / \
        d.ewm(com=period-1, adjust=False).mean()
    return 100 - 100 / (1 + rs)
dataset['RSI10'] = RSI(dataset['Close'], 10)
dataset['RSI30'] = RSI(dataset['Close'], 30)
dataset['RSI200'] = RSI(dataset['Close'], 200)

#calculation of stochastic osillator.
```

```

def STOK(close, low, high, n):
    STOK = ((close - low.rolling(n).min()) / (high.rolling(n).max() - \
    low.rolling(n).min())) * 100
    return STOK

def STOD(close, low, high, n):
    STOK = ((close - low.rolling(n).min()) / (high.rolling(n).max() - \
    low.rolling(n).min())) * 100
    STOD = STOK.rolling(3).mean()
    return STOD

dataset['%K10'] = STOK(dataset['Close'], dataset['Low'], dataset['High'], 10)
dataset['%D10'] = STOD(dataset['Close'], dataset['Low'], dataset['High'], 10)
dataset['%K30'] = STOK(dataset['Close'], dataset['Low'], dataset['High'], 30)
dataset['%D30'] = STOD(dataset['Close'], dataset['Low'], dataset['High'], 30)
dataset['%K200'] = STOK(dataset['Close'], dataset['Low'], dataset['High'], 200)
dataset['%D200'] = STOD(dataset['Close'], dataset['Low'], dataset['High'], 200)

#calculation of moving average
def MA(df, n):
    MA = pd.Series(df['Close'].rolling(n, min_periods=n).mean(), name='MA_'\ 
    + str(n))
    return MA
dataset['MA21'] = MA(dataset, 10)
dataset['MA63'] = MA(dataset, 30)
dataset['MA252'] = MA(dataset, 200)

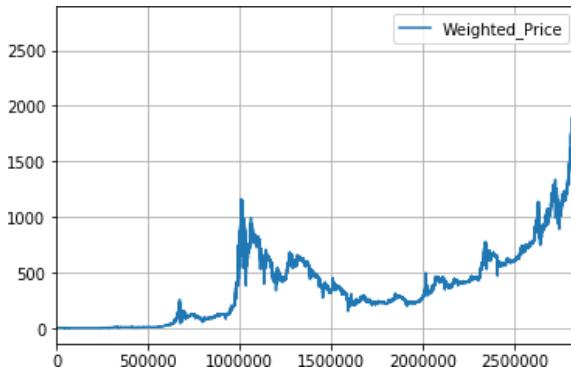
```

With our features completed, we'll prepare them for use.

4.4. Data visualization. In this step, we visualize different properties of the features and the predicted variable:

```
dataset[['Weighted_Price']].plot(grid=True)
plt.show()
```

Output

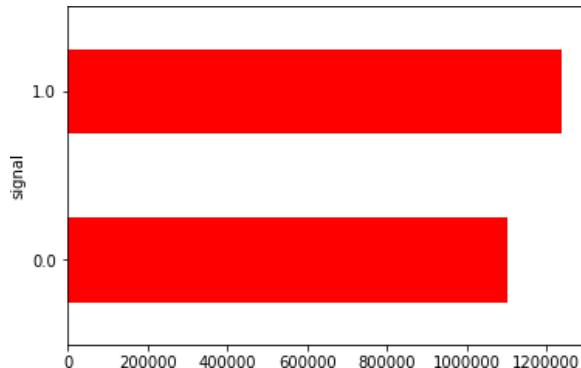


The chart illustrates a sharp rise in the price of bitcoin, increasing from close to \$0 to around \$2,500 in 2017. Also, high price volatility is readily visible.

Let us look at the distribution of the predicted variable:

```
fig = plt.figure()
plot = dataset.groupby(['signal']).size().plot(kind='barh', color='red')
plt.show()
```

Output



The predicted variable is 1 more than 52% of the time, meaning there are more buy signals than sell signals. The predicted variable is relatively balanced, especially as compared to the fraud dataset we saw in the first case study.

5. Evaluate algorithms and models

In this step, we will evaluate different algorithms.

5.1. Train-test split. We first split the dataset into training (80%) and test (20%) sets. For this case study we use 100,000 observations for a faster calculation. The next steps would be same in the event we want to use the entire dataset:

```
# split out validation dataset for the end
subset_dataset= dataset.iloc[-100000:]
Y= subset_dataset["signal"]
X = subset_dataset.loc[:, dataset.columns != 'signal']
validation_size = 0.2
seed = 1
X_train, X_validation, Y_train, Y_validation =\
train_test_split(X, Y, test_size=validation_size, random_state=1)
```

5.2. Test options and evaluation metrics. Accuracy can be used as the evaluation metric since there is not a significant class imbalance in the data:

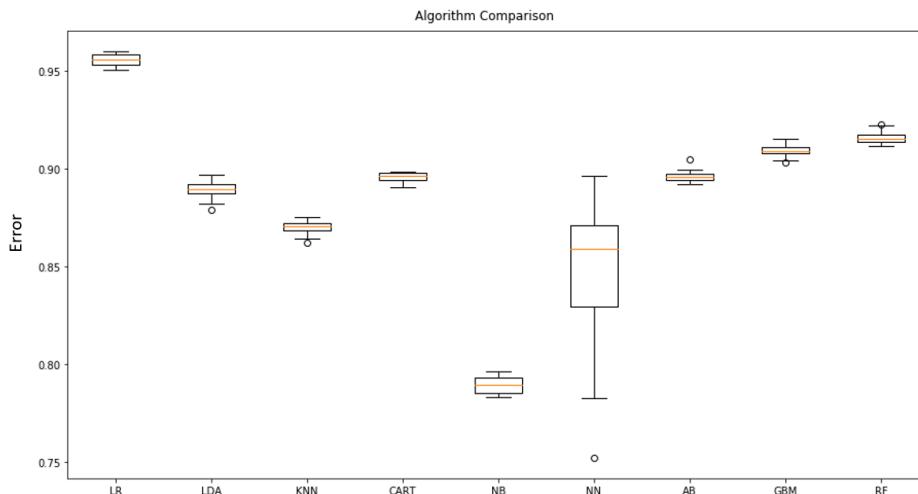
```
# test options for classification
num_folds = 10
scoring = 'accuracy'
```

5.3. Compare models and algorithms. In order to know which algorithm is best for our strategy, we evaluate the linear, nonlinear, and ensemble models.

5.3.1. Models. Checking the classification algorithms:

```
models = []
models.append(('LR', LogisticRegression(n_jobs=-1)))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
#Neural Network
models.append(('NN', MLPClassifier()))
# Ensemble Models
# Boosting methods
models.append(('AB', AdaBoostClassifier()))
models.append(('GBM', GradientBoostingClassifier()))
# Bagging methods
models.append(('RF', RandomForestClassifier(n_jobs=-1)))
```

After performing the k -fold cross validation, the comparison of the models is as follows:



Although some of the models show promising results, we prefer an ensemble model given the huge size of the dataset, the large number of features, and an expected non-linear relationship between the predicted variable and the features. Random forest has the best performance among the ensemble models.

6. Model tuning and grid search

A grid search is performed for the random forest model by varying the number of estimators and maximum depth. The details of the random forest model and the parameters to tune are discussed in [Chapter 4](#):

```
n_estimators = [20,80]
max_depth= [5,10]
criterion = ["gini","entropy"]
param_grid = dict(n_estimators=n_estimators, max_depth=max_depth, \
    criterion = criterion )
model = RandomForestClassifier(n_jobs=-1)
kfold = KFold(n_splits=num_folds, random_state=seed)
grid = GridSearchCV(estimator=model, param_grid=param_grid, \
    scoring=scoring, cv=kfold)
grid_result = grid.fit(X_train, Y_train)
print("Best: %f using %s" % (grid_result.best_score_,\
    grid_result.best_params_))
```

Output

```
Best: 0.903438 using {'criterion': 'gini', 'max_depth': 10, 'n_estimators': 80}
```

7. Finalize the model

Let us finalize the model with the best parameters found during the tuning step and perform the variable intuition.

7.1. Results on the test dataset. In this step, we evaluate the selected model on the test set:

```
# prepare model
model = RandomForestClassifier(criterion='gini', n_estimators=80,max_depth=10)

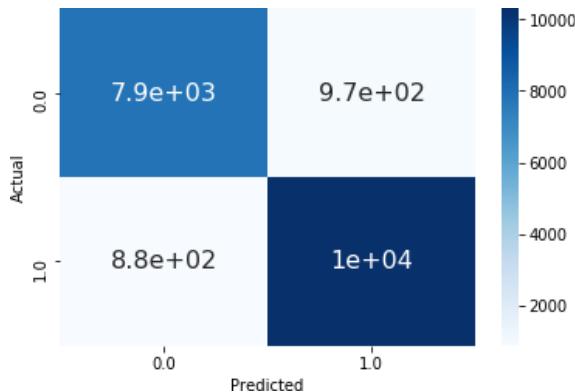
#model = LogisticRegression()
model.fit(X_train, Y_train)

# estimate accuracy on validation set
predictions = model.predict(X_validation)
print(accuracy_score(Y_validation, predictions))
```

Output

```
0.9075
```

The selected model performs quite well, with an accuracy of 90.75%. Let us look at the confusion matrix:

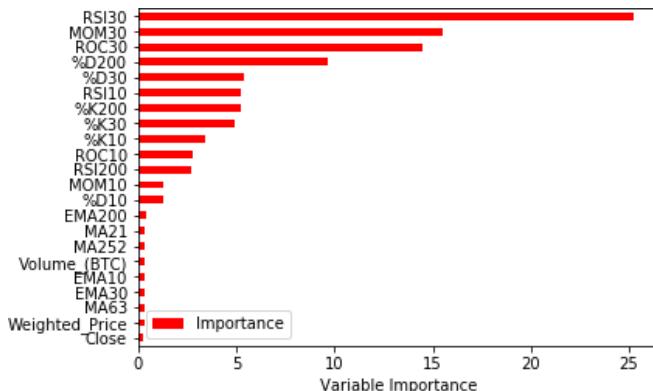


The overall model performance is reasonable and is in line with the training set results.

7.2. Variable intuition/feature importance. Let us look into the feature importance of the model:

```
Importance = pd.DataFrame({'Importance':model.feature_importances_*100},\n                           index=X.columns)\nImportance.sort_values('Importance', axis=0, ascending=True).plot(kind='barh', \n                      color='r')\nplt.xlabel('Variable Importance')
```

Output



The result of the variable importance looks intuitive, and the momentum indicators of RSI and MOM over the last 30 days seem to be the two most important features.

The feature importance chart corroborates the fact that introducing new features leads to an improvement in the model performance.

7.3. Backtesting results. In this additional step, we perform a backtest on the model we've developed. We create a column for strategy returns by multiplying the daily returns by the position that was held at the close of business the previous day and compare it against the actual returns.



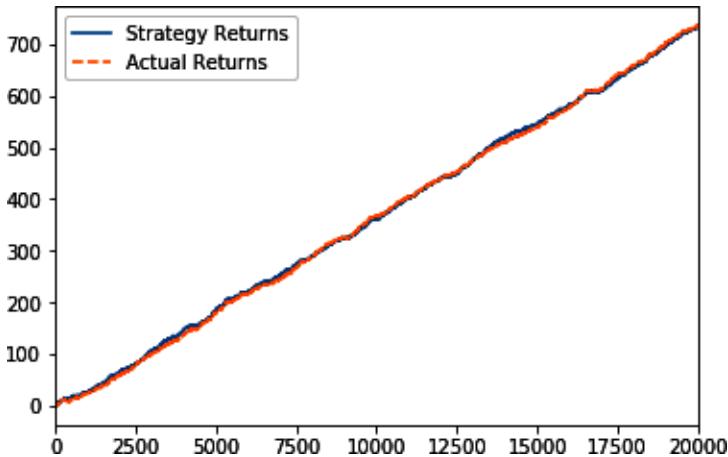
Backtesting Trading Strategies

A backtesting approach similar to the one presented in this case study can be used to quickly backtest any trading strategy.

```
backtestdata = pd.DataFrame(index=X_validation.index)
backtestdata['signal_pred'] = predictions
backtestdata['signal_actual'] = Y_validation
backtestdata['Market Returns'] = X_validation['Close'].pct_change()
backtestdata['Actual Returns'] = backtestdata['Market Returns'] * \
backtestdata['signal_actual'].shift(1)
backtestdata['Strategy Returns'] = backtestdata['Market Returns'] * \
backtestdata['signal_pred'].shift(1)
backtestdata=backtestdata.reset_index()
backtestdata.head()
backtestdata[['Strategy Returns','Actual Returns']].cumsum().hist()
backtestdata[['Strategy Returns','Actual Returns']].cumsum().plot()
```

Output





Looking at the backtesting results, we do not deviate significantly from the actual market return. Indeed, the achieved momentum trading strategy made us better at predicting the price direction to buy or sell in order to make profits. However, as our accuracy is not 100% (but more than 96%), we made relatively few losses compared to the actual returns.

Conclusion

This case study demonstrated that framing the problem is a key step when tackling a finance problem with machine learning. In doing so, it was determined that transforming the labels according to an investment objective and performing feature engineering were required for this trading strategy. We demonstrated the efficiency of using intuitive features related to the trend and momentum of the price movement. This helped increase the predictive power of the model. Finally, we introduced a backtesting framework, which allowed us to simulate a trading strategy using historical data. This enabled us to generate results and analyze risk and profitability before risking any actual capital.

Chapter Summary

In “Case Study 1: Fraud Detection” on page 153, we explored the issue of an unbalanced dataset and the importance of having the right evaluation metric. In “Case Study 2: Loan Default Probability” on page 166, various techniques and concepts of data processing, feature selection, and exploratory analysis were covered. In “Case Study 3: Bitcoin Trading Strategy” on page 179, we looked at ways to create technical

indicators as features in order to use them for model enhancement. We also prepared a backtesting framework for a trading strategy.

Overall, the concepts in Python, machine learning, and finance presented in this chapter can be used as a blueprint for any other classification-based problem in finance.

Exercises

- Predict whether a stock price will go up or down using the features related to the stock or macroeconomic variables (use the ideas from the bitcoin-based case study presented in this chapter).
- Create a model to detect money laundering using the features of a transaction. A sample dataset for this exercise can be obtained from [Kaggle](#).
- Perform a credit rating analysis of corporations using the features related to creditworthiness.

PART III

Unsupervised Learning

Unsupervised Learning: Dimensionality Reduction

In previous chapters, we used supervised learning techniques to build machine learning models using data where the answer was already known (i.e., the class labels were available in our input data). Now we will explore *unsupervised learning*, where we draw inferences from datasets consisting of input data when the answer is unknown. Unsupervised learning algorithms attempt to infer patterns from the data without any knowledge of the output the data is meant to yield. Without requiring labeled data, which can be time-consuming and impractical to create or acquire, this family of models allows for easy use of larger datasets for analysis and model development.

Dimensionality reduction is a key technique within unsupervised learning. It compresses the data by finding a smaller, different set of variables that capture what matters most in the original features, while minimizing the loss of information. Dimensionality reduction helps mitigate problems associated with high dimensionality and permits the visualization of salient aspects of higher-dimensional data that is otherwise difficult to explore.

In the context of finance, where datasets are often large and contain many dimensions, dimensionality reduction techniques prove to be quite practical and useful. Dimensionality reduction enables us to reduce noise and redundancy in the dataset and find an approximate version of the dataset using fewer features. With fewer variables to consider, exploration and visualization of a dataset becomes more straightforward. Dimensionality reduction techniques also enhance supervised learning-based models by reducing the number of features or by finding new ones. Practitioners use these dimensionality reduction techniques to allocate funds across asset classes and individual investments, identify trading strategies and signals, implement portfolio hedging and risk management, and develop instrument pricing models.

In this chapter, we will discuss fundamental dimensionality reduction techniques and walk through three case studies in the areas of portfolio management, interest rate modeling, and trading strategy development. The case studies are designed to not only cover diverse topics from a finance standpoint but also highlight multiple machine learning and data science concepts. The standardized template containing the detailed implementation of modeling steps in Python and machine learning and finance concepts can be used as a blueprint for any other dimensionality reduction-based problem in finance.

In “[Case Study 1: Portfolio Management: Finding an Eigen Portfolio](#)” on page 202, we use a dimensionality reduction algorithm to allocate capital into different asset classes to maximize risk-adjusted returns. We also introduce a backtesting framework to assess the performance of the portfolio we constructed.

In “[Case Study 2: Yield Curve Construction and Interest Rate Modeling](#)” on page 217, we use dimensionality reduction techniques to generate the typical movements of a yield curve. This will illustrate how dimensionality reduction techniques can be used for reducing the dimension of market variables across a number of asset classes to promote faster and effective portfolio management, trading, hedging, and risk management.

In “[Case Study 3: Bitcoin Trading: Enhancing Speed and Accuracy](#)” on page 227, we use dimensionality reduction techniques for algorithmic trading. This case study demonstrates data exploration in low dimension.

In addition to the points mentioned above, readers will understand the following points by the end of this chapter:

- Basic concepts of models and techniques used for dimensionality reduction and how to implement them in Python.
- Concepts of eigenvalues and eigenvectors of Principal Component Analysis (PCA), selecting the right number of principal components, and extracting the factor weights of principal components.
- Usage of dimensionality reduction techniques such as singular value decomposition (SVD) and t-SNE to summarize high-dimensional data for effective data exploration and visualization.
- How to reconstruct the original data using the reduced principal components.
- How to enhance the speed and accuracy of supervised learning algorithms using dimensionality reduction.

- A backtesting framework for the portfolio performance computing and analyzing portfolio performance metrics such as the Sharpe ratio and the annualized return of the portfolio.



This Chapter's Code Repository

A Python-based master template for dimensionality reduction, along with the Jupyter notebook for all the case studies in this chapter, is included in the folder *Chapter 7 - Unsup. Learning - Dimensionality Reduction* in the code repository for this book. To work through any dimensionality reduction–modeling machine learning problems in Python involving the dimensionality reduction models (such as PCA, SVD, Kernel PCA, or t-SNE) presented in this chapter, readers need to modify the template slightly to align with their problem statement. All the case studies presented in this chapter use the standard Python master template with the standardized model development steps presented in [Chapter 3](#). For the dimensionality reduction case studies, steps 6 (i.e., model tuning) and 7 (i.e., finalizing the model) are relatively lighter compared to the supervised learning models, so these steps have been merged with step 5. For situations in which steps are irrelevant, they have been skipped or combined with others to make the flow of the case study more intuitive.

Dimensionality Reduction Techniques

Dimensionality reduction represents the information in a given dataset more efficiently by using fewer features. These techniques project data onto a lower dimensional space by either discarding variation in the data that is not informative or identifying a lower dimensional subspace on or near where the data resides.

There are many types of dimensionality reduction techniques. In this chapter, we will cover these most frequently used techniques for dimensionality reduction:

- Principal component analysis (PCA)
- Kernel principal component analysis (KPCA)
- t-distributed stochastic neighbor embedding (t-SNE)

After application of these dimensionality reduction techniques, the low-dimension feature subspace can be a linear or nonlinear function of the corresponding high-dimensional feature subspace. Hence, on a broad level these dimensionality reduction algorithms can be classified as linear and nonlinear. Linear algorithms, such as PCA, force the new variables to be linear combinations of the original features.

Nonlinear algorithms such KPCA and t-SNE can capture more complex structures in the data. However, given the infinite number of options, the algorithms still need to make assumptions to arrive at a solution.

Principal Component Analysis

The idea of principal component analysis (PCA) is to reduce the dimensionality of a dataset with a large number of variables, while retaining as much variance in the data as possible. PCA allows us to understand whether there is a different representation of the data that can explain a majority of the original data points.

PCA finds a set of new variables that, through a linear combination, yield the original variables. The new variables are called *principal components* (PCs). These principal components are orthogonal (or independent) and can represent the original data. The number of components is a hyperparameter of the PCA algorithm that sets the target dimensionality.

The PCA algorithm works by projecting the original data onto the principal component space. It then identifies a sequence of principal components, each of which aligns with the direction of maximum variance in the data (after accounting for variation captured by previously computed components). The sequential optimization also ensures that new components are not correlated with existing components. Thus the resulting set constitutes an orthogonal basis for a vector space.

The decline in the amount of variance of the original data explained by each principal component reflects the extent of correlation among the original features. The number of components that capture, for example, 95% of the original variation relative to the total number of features provides an insight into the linearly independent information of the original data. In order to understand how PCA works, let's consider the distribution of data shown in [Figure 7-1](#).

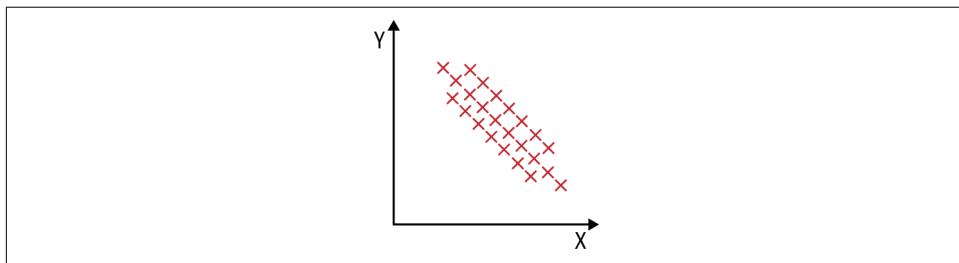


Figure 7-1. PCA-1

PCA finds a new quadrant system (y' and x' axes) that is obtained from the original through translation and rotation. It will move the center of the coordinate system from the original point $(0, 0)$ to the center of the distribution of data points. It will then move the x -axis into the principal axis of variation, which is the one with the

most variation relative to data points (i.e., the direction of maximum spread). Then it moves the other axis orthogonally to the principal one, into a less important direction of variation.

Figure 7-2 shows an example of PCA in which two dimensions explain nearly all the variance of the underlying data.

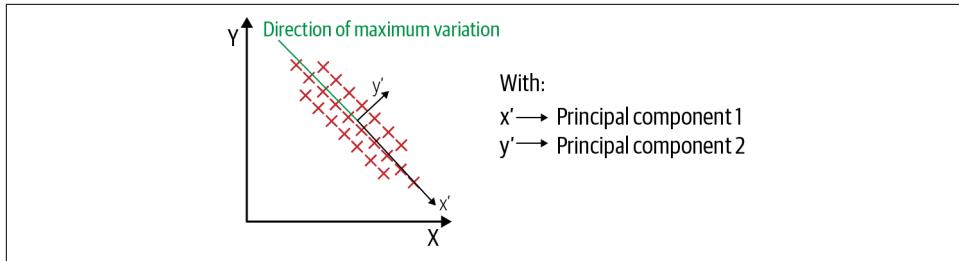


Figure 7-2. PCA-2

These new directions that contain the maximum variance are called principal components and are orthogonal to each other by design.

There are two approaches to finding the principal components: *Eigen decomposition* and *singular value decomposition* (SVD).

Eigen decomposition

The steps of Eigen decomposition are as follows:

1. First, a covariance matrix is created for the features.
2. Once the covariance matrix is computed, the *eigenvectors* of the covariance matrix are calculated.¹ These are the directions of maximum variance.
3. The *eigenvalues* are then created. They define the magnitude of the principal components.

So, for n dimensions, there will be an $n \times n$ variance-covariance matrix, and as a result, we will have an eigenvector of n values and n eigenvalues.

Python's sklearn library offers a powerful implementation of PCA. The `sklearn.decomposition.PCA` function computes the desired number of principal components and projects the data into the component space. The following code snippet illustrates how to create two principal components from a dataset.

¹ Eigenvectors and eigenvalues are concepts of linear algebra.

Implementation

```
# Import PCA Algorithm
from sklearn.decomposition import PCA
# Initialize the algorithm and set the number of PC's
pca = PCA(n_components=2)
# Fit the model to data
pca.fit(data)
# Get list of PC's
pca.components_
# Transform the model to data
pca.transform(data)
# Get the eigenvalues
pca.explained_variance_ratio_
```

There are additional items, such as *factor loading*, that can be obtained using the functions in the `sklearn` library. Their use will be demonstrated in the case studies.

Singular value decomposition

Singular value decomposition (SVD) is factorization of a matrix into three matrices and is applicable to a more general case of $m \times n$ rectangular matrices.

If A is an $m \times n$ matrix, then SVD can express the matrix as:

$$A = U\Sigma V^T$$

where A is an $m \times n$ matrix, U is an $(m \times m)$ orthogonal matrix, Σ is an $(m \times n)$ nonnegative rectangular diagonal matrix, and V is an $(n \times n)$ orthogonal matrix. SVD of a given matrix tells us exactly how we can decompose the matrix. Σ is a diagonal matrix with m diagonal values called *singular values*. Their magnitude indicates how significant they are to preserving the information of the original data. V contains the principal components as column vectors.

As shown above, both Eigen decomposition and SVD tell us that using PCA is effectively looking at the initial data from a different angle. Both will always give the same answer; however, SVD can be much more efficient than Eigen decomposition, as it is able to handle sparse matrices (those which contain very few nonzero elements). In addition, SVD yields better numerical stability, especially when some of the features are strongly correlated.

Truncated SVD is a variant of SVD that computes only the largest singular values, where the number of computes is a user-specified parameter. This method is different from regular SVD in that it produces a factorization where the number of columns is equal to the specified truncation. For example, given an $n \times n$ matrix, SVD will produce matrices with n columns, whereas truncated SVD will produce matrices with a specified number of columns that may be less than n .

Implementation

```
from sklearn.decomposition import TruncatedSVD  
svd = TruncatedSVD(ncomps=20).fit(X)
```

In terms of the weaknesses of the PCA technique, although it is very effective in reducing the number of dimensions, the resulting principal components may be less interpretable than the original features. Additionally, the results may be sensitive to the selected number of principal components. For example, too few principal components may miss some information compared to the original list of features. Also, PCA may not work well if the data is strongly nonlinear.

Kernel Principal Component Analysis

A main limitation of PCA is that it only applies linear transformations. Kernel principal component analysis (KPCA) extends PCA to handle nonlinearity. It first maps the original data to some nonlinear feature space (usually one of higher dimension). Then it applies PCA to extract the principal components in that space.

A simple example of when KPCA is applicable is shown in [Figure 7-3](#). Linear transformations are suitable for the blue and red data points on the left-hand plot. However, if all dots are arranged as per the graph on the right, the result is not linearly separable. We would then need to apply KPCA to separate the components.

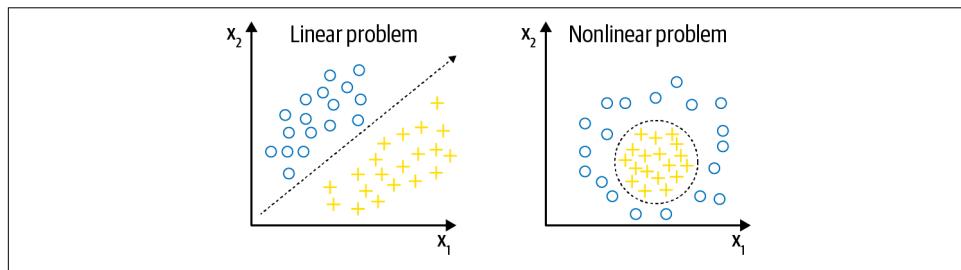


Figure 7-3. Kernel PCA

Implementation

```
from sklearn.decomposition import KernelPCA  
kpca = KernelPCA(n_components=4, kernel='rbf').fit_transform(X)
```

In the Python code, we specify `kernel='rbf'`, which is the **radial basis function kernel**. This is commonly used as a kernel in machine learning techniques, such as in SVMs (see [Chapter 4](#)).

Using KPCA, component separation becomes easier in a higher dimensional space, as mapping into a higher dimensional space often provides greater classification power.

t-distributed Stochastic Neighbor Embedding

t-distributed stochastic neighbor embedding (t-SNE) is a dimensionality reduction algorithm that reduces the dimensions by modeling the probability distribution of neighbors around each point. Here, the term *neighbors* refers to the set of points closest to a given point. The algorithm emphasizes keeping similar points together in low dimensions as opposed to maintaining the distance between points that are apart in high dimensions.

The algorithm starts by calculating the probability of similarity of data points in corresponding high and low dimensional space. The similarity of points is calculated as the conditional probability that a point *A* would choose point *B* as its neighbor if neighbors were picked in proportion to their probability density under a normal distribution centered at *A*. The algorithm then tries to minimize the difference between these conditional probabilities (or similarities) in the high and low dimensional spaces for a perfect representation of data points in the low dimensional space.

Implementation

```
from sklearn.manifold import TSNE  
X_tsne = TSNE().fit_transform(X)
```

An implementation of t-SNE is shown in the third case study presented in this chapter.

Case Study 1: Portfolio Management: Finding an Eigen Portfolio

A primary objective of portfolio management is to allocate capital into different asset classes to maximize risk-adjusted returns. Mean-variance portfolio optimization is the most commonly used technique for asset allocation. This method requires an estimated covariance matrix and expected returns of the assets considered. However, the erratic nature of financial returns leads to estimation errors in these inputs, especially when the sample size of returns is insufficient compared to the number of assets being allocated. These errors greatly jeopardize the optimality of the resulting portfolios, leading to poor and unstable outcomes.

Dimensionality reduction is a technique we can use to address this issue. Using PCA, we can take an $n \times n$ covariance matrix of our assets and create a set of n linearly uncorrelated principal portfolios (sometimes referred to in literature as an *eigen portfolio*) made up of our assets and their corresponding variances. The principal components of the covariance matrix capture most of the covariation among the assets and are mutually uncorrelated. Moreover, we can use standardized principal components as the portfolio weights, with the statistical guarantee that the returns from these principal portfolios are linearly uncorrelated.

By the end of this case study, readers will be familiar with a general approach to finding an eigen portfolio for asset allocation, from understanding concepts of PCA to backtesting different principal components.

This case study will focus on:

- Understanding eigenvalues and eigenvectors of PCA and deriving portfolio weights using the principal components.
- Developing a backtesting framework to evaluate portfolio performance.
- Understanding how to work through a dimensionality reduction modeling problem from end to end.



Blueprint for Using Dimensionality Reduction for Asset Allocation

1. Problem definition

Our goal in this case study is to maximize the risk-adjusted returns of an equity portfolio using PCA on a dataset of stock returns.

The dataset used for this case study is the Dow Jones Industrial Average (DJIA) index and its respective 30 stocks. The return data used will be from the year 2000 onwards and can be downloaded from Yahoo Finance.

We will also compare the performance of our hypothetical portfolios against a benchmark and backtest the model to evaluate the effectiveness of the approach.

2. Getting started—loading the data and Python packages

2.1. Loading the Python packages. The list of the libraries used for data loading, data analysis, data preparation, model evaluation, and model tuning are shown below. The details of most of these packages and functions can be found in Chapters 2 and 4.

Packages for dimensionality reduction

```
from sklearn.decomposition import PCA
from sklearn.decomposition import TruncatedSVD
from numpy.linalg import inv, eig, svd
from sklearn.manifold import TSNE
from sklearn.decomposition import KernelPCA
```

Packages for data processing and visualization

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from pandas import read_csv, set_option
from pandas.plotting import scatter_matrix
import seaborn as sns
from sklearn.preprocessing import StandardScaler
```

2.2. Loading the data. We import the dataframe containing the adjusted closing prices for all the companies in the DJIA index:

```
# load dataset
dataset = read_csv('Dow_adjcloses.csv', index_col=0)
```

3. Exploratory data analysis

Next, we inspect the dataset.

3.1. Descriptive statistics. Let's look at the shape of the data:

```
dataset.shape
```

Output

```
(4804, 30)
```

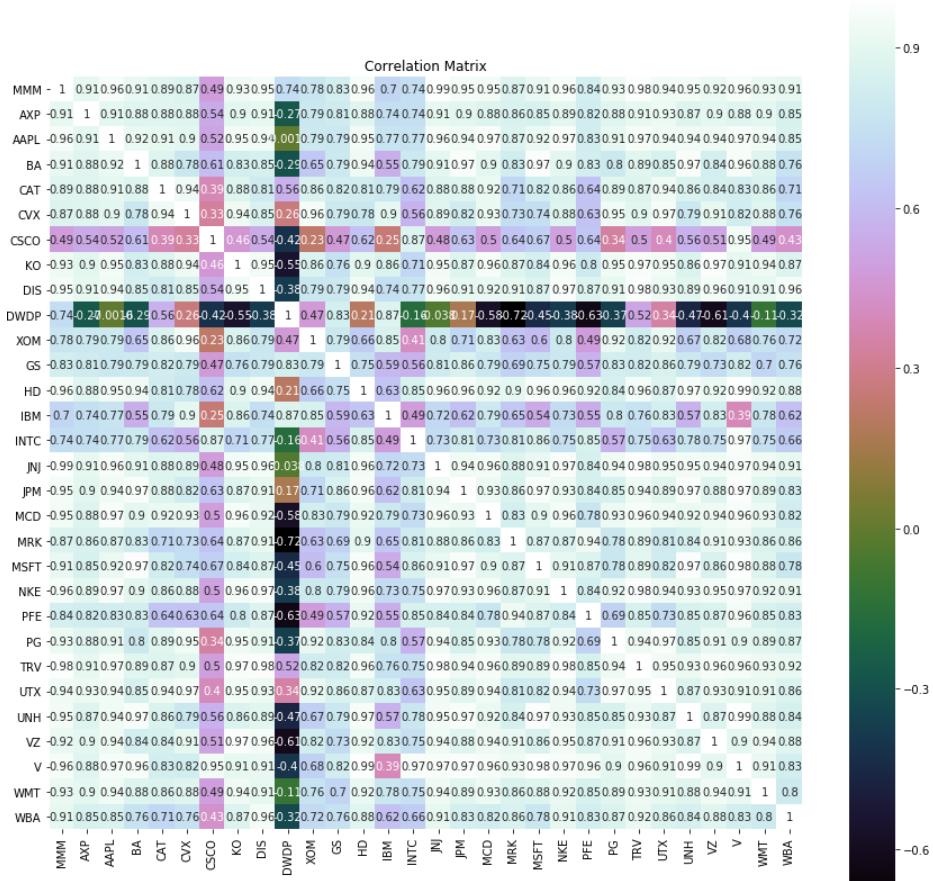
The data is comprised of 30 columns and 4,804 rows containing the daily closing prices of the 30 stocks in the index since 2000.

3.2. Data visualization. The first thing we must do is gather a basic sense of our data. Let us take a look at the return correlations:

```
correlation = dataset.corr()
plt.figure(figsize=(15, 15))
plt.title('Correlation Matrix')
sns.heatmap(correlation, vmax=1, square=True, annot=True, cmap='cubehelix')
```

There is a significant positive correlation between the daily returns. The plot (full-size version available on [GitHub](#)) also indicates that the information embedded in the data may be represented by fewer variables (i.e., something smaller than the 30 dimensions we have now). We will perform another detailed look at the data after implementing dimensionality reduction.

Output



4. Data preparation

We prepare the data for modeling in the following sections.

4.1. Data cleaning. First, we check for NAs in the rows and either drop them or fill them with the mean of the column:

```
#Checking for any null values and removing the null values'''  
print('Null Values =',dataset.isnull().values.any())
```

Output

Null Values = True

Some stocks were added to the index after our start date. To ensure proper analysis, we will drop those with more than 30% missing values. Two stocks fit this criteria—Dow Chemicals and Visa:

```
missing_fractions = dataset.isnull().mean().sort_values(ascending=False)
missing_fractions.head(10)
drop_list = sorted(list(missing_fractions[missing_fractions > 0.3].index))
dataset.drop(labels=drop_list, axis=1, inplace=True)
dataset.shape
```

Output

```
(4804, 28)
```

We end up with return data for 28 companies and an additional one for the DJIA index. Now we fill the NAs with the mean of the columns:

```
# Fill the missing values with the last value available in the dataset.
dataset=dataset.fillna(method='ffill')
```

4.2. Data transformation. In addition to handling the missing values, we also want to standardize the dataset features onto a unit scale (mean = 0 and variance = 1). All the variables should be on the same scale before applying PCA; otherwise, a feature with large values will dominate the result. We use `StandardScaler` in `sklearn` to standardize the dataset, as shown below:

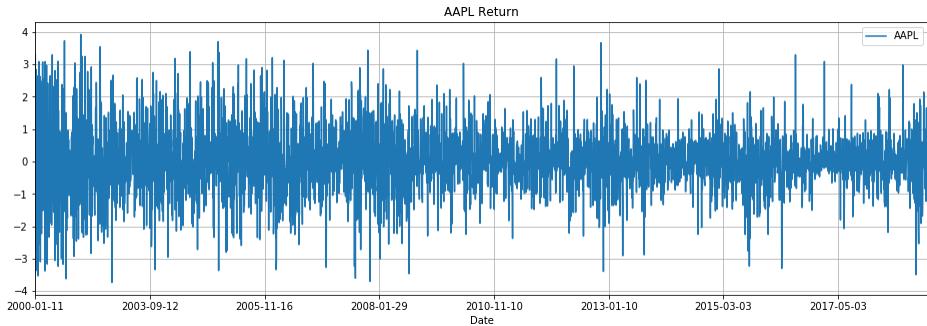
```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler().fit(datareturns)
rescaledDataset = pd.DataFrame(scaler.fit_transform(datareturns),columns =
    datareturns.columns, index = datareturns.index)
# summarize transformed data
datareturns.dropna(how='any', inplace=True)
rescaledDataset.dropna(how='any', inplace=True)
```

Overall, cleaning and standardizing the data is important in order to create a meaningful and reliable dataset to be used in dimensionality reduction without error.

Let us look at the returns of one of the stocks from the cleaned and standardized dataset:

```
# Visualizing Log Returns for the DJIA
plt.figure(figsize=(16, 5))
plt.title("AAPL Return")
rescaledDataset.AAPL.plot()
plt.grid(True);
plt.legend()
plt.show()
```

Output



5. Evaluate algorithms and models

5.1. Train-test split. The portfolio is divided into training and test sets to perform the analysis regarding the best portfolio and to perform backtesting:

```
# Dividing the dataset into training and testing sets
percentage = int(len(rescaledDataset) * 0.8)
X_train = rescaledDataset[:percentage]
X_test = rescaledDataset[percentage:]

stock_tickers = rescaledDataset.columns.values
n_tickers = len(stock_tickers)
```

5.2. Model evaluation: applying principal component analysis. As the next step, we create a function to perform PCA using the sklearn library. This function generates the principal components from the data that will be used for further analysis:

```
pca = PCA()
PrincipalComponent=pca.fit(X_train)
```

5.2.1. Explained variance using PCA. In this step, we look at the variance explained using PCA. The decline in the amount of variance of the original data explained by each principal component reflects the extent of correlation among the original features. The first principal component captures the most variance in the original data, the second component is a representation of the second highest variance, and so on. The eigenvectors with the lowest eigenvalues describe the least amount of variation within the dataset. Therefore, these values can be dropped.

The following charts show the number of principal components and the variance explained by each.

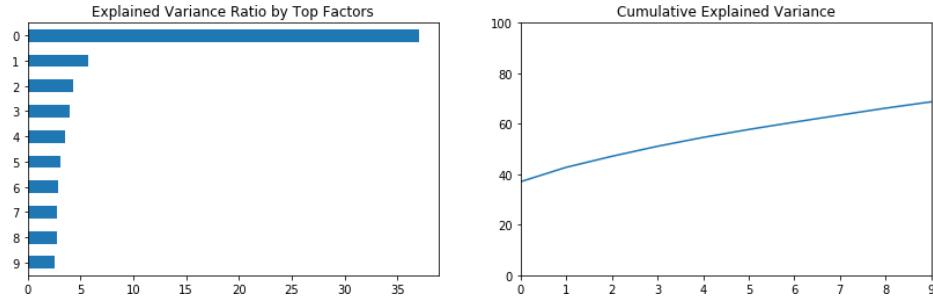
```
NumEigenvalues=20
fig, axes = plt.subplots(ncols=2, figsize=(14,4))
Series1 = pd.Series(pca.explained_variance_ratio_[:NumEigenvalues]).sort_values()
```

```

Series2 = pd.Series(pca.explained_variance_ratio_[:NumEigenvalues]).cumsum()
Series1.plot.barh(title='Explained Variance Ratio by Top Factors', ax=axes[0]);
Series1.plot(ylim=(0,1), ax=axes[1], title='Cumulative Explained Variance');

```

Output



We find that the most important factor explains around 40% of the daily return variation. This dominant principal component is usually interpreted as the “market” factor. We will discuss the interpretation of this and the other factors when looking at the portfolio weights.

The plot on the right shows the cumulative explained variance and indicates that around ten factors explain 73% of the variance in returns of the 28 stocks analyzed.

5.2.2. Looking at portfolio weights. In this step, we look more closely at the individual principal components. These may be less interpretable than the original features. However, we can look at the weights of the factors on each principal component to assess any intuitive themes relative to the 28 stocks. We construct five portfolios, defining the weights of each stock as each of the first five principal components. We then create a scatterplot that visualizes an organized descending plot with the respective weight of every company at the current chosen principal component:

```

def PCWeights():
    #Principal Components (PC) weights for each 28 PCs

    weights = pd.DataFrame()
    for i in range(len(pca.components_)):
        weights["weights_{}".format(i)] = \
            pca.components_[i] / sum(pca.components_[i])
    weights = weights.values.T
    return weights
weights=PCWeights()
sum(pca.components_[0])

```

Output

```
-5.247808242068631
```

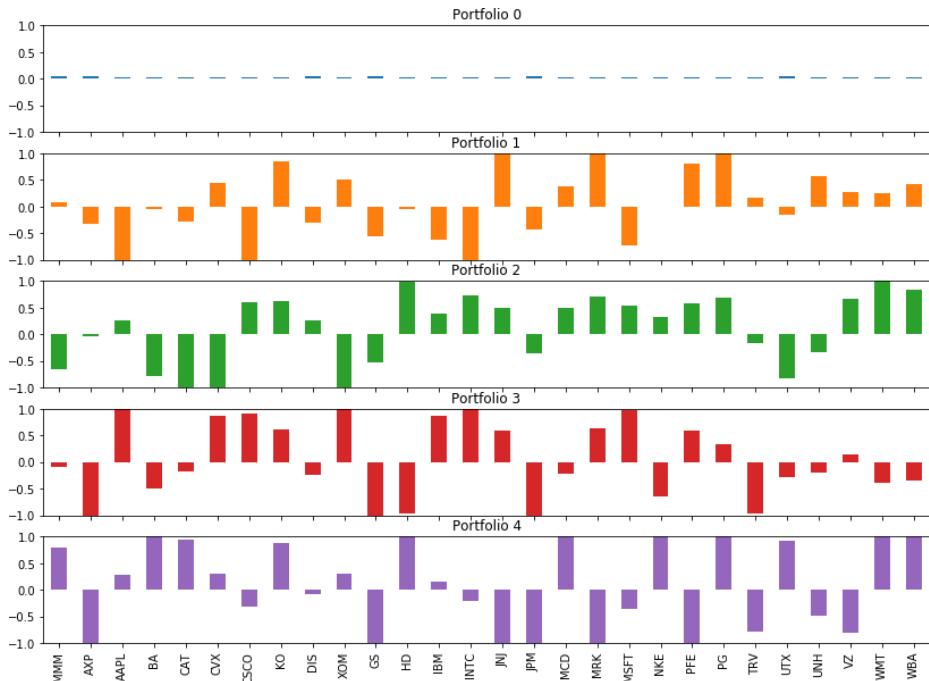
```

NumComponents=5
topPortfolios = pd.DataFrame(pca.components_[:NumComponents],\
    columns=dataset.columns)
eigen_portfolios = topPortfolios.div(topPortfolios.sum(1), axis=0)
eigen_portfolios.index = [f'Portfolio {i}' for i in range( NumComponents)]
np.sqrt(pca.explained_variance_)
eigen_portfolios.T.plot.bar(subplots=True, layout=(int(NumComponents),1), \
    figsize=(14,10), legend=False, sharey=True, ylim= (-1,1))

```

Given that scale for the plots are the same, we can also look at the heatmap as follows:

Output

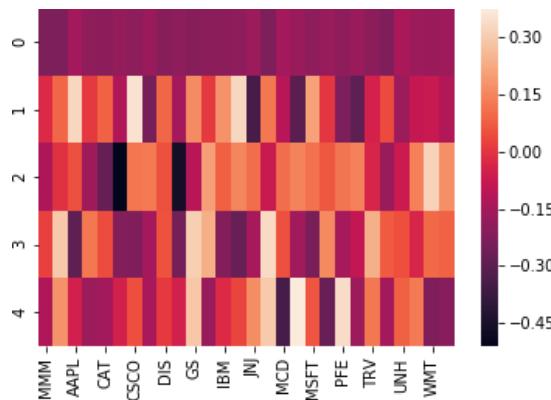


```

# plotting heatmap
sns.heatmap(topPortfolios)

```

Output



The heatmap and barplots show the contribution of different stocks in each eigenvector.

Traditionally, the intuition behind each principal portfolio is that it represents some sort of independent risk factor. The manifestation of those risk factors depends on the assets in the portfolio. In our case study, the assets are all U.S. domestic equities. The principal portfolio with the largest variance is typically a systematic risk factor (i.e., “market” factor). Looking at the first principal component (*Portfolio 0*), we see that the weights are distributed homogeneously across the stocks. This nearly equal weighted portfolio explains 40% of the variance in the index and is a fair representation of a systematic risk factor.

The rest of the eigen portfolios typically correspond to sector or industry factors. For example, *Portfolio 1* assigns a high weight to JNJ and MRK, which are stocks from the health care sector. Similarly, *Portfolio 3* has high weights on technology and electronics companies, such as AAPL, MSFT, and IBM.

When the asset universe for our portfolio is expanded to include broad, global investments, we may identify factors for international equity risk, interest rate risk, commodity exposure, geographic risk, and many others.

In the next step, we find the best eigen portfolio.

5.2.3. Finding the best eigen portfolio. To determine the best eigen portfolio, we use the *Sharpe ratio*. This is an assessment of risk-adjusted performance that explains the annualized returns against the annualized volatility of a portfolio. A high Sharpe ratio explains higher returns and/or lower volatility for the specified portfolio. The annualized Sharpe ratio is computed by dividing the annualized returns against the annualized volatility. For annualized return we apply the geometric average of all the returns

in respect to the periods per year (days of operations in the exchange in a year). Annualized volatility is computed by taking the standard deviation of the returns and multiplying it by the square root of the periods per year.

The following code computes the Sharpe ratio of a portfolio:

```
# Sharpe Ratio Calculation
# Calculation based on conventional number of trading days per year (i.e., 252).
def sharpe_ratio(ts_returns, periods_per_year=252):
    n_years = ts_returns.shape[0]/ periods_per_year
    annualized_return = np.power(np.prod(1+ts_returns), (1/n_years))-1
    annualized_vol = ts_returns.std() * np.sqrt(periods_per_year)
    annualized_sharpe = annualized_return / annualized_vol

    return annualized_return, annualized_vol, annualized_sharpe
```

We construct a loop to compute the principal component weights for each eigen portfolio. Then it uses the Sharpe ratio function to look for the portfolio with the highest Sharpe ratio. Once we know which portfolio has the highest Sharpe ratio, we can visualize its performance against the index for comparison:

```
def optimizedPortfolio():
    n_portfolios = len(pca.components_)
    annualized_ret = np.array([0.] * n_portfolios)
    sharpe_metric = np.array([0.] * n_portfolios)
    annualized_vol = np.array([0.] * n_portfolios)
    highest_sharpe = 0
    stock_tickers = rescaledDataset.columns.values
    n_tickers = len(stock_tickers)
    pcs = pca.components_

    for i in range(n_portfolios):

        pc_w = pcs[i] / sum(pcs[i])
        eigen_prtfi = pd.DataFrame(data ={'weights': pc_w.squeeze()*100}, \
        index = stock_tickers)
        eigen_prtfi.sort_values(by=[ 'weights'], ascending=False, inplace=True)
        eigen_prti_returns = np.dot(X_train_raw.loc[:, eigen_prtfi.index], pc_w)
        eigen_prti_returns = pd.Series(eigen_prti_returns.squeeze(),\
        index=X_train_raw.index)
        er, vol, sharpe = sharpe_ratio(eigen_prti_returns)
        annualized_ret[i] = er
        annualized_vol[i] = vol
        sharpe_metric[i] = sharpe

    sharpe_metric= np.nan_to_num(sharpe_metric)

    # find portfolio with the highest Sharpe ratio
    highest_sharpe = np.argmax(sharpe_metric)

    print('Eigen portfolio #{} with the highest Sharpe. Return {:.2f}%,\
    vol = {:.2f}%, Sharpe = {:.2f}' %
```

```

(highest_sharpe,
annualized_ret[highest_sharpe]*100,
annualized_vol[highest_sharpe]*100,
sharpe_metric[highest_sharpe]))
```

```

fig, ax = plt.subplots()
fig.set_size_inches(12, 4)
ax.plot(sharpe_metric, linewidth=3)
ax.set_title('Sharpe ratio of eigen-portfolios')
ax.set_ylabel('Sharpe ratio')
ax.set_xlabel('Portfolios')

results = pd.DataFrame(data={'Return': annualized_ret,
'Vol': annualized_vol,
'Sharpe': sharpe_metric})
results.dropna(inplace=True)
results.sort_values(by=['Sharpe'], ascending=False, inplace=True)
print(results.head(5))

plt.show()

optimizedPortfolio()

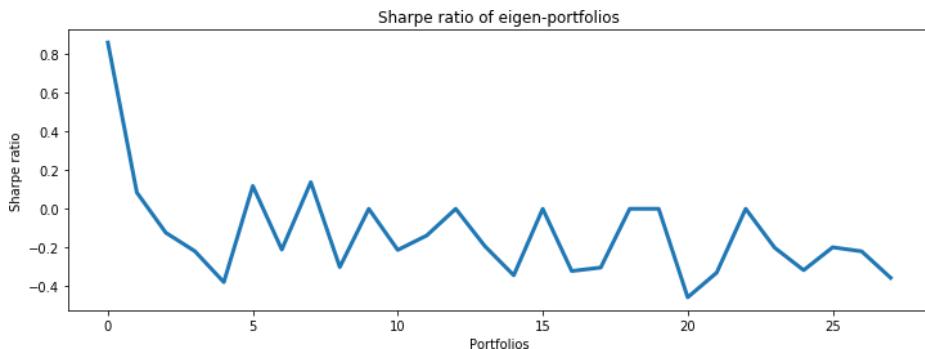
```

Output

```

Eigen portfolio #0 with the highest Sharpe. Return 11.47%, vol = 13.31%, \
Sharpe = 0.86
   Return    Vol  Sharpe
0    0.115  0.133  0.862
7    0.096  0.693  0.138
5    0.100  0.845  0.118
1    0.057  0.670  0.084

```



As shown by the results above, *Portfolio 0* is the best performing one, with the highest return *and* the lowest volatility. Let us look at the composition of this portfolio:

```

weights = PCWeights()
portfolio = portfolio = pd.DataFrame()

```

```

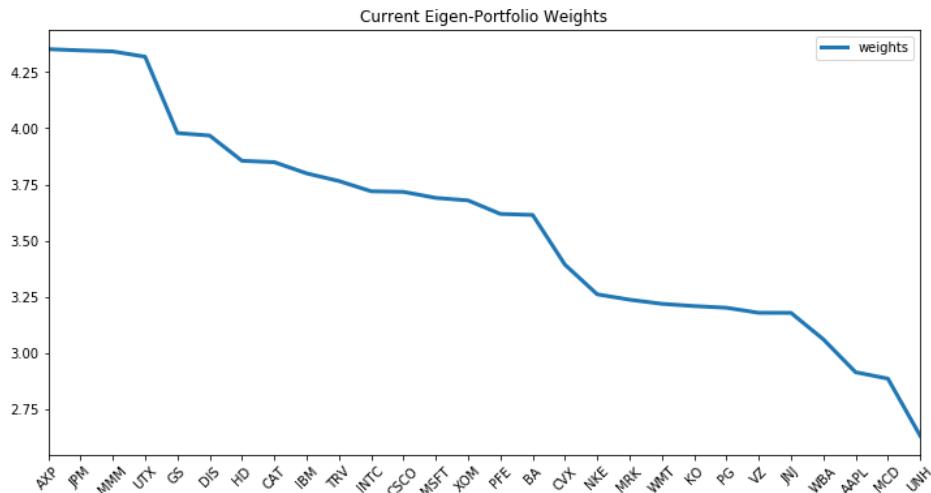
def plotEigen(weights, plot=False, portfolio=portfolio):
    portfolio = pd.DataFrame(data ={'weights': weights.squeeze() * 100}, \
    index = stock_tickers)
    portfolio.sort_values(by=['weights'], ascending=False, inplace=True)
    if plot:
        portfolio.plot(title='Current Eigen-Portfolio Weights',
                      figsize=(12, 6),
                      xticks=range(0, len(stock_tickers), 1),
                      rot=45,
                      linewidth=3
                     )
    plt.show()

    return portfolio

# Weights are stored in arrays, where 0 is the first PC's weights.
plotEigen(weights=weights[0], plot=True)

```

Output



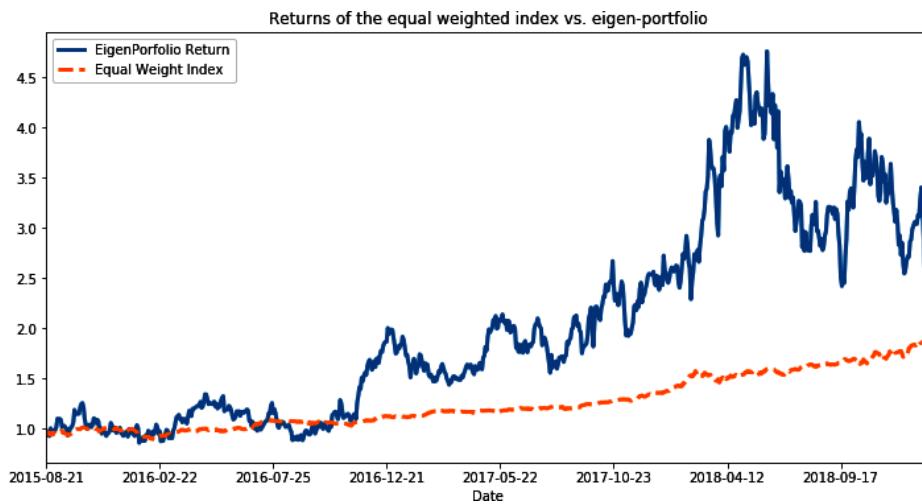
Recall that this is the portfolio that explains 40% of the variance and represents the systematic risk factor. Looking at the portfolio weights (in percentages in the y-axis), they do not vary much and are in the range of 2.7% to 4.5% across all stocks. However, the weights seem to be higher in the financial sector, and stocks such as AXP, JPM, and GS have higher-than-average weights.

5.2.4. Backtesting the eigen portfolios. We will now try to backtest this algorithm on the test set. We will look at a few of the top performers and the worst performer. For the top performers we look at the 3rd- and 4th-ranked eigen portfolios (*Portfolios 5* and *1*), while the worst performer reviewed was ranked 19th (*Portfolio 14*):

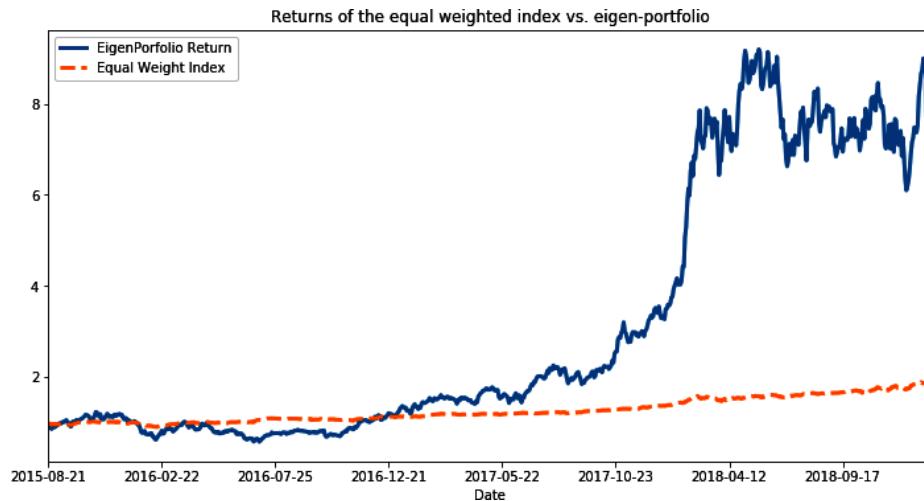
```
def Backtest(eigen):  
    """  
    Plots principal components returns against real returns.  
    """  
  
    eigen_prtfi = pd.DataFrame(data ={'weights': eigen.squeeze()}, \  
                               index=stock_tickers)  
    eigen_prtfi.sort_values(by=['weights'], ascending=False, inplace=True)  
  
    eigen_prti_returns = np.dot(X_test_raw.loc[:, eigen_prtfi.index], eigen)  
    eigen_portfolio_returns = pd.Series(eigen_prti_returns.squeeze(), \  
                                         index=X_test_raw.index)  
    returns, vol, sharpe = sharpe_ratio(eigen_portfolio_returns)  
    print('Current Eigen-Portfolio:\nReturn = %.2f%\nVolatility = %.2f%\n\n' \  
         'Sharpe = %.2f' % (returns * 100, vol * 100, sharpe))  
    equal_weight_return=(X_test_raw * (1/len(pca.components_))).sum(axis=1)  
    df_plot = pd.DataFrame({'EigenPorfolio Return': eigen_portfolio_returns, \  
                           'Equal Weight Index': equal_weight_return}, index=X_test.index)  
    np.cumprod(df_plot + 1).plot(title='Returns of the equal weighted\  
        index vs. First eigen-portfolio',  
        figsize=(12, 6), linewidth=3)  
    plt.show()  
  
Backtest(eigen=weights[5])  
Backtest(eigen=weights[1])  
Backtest(eigen=weights[14])
```

Output

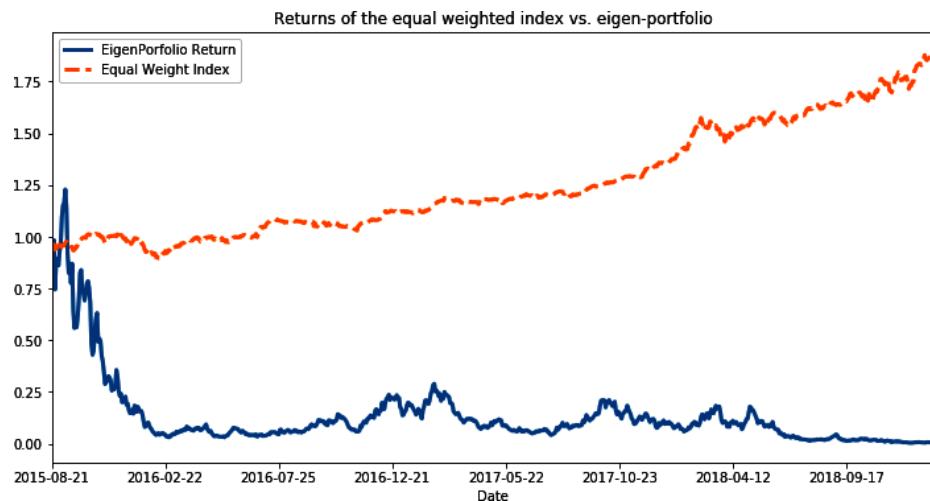
```
Current Eigen-Portfolio:  
Return = 32.76%  
Volatility = 68.64%  
Sharpe = 0.48
```



Current Eigen-Portfolio:
 Return = 99.80%
 Volatility = 58.34%
 Sharpe = 1.71



Current Eigen-Portfolio:
 Return = -79.42%
 Volatility = 185.30%
 Sharpe = -0.43



As shown in the preceding charts, the eigen portfolio return of the top portfolios outperforms the equally weighted index. The eigen portfolio ranked 19th underperformed the market significantly in the test set. The outperformance and underperformance are attributed to the weights of the stocks or sectors in the eigen portfolio. We can drill down further to understand the individual drivers of each portfolio. For example, *Portfolio 1* assigns high weight to several stocks in the health care sector, as discussed previously. This sector saw a significant increase in 2017 onwards, which is reflected in the chart for *Eigen Portfolio 1*.

Given that these eigen portfolios are independent, they also provide diversification opportunities. As such, we can invest across these uncorrelated eigen portfolios, providing other potential portfolio management benefits.

Conclusion

In this case study, we applied dimensionality reduction techniques in the context of portfolio management, using eigenvalues and eigenvectors from PCA to perform asset allocation.

We demonstrated that, while some interpretability is lost, the intuition behind the resulting portfolios can be matched to risk factors. In this example, the first eigen portfolio represented a systematic risk factor, while others exhibited sector or industry concentration.

Through backtesting, we found that the portfolio with the best result on the training set also achieved the strongest performance on the test set. Several of the portfolios outperformed the index based on the Sharpe ratio, the risk-adjusted performance metric used in this exercise.

Overall, we found that using PCA and analyzing eigen portfolios can yield a robust methodology for asset allocation and portfolio management.

Case Study 2: Yield Curve Construction and Interest Rate Modeling

A number of problems in portfolio management, trading, and risk management require a deep understanding and modeling of yield curves.

A yield curve represents interest rates, or yields, across a range of maturities, usually depicted in a line graph, as discussed in “[Case Study 4: Yield Curve Prediction](#)” on [page 141](#) in [Chapter 5](#). Yield curve illustrates the “price of funds” at a given point in time and, due to the time value of money, often shows interest rates rising as a function of maturity.

Researchers in finance have studied the yield curve and found that shifts or changes in the shape of the yield curve are attributable to a few unobservable factors. Specifically, empirical studies reveal that more than 99% of the movement of various U.S. Treasury bond yields are captured by three factors, which are often referred to as level, slope, and curvature. The names describe how each influences the yield curve shape in response to a shock. A level shock changes the interest rates of all maturities by almost identical amounts, inducing a *parallel shift* that changes the level of the entire curve up or down. A shock to the slope factor changes the difference in short-term and long-term rates. For instance, when long-term rates increase by a larger amount than do short-term rates, it results in a curve that becomes steeper (i.e., visually, the curve becomes more upward sloping). Changes in the short- and long-term rates can also produce a flatter yield curve. The main effects of the shock to the curvature factor focuses on medium-term interest rates, leading to hump, twist, or U-shaped characteristics.

Dimensionality reduction breaks down the movement of the yield curve into these three factors. Reducing the yield curve into fewer components means we can focus on a few intuitive dimensions in the yield curve. Traders and risk managers use this technique to condense the curve in risk factors for hedging the interest rate risk. Similarly, portfolio managers then have fewer dimensions to analyze when allocating funds. Interest rate structurers use this technique to model the yield curve and analyze its shape. Overall, it promotes faster and more effective portfolio management, trading, hedging, and risk management.

In this case study, we use PCA to generate typical movements of a yield curve and show that the first three principal components correspond to a yield curve’s level, slope, and curvature, respectively.

This case study will focus on:

- Understanding the intuition behind eigenvectors.
- Using dimensions resulting from dimensionality reduction to reproduce the original data.



Blueprint for Using Dimensionality Reduction to Generate a Yield Curve

1. Problem definition

Our goal in this case study is to use dimensionality reduction techniques to generate the typical movements of a yield curve.

The data used for this case study is obtained from [Quandl](#), a premier source for financial, economic, and alternative datasets. We use the data of 11 tenors (or maturities), from 1-month to 30-years, of Treasury curves. These are of daily frequency and are available from 1960 onwards.

2. Getting started—loading the data and Python packages

2.1. Loading the Python packages. The loading of Python packages is similar to the previous dimensionality reduction case study. Please refer to the Jupyter notebook of this case study for more details.

2.2. Loading the data. In the first step, we load the data of different tenors of the Treasury curves from Quandl:

```
# In order to use quandl, ApiConfig.api_key will need to be
# set to identify you to the quandl API. Please see API
# Documentation of quandl for more details
quandl.ApiConfig.api_key = 'API Key'

treasury = ['FRED/DGS1MO', 'FRED/DGS3MO', 'FRED/DGS6MO', 'FRED/DGS1', \
'FRED/DGS2', 'FRED/DGS3', 'FRED/DGSS', 'FRED/DGS7', 'FRED/DGS10', \
'FRED/DGS20', 'FRED/DGS30']

treasury_df = quandl.get(treasury)
treasury_df.columns = ['TRESY1mo', 'TRESY3mo', 'TRESY6mo', 'TRESY1y', \
'TRESY2y', 'TRESY3y', 'TRESY5y', 'TRESY7y', 'TRESY10y', 'TRESY20y', 'TRESY30y']
dataset = treasury_df
```

3. Exploratory data analysis

Here, we will take our first look at the data.

3.1. Descriptive statistics. In the next step we look at the shape of the dataset:

```
# shape  
dataset.shape
```

Output

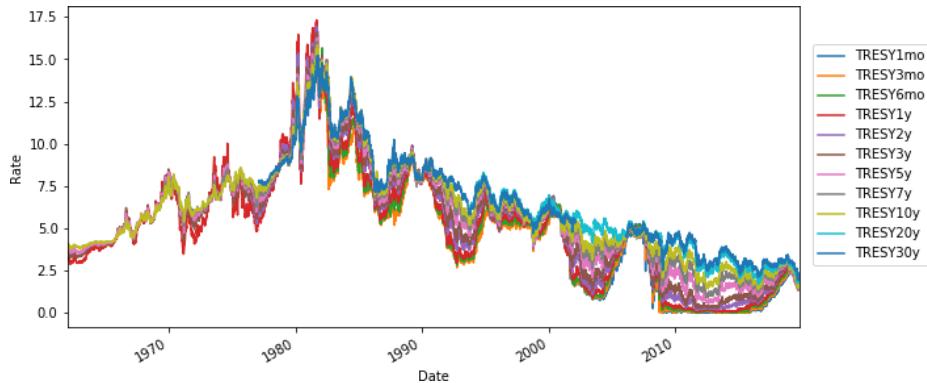
```
(14420, 11)
```

The dataset has 14,420 rows and has the data of 11 tenors of the Treasury curve for more than 50 years.

3.2. Data visualization. Let us look at the movement of the rates from the downloaded data:

```
dataset.plot(figsize=(10,5))  
plt.ylabel("Rate")  
plt.legend(bbox_to_anchor=(1.01, 0.9), loc=2)  
plt.show()
```

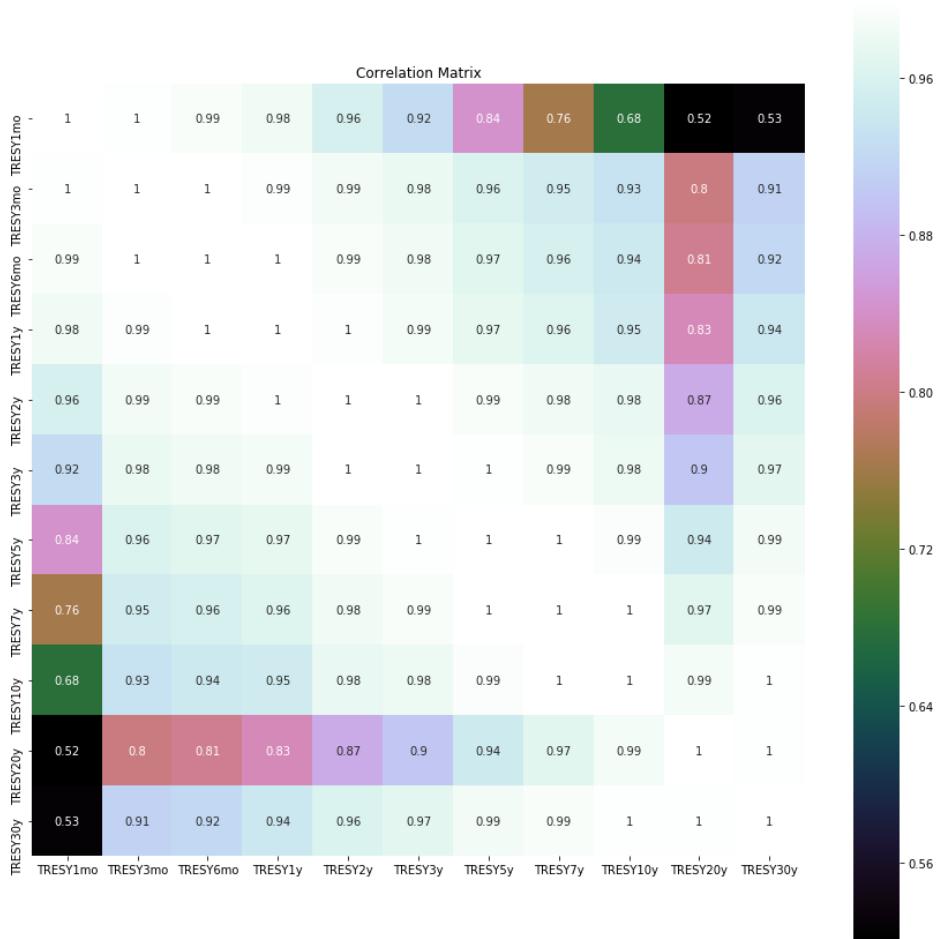
Output



In the next step we look at the correlations across tenors:

```
# correlation  
correlation = dataset.corr()  
plt.figure(figsize=(15, 15))  
plt.title('Correlation Matrix')  
sns.heatmap(correlation, vmax=1, square=True, annot=True, cmap='cubehelix')
```

Output



There is a significant positive correlation between the tenors, as you can see in the output (full-size version available on [GitHub](#)). This is an indication that reducing the number dimensions may be useful when modeling with the data. Additional visualizations of the data will be performed after implementing the dimensionality reduction models.

4. Data preparation

Data cleaning and transformation are a necessary modeling prerequisite in this case study.

4.1. Data cleaning. Here, we check for NAs in the data and either drop them or fill them with the mean of the column.

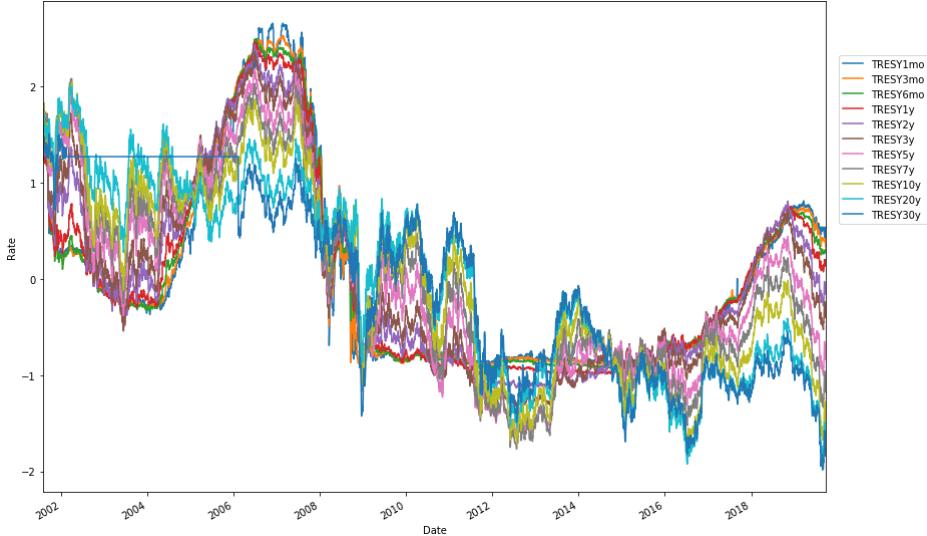
4.2. Data transformation. We standardize the variables on the same scale before applying PCA in order to prevent a feature with large values from dominating the result. We use the `StandardScaler` function in `sklearn` to standardize the dataset's features onto a unit scale (mean = 0 and variance = 1):

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler().fit(dataset)
rescaledDataset = pd.DataFrame(scaler.fit_transform(dataset),\
columns = dataset.columns,
index = dataset.index)
# summarize transformed data
dataset.dropna(how='any', inplace=True)
rescaledDataset.dropna(how='any', inplace=True)
```

Visualizing the standardized dataset

```
rescaledDataset.plot(figsize=(14, 10))
plt.ylabel("Rate")
plt.legend(bbox_to_anchor=(1.01, 0.9), loc=2)
plt.show()
```

Output



5. Evaluate algorithms and models

5.2. Model evaluation—applying principal component analysis. As a next step, we create a function to perform PCA using the sklearn library. This function generates the principal components from the data that will be used for further analysis:

```
pca = PCA()  
PrincipalComponent=pca.fit(rescaledDataset)
```

5.2.1. Explained variance using PCA.

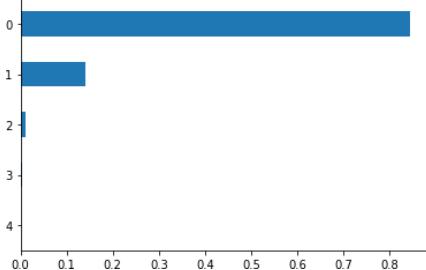
```
NumEigenvalues=5  
fig, axes = plt.subplots(ncols=2, figsize=(14, 4))  
pd.Series(pca.explained_variance_ratio_[:NumEigenvalues]).sort_values().\\  
plot.barh(title='Explained Variance Ratio by Top Factors',ax=axes[0]);  
pd.Series(pca.explained_variance_ratio_[:NumEigenvalues]).cumsum()\\  
.plot(ylim=(0,1),ax=axes[1], title='Cumulative Explained Variance');  
# explained variance  
pd.Series(np.cumsum(pca.explained_variance_ratio_)).to_frame\\  
('Explained Variance_Top 5').head(NumEigenvalues).style.format('{:, .2%}'.format)
```

Output

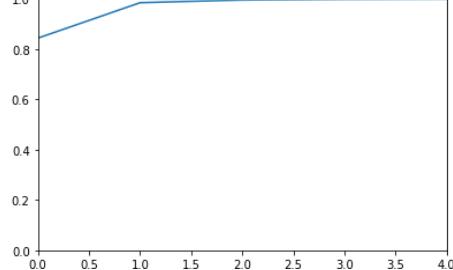
Explained Variance_Top 5

0	84.36%
1	98.44%
2	99.53%
3	99.83%
4	99.94%

Explained Variance Ratio by Top Factors



Cumulative Explained Variance



The first three principal components account for 84.4%, 14.08%, and 1.09% of variance, respectively. Cumulatively, they describe over 99.5% of all movement in the data. This is an incredibly efficient reduction in dimensions. Recall that in the first case study, we saw the first 10 components account for only 73% of variance.

5.2.2. Intuition behind the principal components. Ideally, we can have some intuition and interpretation of these principal components. To explore this, we first have a function to determine the weights of each principal component, and then perform the visualization of the principal components:

```
def PCWeights():
    """
    Principal Components (PC) weights for each 28 PCs
    """
    weights = pd.DataFrame()

    for i in range(len(pca.components_)):
        weights["weights_{}".format(i)] = \
            pca.components_[i] / sum(pca.components_[i])

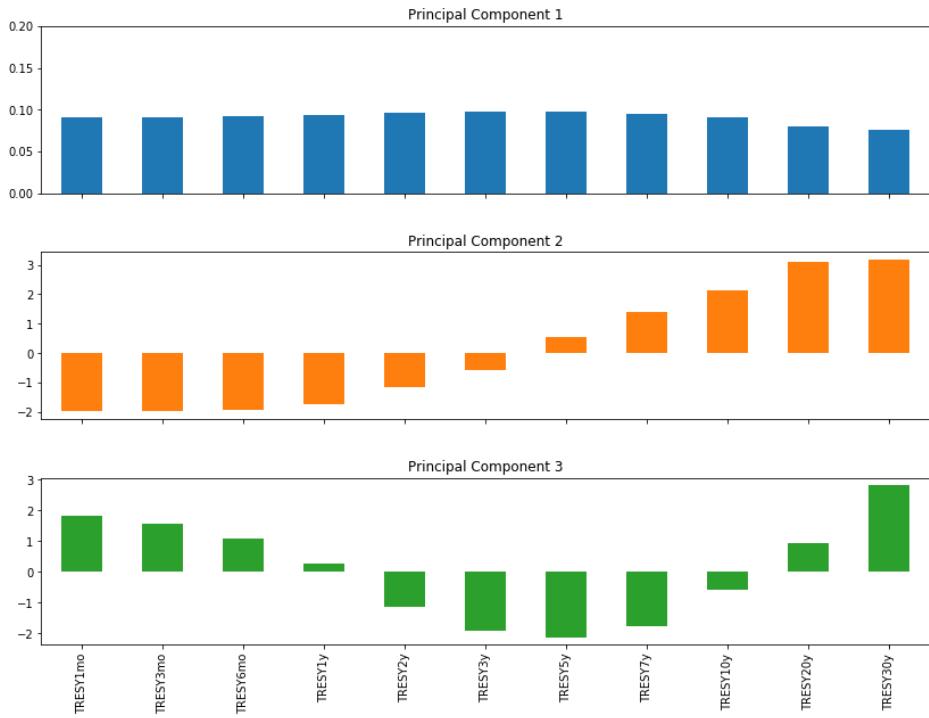
    weights = weights.values.T
    return weights

weights=PCWeights()
weights = PCWeights()
NumComponents=3

topPortfolios = pd.DataFrame(weights[:NumComponents], columns=dataset.columns)
topPortfolios.index = [f'Principal Component {i}' \
    for i in range(1, NumComponents+1)]

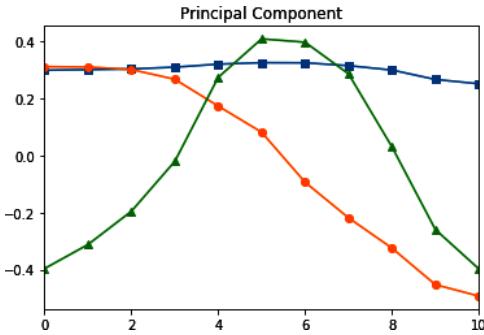
axes = topPortfolios.T.plot.bar(subplots=True, legend=False, figsize=(14, 10))
plt.subplots_adjust(hspace=0.35)
axes[0].set_ylim(0, .2);
```

Output



```
pd.DataFrame(pca.components_[0:3].T).plot(style= ['s-', 'o-', '^-'], \n                                             legend=False, title="Principal Component")
```

Output



By plotting the components of the eigenvectors we can make the following interpretation:

Principal Component 1

This eigenvector has all positive values, with all tenors weighted in the same direction. This means that the first principal component reflects movements that cause all maturities to move in the same direction, corresponding to *directional movements* in the yield curve. These are movements that shift the entire yield curve up or down.

Principal Component 2

The second eigenvector has the first half of the components negative and the second half positive. Treasury rates on the short end (long end) of the curve are weighted positively (negatively). This means that the second principal component reflects movements that cause the short end to go in one direction and the long end in the other. Consequently, it represents *slope movements* in the yield curve.

Principal Component 3

The third eigenvector has the first third of the components negative, the second third positive, and the last third negative. This means that the third principal component reflects movements that cause the short and long end to go in one direction, and the middle to go in the other, resulting in *curvature movements* of the yield curve.

5.2.3. Reconstructing the curve using principal components. One of the key features of PCA is the ability to reconstruct the initial dataset using the outputs of PCA. Using simple matrix reconstruction, we can generate a near exact replica of the initial data:

```
pca.transform(rescaledDataset)[:, :2]
```

Output

```
array([[ 4.97514826, -0.48514999],
       [ 5.03634891, -0.52005102],
       [ 5.14497849, -0.58385444],
       ...,
       [-1.82544584,  2.82360062],
       [-1.69938513,  2.6936174 ],
       [-1.73186029,  2.73073137]])
```

Mechanically, PCA is just a matrix multiplication:

$$Y = XW$$

where Y is the principal components, X is input data, and W is a matrix of coefficients, which we can use to recover the original matrix as per the equation below:

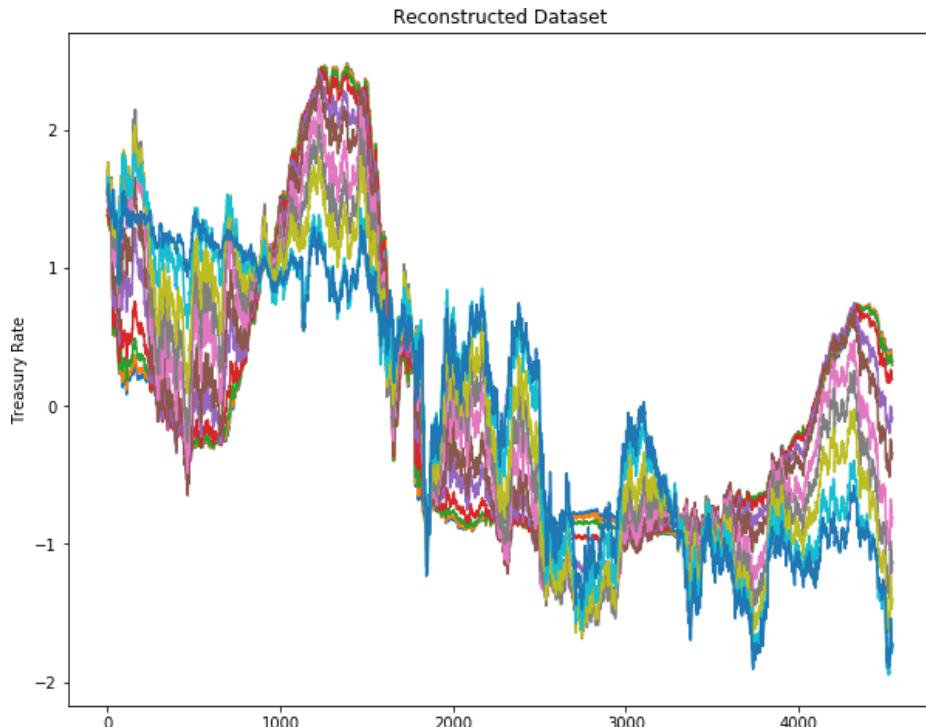
$$X = YW'$$

where W' is the inverse of the matrix of coefficients W .

```
nComp=3
reconst= pd.DataFrame(np.dot(pca.transform(rescaledDataset)[:, :nComp], \
pca.components_[:nComp,:]),columns=dataset.columns)
plt.figure(figsize=(10,8))
plt.plot(reconst)
plt.ylabel("Treasury Rate")
plt.title("Reconstructed Dataset")
plt.show()
```

This figure shows the replicated Treasury rate chart and demonstrates that, using just the first three principal components, we are able to replicate the original chart. Despite reducing the data from 11 dimensions to three, we still retain more than 99% of the information and can reproduce the original data easily. Additionally, we also have intuition around these three drivers of yield curve moments. Reducing the yield curve into fewer components means practitioners can focus on fewer factors that influence interest rates. For example, in order to hedge a portfolio, it may be sufficient to protect the portfolio against moves in the first three principal components only.

Output



Conclusion

In this case study, we introduced dimensionality reduction to break down the Treasury rate curve into fewer components. We saw that the principal components are quite intuitive for this case study. The first three principal components explain more than 99.5% of the variation and represent directional movements, slope movements, and curvature movements, respectively.

By using principal component analysis, analyzing the eigenvectors, and understanding the intuition behind them, we demonstrated how using dimensionality reduction led to fewer intuitive dimensions in the yield curve. Such dimensionality reduction of the yield curve can potentially lead to faster and more effective portfolio management, trading, hedging, and risk management.

Case Study 3: Bitcoin Trading: Enhancing Speed and Accuracy

As trading becomes more automated, traders will continue to seek to use as many features and technical indicators as they can to make their strategies more accurate and efficient. One of the many challenges in this is that adding more variables leads to ever more complexity, making it increasingly difficult to arrive at solid conclusions. Using dimensionality reduction techniques, we can compress many features and technical indicators into a few logical collections, while still maintaining a significant amount of the variance of the original data. This helps speed up model training and tuning. Additionally, it helps prevent overfitting by getting rid of correlated variables, which can ultimately cause more harm than good. Dimensionality reduction also enhances exploration and visualization of a dataset to understand grouping or relationships, an important task when building and continuously monitoring trading strategies.

In this case study, we will use dimensionality reduction to enhance “[Case Study 3: Bitcoin Trading Strategy](#)” on page 179 presented in [Chapter 6](#). In this case study, we design a trading strategy for bitcoin that considers the relationship between the short-term and long-term prices to predict a buy or sell signal. We create several new intuitive, technical indicator features, including trend, volume, volatility, and momentum. We apply dimensionality reduction techniques on these features in order to achieve better results.

In this case study, we will focus on:

- Reducing the dimensions of a dataset to yield better and faster results for supervised learning.
- Using SVD and t-SNE to visualize data in lower dimensions.



Blueprint for Using Dimensionality Reduction to Enhance a Trading Strategy

1. Problem definition

Our goal in this case study is to use dimensionality reduction techniques to enhance an algorithmic trading strategy. The data and the variables used in this case study are the same as in “[Case Study 3: Bitcoin Trading Strategy](#)” on page 179. For reference, we are using intraday bitcoin price data, volume, and weighted bitcoin price from January 2012 to October 2017. Steps 3 and 4 presented in this case study use the same steps as the case study in [Chapter 6](#). As such, these steps are condensed in this case study to avoid repetition.

2. Getting started—loading the data and Python packages

2.1. Loading the Python packages. The Python packages used for this case study are the same as those presented in the previous two case studies in this chapter.

3. Exploratory data analysis

Refer to “[3. Exploratory data analysis](#)” on page 181 for more details of this step.

4. Data preparation

We prepare the data for modeling in the following sections.

4.1. Data cleaning. We clean the data by filling the NAs with the last available values:

```
dataset[dataset.columns] = dataset[dataset.columns].ffill()
```

4.2. Preparing the data for classification. We attach the following label to each movement: 1 if the short-term price increases compared to the long-term price; 0 if the short-term price decreases compared to the long-term price. This label is assigned to

a variable we will call *signal*, which is the predicted variable for this case study. Let us look at the data for prediction:

```
dataset.tail(5)
```

Output

	Open	High	Low	Close	Volume_(BTC)	Volume_(Currency)	Weighted_Price	short_mavg	long_mavg	signal
2841372	2190.49	2190.49	2181.37	2181.37	1.700	3723.785	2190.247	2179.259	2189.616	0.0
2841373	2190.50	2197.52	2186.17	2195.63	6.561	14402.812	2195.206	2181.622	2189.877	0.0
2841374	2195.62	2197.52	2191.52	2191.83	15.663	34361.024	2193.792	2183.605	2189.943	0.0
2841375	2195.82	2216.00	2195.82	2203.51	27.090	59913.493	2211.621	2187.018	2190.204	0.0
2841376	2201.70	2209.81	2196.98	2208.33	9.962	21972.309	2205.649	2190.712	2190.510	1.0

The dataset contains the signal column along with all other columns.

4.3. Feature engineering. In this step, we construct a dataset that contains the predictors that will be used to make the signal prediction. Using the bitcoin intraday price data, including daily open, high, low, close, and volume, we compute the following technical indicators:

- Moving Average
- Stochastic Oscillator %K and %D
- Relative Strength Index (RSI)
- Rate Of Change (ROC)
- Momentum (MOM)

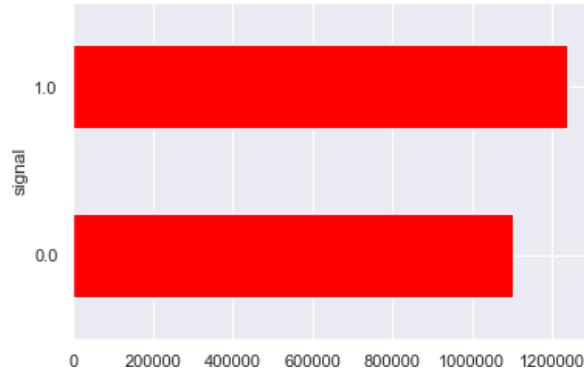
The code for the construction of all of the indicators, along with their descriptions, is presented in [Chapter 6](#). The final dataset and the columns used are as follows:

Close	Volume_(BTC)	Weighted_Price	signal	EMA10	EMA30	EMA200	ROC10	ROC30	MOM10	...	RSI200	%K10	%D10	%K30
2181.37	1.700	2190.247	0.0	2181.181	2182.376	2211.244	0.431	-0.649	8.42	...	46.613	56.447	73.774	47.883
2195.63	6.561	2195.206	0.0	2183.808	2183.231	2211.088	1.088	-0.062	23.63	...	47.638	93.687	71.712	93.805
2191.83	15.663	2193.792	0.0	2185.266	2183.786	2210.897	1.035	-0.235	19.83	...	47.395	80.995	77.043	81.350
2203.51	27.090	2211.621	0.0	2188.583	2185.058	2210.823	1.479	0.297	34.13	...	48.213	74.205	82.963	74.505
2208.33	9.962	2205.649	1.0	2192.174	2186.560	2210.798	1.626	0.516	36.94	...	48.545	82.810	79.337	84.344

4.4. Data visualization. Let us look at the distribution of the predicted variable:

```
fig = plt.figure()
plot = dataset.groupby(['signal']).size().plot(kind='barh', color='red')
plt.show()
```

Output



The predicted signal is “buy” 52.9% of the time.

5. Evaluate algorithms and models

Next, we perform dimensionality reduction and evaluate the models.

5.1. Train-test split. In this step, we split the dataset into training and test sets:

```
Y= subset_dataset["signal"]
X = subset_dataset.loc[:, dataset.columns != 'signal'] validation_size = 0.2
X_train, X_validation, Y_train, Y_validation = train_test_split\
(X, Y, test_size=validation_size, random_state=1)
```

We standardize the variables on the same scale before applying dimensionality reduction. Data standardization is performed using the following Python code:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler().fit(X_train)
rescaledDataset = pd.DataFrame(scaler.fit_transform(X_train),\
columns = X_train.columns, index = X_train.index)
# summarize transformed data
X_train.dropna(how='any', inplace=True)
rescaledDataset.dropna(how='any', inplace=True)
rescaledDataset.head(2)
```

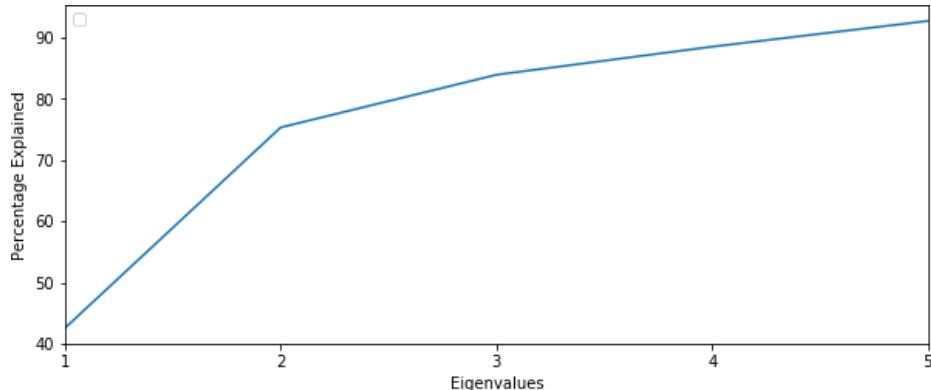
Output

	Close	Volume_(BTC)	Weighted_Price	EMA10	EMA30	EMA200	ROC10	ROC30	MOM10	MOM30	...	RSI200	%K10	%D10	%K30	%D30	
2834071	1.072	-0.367		1.040	1.064	1.077	1.014	0.005	-0.159	0.009	-0.183	...	-0.325	1.322	0.427	-0.205	-0.412
2836517	-1.738	1.126		-1.714	-1.687	-1.653	-1.733	-0.533	-0.597	-0.066	-0.416	...	-0.465	-1.620	-0.511	-1.283	-0.970

5.2. Singular value decomposition (feature reduction). Here we will use SVD to perform PCA. Specifically, we are using the TruncatedSVD method in the sklearn package to transform the full dataset into a representation using the top five components:

```
ncomps = 5
svd = TruncatedSVD(n_components=ncomps)
svd_fit = svd.fit(rescaledDataset)
Y_pred = svd_fit_transform(rescaledDataset)
ax = pd.Series(svd_fit.explained_variance_ratio_.cumsum()).plot(kind='line', \
figsize=(10, 3))
ax.set_xlabel("Eigenvalues")
ax.set_ylabel("Percentage Explained")
print('Variance preserved by first 5 components == {:.2%}'.\
format(svd_fit.explained_variance_ratio_.cumsum()[-1]))
```

Output



Following the computation, we preserve 92.75% of the variance by using just five components rather than the full 25+ original features. This is a tremendously useful compression for the analysis and iterations of the model.

For convenience, we will create a Python dataframe specifically for these top five components:

```
dfsrd = pd.DataFrame(Y_pred, columns=['c{}'.format(c) for \
c in range(ncmps)], index=rescaledDataset.index)
print(dfsrd.shape)
dfsrd.head()
```

Output

```
(8000, 5)
```

	c0	c1	c2	c3	c4
2834071	-2.252	1.920	0.538	-0.019	-0.967
2836517	5.303	-1.689	-0.678	0.473	0.643
2833945	-2.315	-0.042	1.697	-1.704	1.672
2835048	-0.977	0.782	3.706	-0.697	0.057
2838804	2.115	-1.915	0.475	-0.174	-0.299

5.2.1. Basic visualization of reduced features.

Let us visualize the compressed dataset:

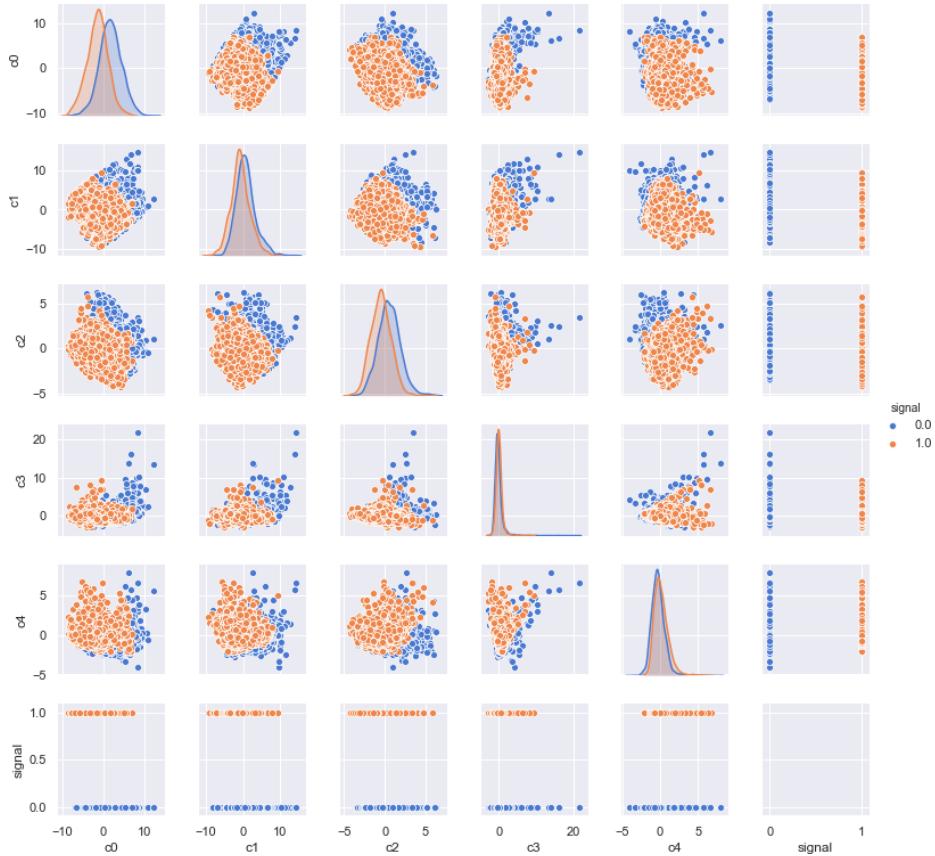
```
svdcols = [c for c in dfsrd.columns if c[0] == 'c']
```

Pairs-plots

Pairs-plots are a simple representation of a set of 2D scatterplots, with each component plotted against every other component. The data points are colored according to their signal classification:

```
plotdims = 5
plotrows = 1
dfsrdplot = dfsrd[svdcols].iloc[:, :plotdims]
dfsrdplot['signal']=Y_train
ax = sns.pairplot(dfsrdplot.iloc[::plotrows, :], hue='signal', size=1.8)
```

Output



We can see that there is clear separation of the colored dots (full color version available on [GitHub](#)), meaning that data points from the same signal tend to cluster together. The separation is more distinct for the first components, with the characteristics of signal distributions growing more similar as you progress from the first to the fifth component. That said, the plot provides support for using all five components in our model.

5.3. t-SNE visualization. In this step, we implement t-SNE and look at the related visualization. We will use the basic implementation available in Scikit-learn:

```
tsne = TSNE(n_components=2, random_state=0)

Z = tsne.fit_transform(dfsvd[svdcols])
dftsne = pd.DataFrame(Z, columns=['x', 'y'], index=dfsvd.index)
```

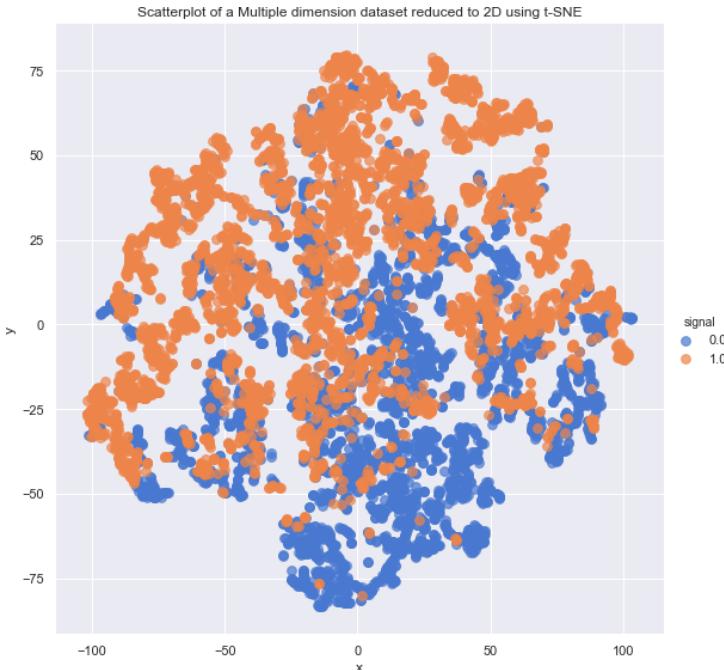
```

dftsne['signal'] = Y_train

g = sns.lmplot('x', 'y', dftsne, hue='signal', fit_reg=False, size=8
                , scatter_kws={'alpha':0.7,'s':60})

```

Output



The plot shows us that there is a good degree of clustering for the trading signal. There is some overlap of the long and short signals, but they can be distinguished quite well using the reduced number of features.

5.4. Compare models with and without dimensionality reduction. In this step, we analyze the impact of the dimensionality reduction on the classification and the impact on the overall accuracy and computation time:

```

# test options for classification
scoring = 'accuracy'

```

5.4.1. Models. We first look at the time taken by the model without dimensionality reduction, where we have all the technical indicators:

```

import time
start_time = time.time()

# spot-check the algorithms

```

```

models = RandomForestClassifier(n_jobs=-1)
cv_results_XTrain= cross_val_score(models, X_train, Y_train, cv=kfold, \
    scoring=scoring)
print("Time Without Dimensionality Reduction--- %s seconds ---" % \
    (time.time() - start_time))

```

Output

```

Time Without Dimensionality Reduction
7.781347990036011 seconds

```

The total time taken without dimensionality reduction is around eight seconds. Let us look at the time it takes with dimensionality reduction, when only the five principal components from the truncated SVD are used:

```

start_time = time.time()
X_SVD= dfsvd[svdcols].iloc[:, :5]
cv_results_SVD = cross_val_score(models, X_SVD, Y_train, cv=kfold, \
    scoring=scoring)
print("Time with Dimensionality Reduction--- %s seconds ---" % \
    (time.time() - start_time))

```

Output

```

Time with Dimensionality Reduction
2.281977653503418 seconds

```

The total time taken with dimensionality reduction is around two seconds—four times a reduction in time, which is a significant improvement. Let us investigate whether there is any decline in the accuracy when using the condensed dataset:

```

print("Result without dimensionality Reduction: %f (%f)" %\
    (cv_results_XTrain.mean(), cv_results_XTrain.std()))
print("Result with dimensionality Reduction: %f (%f)" %\
    (cv_results_SVD.mean(), cv_results_SVD.std()))

```

Output

```

Result without dimensionality Reduction: 0.936375 (0.010774)
Result with dimensionality Reduction: 0.887500 (0.012698)

```

Accuracy declines roughly 5%, from 93.6% to 88.7%. The improvement in speed has to be balanced against this loss in accuracy. Whether the loss in accuracy is acceptable likely depends on the problem. If this is a model that needs to be recalibrated very frequently, then a lower computation time will be essential, especially when handling large, high-velocity datasets. The improvement in the computation time does have other benefits, especially in the early stages of trading strategy development. It enables us to test a greater number of features (or technical indicators) in less time.

Conclusion

In this case study, we demonstrated the efficiency of dimensionality reduction and principal components analysis in reducing the number of dimensions in the context

of a trading strategy. Through dimensionality reduction, we achieved a commensurate accuracy rate with a fourfold improvement in the modeling speed. In trading strategy development involving expansive datasets, such speed enhancements can lead to improvements for the entire process.

We demonstrated that both SVD and t-SNE yield reduced datasets that can easily be visualized for evaluating trading signal data. This allowed us to distinguish the long and short signals of this trading strategy in ways not possible with the original number of features.

Chapter Summary

The case studies presented in this chapter focused on understanding the concepts of the different dimensionality reduction methods, developing intuition around the principal components, and visualizing the condensed datasets.

Overall, the concepts in Python, machine learning, and finance presented in this chapter through the case studies can be used as a blueprint for any other dimensionality reduction-based problem in finance.

In the next chapter, we explore concepts and case studies for another type of unsupervised learning—clustering.

Exercises

1. Using dimensionality reduction, extract the different factors from the stocks within a different index and use them to build a trading strategy.
2. Pick any of the regression-based case studies in [Chapter 5](#) and use dimensionality reduction to see whether there is any improvement in computation time. Explain the components using the factor loading and develop some high-level intuition of them.
3. For case study 3 presented in this chapter, perform factor loading of the principal components and understand the intuition of the different components.
4. Get the principal components of different currency pairs or different commodity prices. Identify the drivers of the primary principal components and link them to some intuitive macroeconomic variables.

Unsupervised Learning: Clustering

In the previous chapter, we explored dimensionality reduction, which is one type of unsupervised learning. In this chapter, we will explore *clustering*, a category of unsupervised learning techniques that allows us to discover hidden structures in data.

Both clustering and dimensionality reduction summarize the data. Dimensionality reduction compresses the data by representing it using new, fewer features while still capturing the most relevant information. Similarly, clustering is a way to reduce the volume of data and find patterns. However, it does so by categorizing the original data and not by creating new variables. Clustering algorithms assign observations to subgroups that consist of similar data points. The goal of clustering is to find a natural grouping in data so that items in a given cluster are more similar to each other than to those of different clusters. Clustering serves to better understand the data through the lens of several categories or groups created. It also permits the automatic categorization of new objects according to the learned criteria.

In the field of finance, clustering has been used by traders and investment managers to find homogeneous groups of assets, classes, sectors, and countries based on similar characteristics. Clustering analysis augments trading strategies by providing insights into categories of trading signals. The technique has been used to segment customers or investors into a number of groups to better understand their behavior and to perform additional analysis.

In this chapter, we will discuss fundamental clustering techniques and introduce three case studies in the areas of portfolio management and trading strategy development.

In “[Case Study 1: Clustering for Pairs Trading](#)” on page 243, we use clustering methods to select pairs of stocks for a trading strategy. A *pairs trading strategy* involves matching a long position with a short position in two financial instruments

that are closely related. Finding appropriate pairs can be a challenge when the number of instruments is high. In this case study, we demonstrate how clustering can be a useful technique in trading strategy development and other similar situations.

In “[Case Study 2: Portfolio Management: Clustering Investors](#)” on page 259, we identify clusters of investors with similar abilities and willingness to take risks. We show how clustering techniques can be used for effective asset allocation and portfolio rebalancing. This illustrates how part of the portfolio management process can be automated, which is immensely useful for investment managers and robo-advisors alike.

In “[Case Study 3: Hierarchical Risk Parity](#)” on page 267, we use a clustering-based algorithm to allocate capital into different asset classes and compare the results against other portfolio allocation techniques.

In this chapter, we will learn about the following concepts related to clustering techniques:

- Basic concepts of models and techniques used for clustering.
- How to implement different clustering techniques in Python.
- How to effectively perform visualizations of clustering outcomes.
- Understanding the intuitive meaning of clustering results.
- How to choose the right clustering techniques for a problem.
- Selecting the appropriate number of clusters in different clustering algorithms.
- Building hierarchical clustering trees using Python.



This Chapter's Code Repository

A Python-based master template for clustering, along with the Jupyter notebook for the case studies presented in this chapter are in [Chapter 8 - Unsup. Learning - Clustering](#) in the code repository for this book. To work through any machine learning problems in Python involving the models for clustering (such as k -means, hierarchical clustering, etc.) presented in this chapter, readers simply need to modify the template to align with their problem statement. Similar to the previous chapters, the case studies presented in this chapter use the standard Python master template with the standardized model development steps presented in [Chapter 2](#). For the clustering case studies, steps 6 (Model Tuning and Grid Search) and 7 (Finalizing the Model) have merged with step 5 (Evaluate Algorithms and Models).

Clustering Techniques

There are many types of clustering techniques, and they differ with respect to their strategy of identifying groupings. Choosing which technique to apply depends on the nature and structure of the data. In this chapter, we will cover the following three clustering techniques:

- k -means clustering
- Hierarchical clustering
- Affinity propagation clustering

The following section summarizes these clustering techniques, including their strengths and weaknesses. Additional details for each of the clustering methods are provided in the case studies.

k-means Clustering

k -means is the most well-known clustering technique. The algorithm of k -means aims to find and group data points into classes that have high similarity between them. This similarity is understood as the opposite of the distance between data points. The closer the data points are, the more likely they are to belong to the same cluster.

The algorithm finds k centroids and assigns each data point to exactly one cluster with the goal of minimizing the within-cluster variance (called *inertia*). It typically uses the Euclidean distance (ordinary distance between two points), but other distance metrics can be used. The k -means algorithm delivers a local optimum for a given k and proceeds as follows:

1. This algorithm specifies the number of clusters.
2. Data points are randomly selected as cluster centers.
3. Each data point is assigned to the cluster center it is nearest to.
4. Cluster centers are updated to the mean of the assigned points.
5. Steps 3–4 are repeated until all cluster centers remain unchanged.

In simple terms, we randomly move around the specified number of centroids in each iteration, assigning each data point to the closest centroid. Once we have done that, we calculate the mean distance of all points in each centroid. Then, once we can no longer reduce the minimum distance from data points to their respective centroids, we have found our clusters.

k-means hyperparameters

The k -means hyperparameters include:

Number of clusters

The number of clusters and centroids to generate.

Maximum iterations

Maximum iterations of the algorithm for a single run.

Number initial

The number of times the algorithm will be run with different centroid seeds. The final result will be the best output of the defined number of consecutive runs, in terms of inertia.

With k -means, different random starting points for the cluster centers often result in very different clustering solutions. Therefore, the k -means algorithm is run in sklearn with at least 10 different random initializations, and the solution occurring the greatest number of times is chosen.

The strengths of k -means include its simplicity, wide range of applicability, fast convergence, and linear scalability to large data while producing clusters of an even size. It is most useful when we know the exact number of clusters, k , beforehand. In fact, a main weakness of k -means is having to tune this hyperparameter. Additional drawbacks include the lack of a guarantee to find a global optimum and its sensitivity to outliers.

Implementation in Python

Python's sklearn library offers a powerful implementation of k -means. The following code snippet illustrates how to apply k -means clustering on a dataset:

```
from sklearn.cluster import KMeans
#Fit with k-means
k_means = KMeans(n_clusters=nclust)
k_means.fit(X)
```

The number of clusters is the key hyperparameter to be tuned. We will look at the k -means clustering technique in case studies 1 and 2 of this chapter, in which further details on choosing the right number of clusters and detailed visualizations are provided.

Hierarchical Clustering

Hierarchical clustering involves creating clusters that have a predominant ordering from top to bottom. The main advantage of hierarchical clustering is that we do not need to specify the number of clusters; the model determines that by itself. This

clustering technique is divided into two types: agglomerative hierarchical clustering and divisive hierarchical clustering.

Agglomerative hierarchical clustering is the most common type of hierarchical clustering and is used to group objects based on their similarity. It is a “bottom-up” approach where each observation starts in its own cluster, and pairs of clusters are merged as one moves up the hierarchy. The agglomerative hierarchical clustering algorithm delivers a *local optimum* and proceeds as follows:

1. Make each data point a single-point cluster and form N clusters.
2. Take the two closest data points and combine them, leaving $N-1$ clusters.
3. Take the two closest clusters and combine them, forming $N-2$ clusters.
4. Repeat step 3 until left with only one cluster.

Divisive hierarchical clustering works “top-down” and sequentially splits the remaining clusters to produce the most distinct subgroups.

Both produce $N-1$ hierarchical levels and facilitate the clustering creation at the level that best partitions data into homogeneous groups. We will focus on the more common agglomerative clustering approach.

Hierarchical clustering enables the plotting of *dendograms*, which are visualizations of a binary hierarchical clustering. A dendrogram is a type of tree diagram showing hierarchical relationships between different sets of data. They provide an interesting and informative visualization of hierarchical clustering results. A dendrogram contains the memory of the hierarchical clustering algorithm, so you can tell how the cluster is formed simply by inspecting the chart.

Figure 8-1 shows an example of dendograms based on hierarchical clustering. The distance between data points represents dissimilarities, and the height of the blocks represents the distance between clusters.

Observations that fuse at the bottom are similar, while those at the top are quite different. With dendograms, conclusions are made based on the location of the vertical axis rather than on the horizontal one.

The advantages of hierarchical clustering are that it is easy to implement it, does not require one to specify the number of clusters, and it produces dendograms that are very useful in understanding the data. However, the time complexity for hierarchical clustering can result in long computation times relative to other algorithms, such as k -means. If we have a large dataset, it can be difficult to determine the correct number of clusters by looking at the dendrogram. Hierarchical clustering is very sensitive to outliers, and in their presence, model performance decreases significantly.

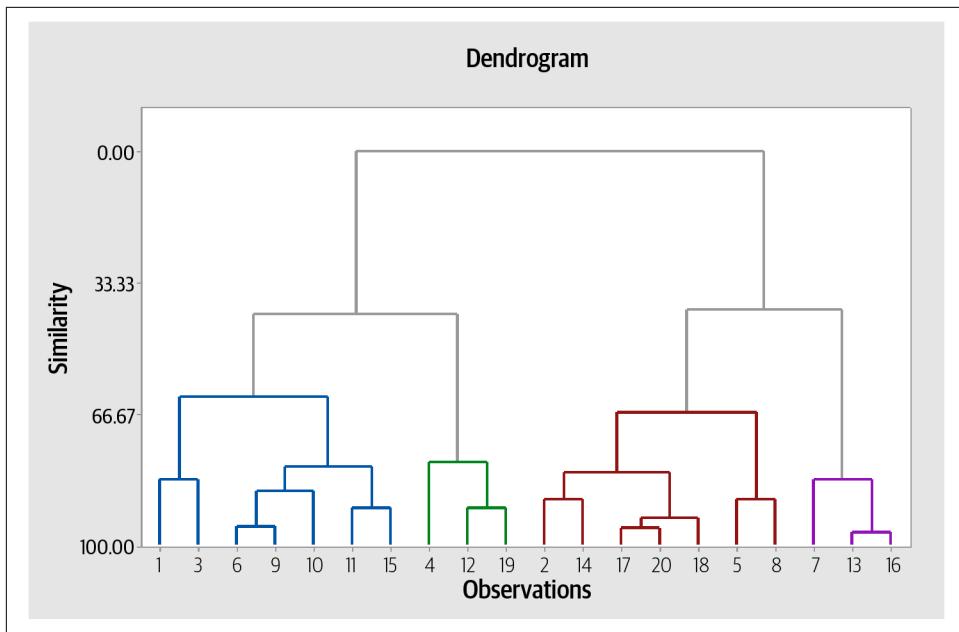


Figure 8-1. Hierarchical clustering

Implementation in Python

The following code snippet illustrates how to apply agglomerative hierarchical clustering with four clusters on a dataset:

```
from sklearn.cluster import AgglomerativeClustering
model = AgglomerativeClustering(n_clusters=4, affinity='euclidean',
                                 linkage='ward')
clust_labels1 = model.fit_predict(X)
```

More details regarding the hyperparameters of agglomerative hierarchical clustering can be found on the [sklearn website](#). We will look at the hierarchical clustering technique in case studies 1 and 3 in this chapter.

Affinity Propagation Clustering

Affinity propagation creates clusters by sending messages between data points until convergence. Unlike clustering algorithms such as *k*-means, affinity propagation does not require the number of clusters to be determined or estimated before running the algorithm. Two important parameters are used in affinity propagation to determine the number of clusters: the *preference*, which controls how many *exemplars* (or prototypes) are used; and the *damping factor*, which dampens the responsibility and availability of messages to avoid numerical oscillations when updating these messages.

A dataset is described using a small number of exemplars. These are members of the input set that are representative of clusters. The affinity propagation algorithm takes in a set of pairwise similarities between data points and finds clusters by maximizing the total similarity between data points and their exemplars. The messages sent between pairs represent the suitability of one sample to be the exemplar of the other, which is updated in response to the values from other pairs. This updating happens iteratively until convergence, at which point the final exemplars are chosen, and we obtain the final clustering.

In terms of strengths, affinity propagation does not require the number of clusters to be determined before running the algorithm. The algorithm is fast and can be applied to large similarity matrices. However, the algorithm often converges to suboptimal solutions, and at times it can fail to converge.

Implementation in Python

The following code snippet illustrates how to implement the affinity propagation algorithm for a dataset:

```
from sklearn.cluster import AffinityPropagation
# Initialize the algorithm and set the number of PC's
ap = AffinityPropagation()
ap.fit(X)
```

More details regarding the hyperparameters of affinity propagation clustering can be found on the [sklearn website](#). We will look at the affinity propagation technique in case studies 1 and 2 in this chapter.

Case Study 1: Clustering for Pairs Trading

A pairs trading strategy constructs a portfolio of correlated assets with similar market risk factor exposure. Temporary price discrepancies in these assets can create opportunities to profit through a long position in one instrument and a short position in another. A pairs trading strategy is designed to eliminate market risk and exploit these temporary discrepancies in the relative returns of stocks.

The fundamental premise in pairs trading is that *mean reversion* is an expected dynamic of the assets. This mean reversion should lead to a long-run equilibrium relationship, which we try to approximate through statistical methods. When moments of (presumably temporary) divergence from this long-term trend arise, one can possibly profit. The key to successful pairs trading is the ability to select the right pairs of assets to be used.

Traditionally, trial and error was used for pairs selection. Stocks or instruments that were merely in the same sector or industry were grouped together. The idea was that if these stocks were for companies in similar industries, their stocks should move

similarly as well. However, this was and is not necessarily the case. Additionally, with a large universe of stocks, finding a suitable pair is a difficult task, given that there are a total of $n(n-1)/2$ possible pairs, where n is the number of instruments. Clustering can be a useful technique here.

In this case study, we will use clustering algorithms to select pairs of stocks for a pairs trading strategy.

This case study will focus on:

- Evaluating three main clustering methods: k -means, hierarchical clustering, and affinity propagation clustering.
- Understanding approaches to finding the right number of clusters in k -means and hierarchical clustering.
- Visualizing data in the clusters, including viewing dendograms.
- Selecting the right clustering algorithm.



Blueprint for Using Clustering to Select Pairs

1. Problem definition

Our goal in this case study is to perform clustering analysis on the stocks in the S&P 500 to come up with pairs for a pairs trading strategy. S&P 500 stock data was obtained using `pandas_datareader` from Yahoo Finance. It includes price data from 2018 onwards.

2. Getting started—loading the data and Python packages

The list of the libraries used for data loading, data analysis, data preparation, and model evaluation are shown below.

2.1. Loading the Python packages. The details of most of these packages and functions have been provided in Chapters 2 and 4. The use of these packages will be demonstrated in different steps of the model development process.

Packages for clustering

```
from sklearn.cluster import KMeans, AgglomerativeClustering, AffinityPropagation  
from scipy.cluster.hierarchy import fcluster
```

```
from scipy.cluster.hierarchy import dendrogram, linkage, cophenet
from scipy.spatial.distance import pdist
from sklearn.metrics import adjusted_mutual_info_score
from sklearn import cluster, covariance, manifold
```

Packages for data processing and visualization

```
# Load libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from pandas import read_csv, set_option
from pandas.plotting import scatter_matrix
import seaborn as sns
from sklearn.preprocessing import StandardScaler
import datetime
import pandas_datareader as dr
import matplotlib.ticker as ticker
from itertools import cycle
```

2.2. Loading the data. The stock data is loaded below.¹

```
dataset = read_csv('SP500Data.csv', index_col=0)
```

3. Exploratory data analysis

We take a quick look at the data in this section.

3.1. Descriptive statistics. Let us look at the shape of the data:

```
# shape
dataset.shape
```

Output

```
(448, 502)
```

The data contains 502 columns and 448 observations.

3.2. Data visualization. We will take a detailed look into the visualization postclustering.

4. Data preparation

We prepare the data for modeling in the following sections.

¹ Refer to the Jupyter notebook to understand fetching price data using pandas_datareader.

4.1. Data cleaning. In this step, we check for NAs in the rows and either drop them or fill them with the mean of the column:

```
#Checking for any null values and removing the null values'''  
print('Null Values =',dataset.isnull().values.any())
```

Output

```
Null Values = True
```

Let us get rid of the columns with more than 30% missing values:

```
missing_fractions = dataset.isnull().mean().sort_values(ascending=False)  
missing_fractions.head(10)  
drop_list = sorted(list(missing_fractions[missing_fractions > 0.3].index))  
dataset.drop(labels=drop_list, axis=1, inplace=True)  
dataset.shape
```

Output

```
(448, 498)
```

Given that there are null values, we drop some rows:

```
# Fill the missing values with the last value available in the dataset.  
dataset=dataset.fillna(method='ffill')
```

The data cleaning steps identified those with missing values and populated them. This step is important for creating a meaningful, reliable, and clean dataset that can be used without any errors in the clustering.

4.2. Data transformation. For the purpose of clustering, we will be using *annual returns* and *variance* as the variables, as they are primary indicators of stock performance and volatility. The following code prepares these variables:

```
#Calculate average annual percentage return and volatilities  
returns = pd.DataFrame(dataset.pct_change().mean() * 252)  
returns.columns = ['Returns']  
returns['Volatility'] = dataset.pct_change().std() * np.sqrt(252)  
data = returns
```

All the variables should be on the same scale before applying clustering; otherwise, a feature with large values will dominate the result. We use `StandardScaler` in `sklearn` to standardize the dataset features onto unit scale (mean = 0 and variance = 1):

```
from sklearn.preprocessing import StandardScaler  
scaler = StandardScaler().fit(data)  
rescaledDataset = pd.DataFrame(scaler.fit_transform(data),\  
    columns = data.columns, index = data.index)  
# summarize transformed data  
rescaledDataset.head(2)
```

Output

	Returns	Volatility
ABT	0.794067	-0.702741

With the data prepared, we can now explore the clustering algorithms.

5. Evaluate algorithms and models

We will look at the following models:

- k -means
- Hierarchical clustering (agglomerative clustering)
- Affinity propagation

5.1. k -means clustering. Here, we model using k -means and evaluate two ways to find the optimal number of clusters.

5.1.1. Finding the optimal number of clusters. We know that k -means initially assigns data points to clusters randomly and then calculates centroids or mean values. Further, it calculates the distances within each cluster, squares these, and sums them to get the sum of squared errors.

The basic idea is to define k clusters so that the total within-cluster variation (or error) is minimized. The following two methods are useful in finding the number of clusters in k -means:

Elbow method

Based on the sum of squared errors (SSE) within clusters

Silhouette method

Based on the silhouette score

First, let's examine the elbow method. The SSE for each point is the square of the distance of the point from its representation (i.e., its predicted cluster center). The sum of squared errors is plotted for a range of values for the number of clusters. The first cluster will add much information (explain a lot of variance), but eventually the marginal gain will drop, giving an angle in the graph. The number of clusters is chosen at this point; hence it is referred to as the “elbow criterion.”

Let us implement this in Python using the sklearn library and plot the SSE for a range of values for k :

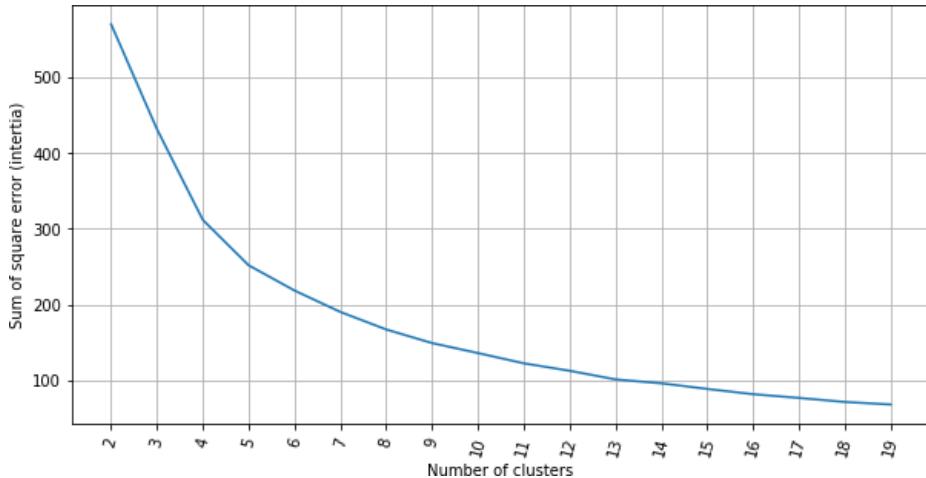
```
distortions = []
max_loop=20
for k in range(2, max_loop):
```

```

kmeans = KMeans(n_clusters=k)
kmeans.fit(X)
distortions.append(kmeans.inertia_)
fig = plt.figure(figsize=(15, 5))
plt.plot(range(2, max_loop), distortions)
plt.xticks([i for i in range(2, max_loop)], rotation=75)
plt.grid(True)

```

Output



Inspecting the sum of squared errors chart, it appears the elbow kink occurs around five or six clusters for this data. Certainly we can see that as the number of clusters increases past six, the SSE within clusters begins to plateau.

Now let's look at the silhouette method. The silhouette score measures how similar a point is to its own cluster (*cohesion*) compared to other clusters (*separation*). The range of the silhouette value is between 1 and -1. A high value is desirable and indicates that the point is placed in the correct cluster. If many points have a negative silhouette value, that may indicate that we have created too many or too few clusters.

Let us implement this in Python using the sklearn library and plot the silhouette score for a range of values for k :

```

from sklearn import metrics

silhouette_score = []
for k in range(2, max_loop):
    kmeans = KMeans(n_clusters=k, random_state=10, n_init=10, n_jobs=-1)
    kmeans.fit(X)
    silhouette_score.append(metrics.silhouette_score(X, kmeans.labels_, \
        random_state=10))
fig = plt.figure(figsize=(15, 5))

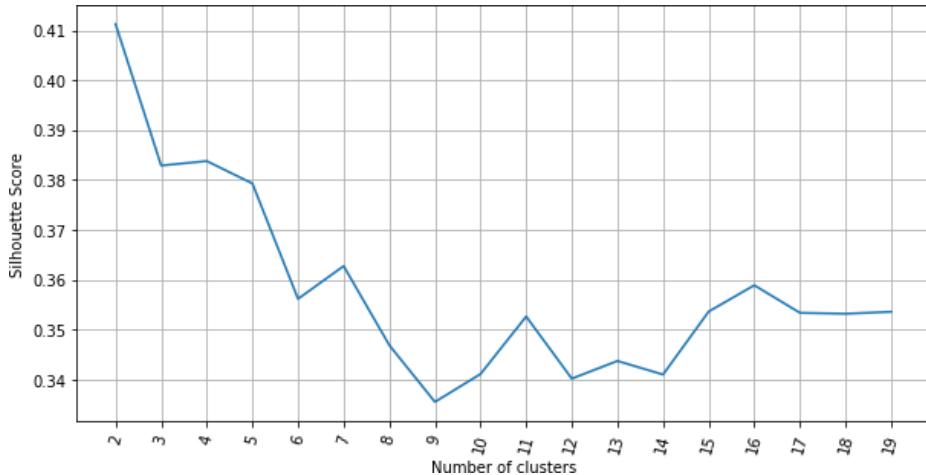
```

```

plt.plot(range(2, max_loop), silhouette_score)
plt.xticks([i for i in range(2, max_loop)], rotation=75)
plt.grid(True)

```

Output



Looking at the silhouette score chart, we can see that there are various parts of the graph at which a kink can be seen. Since there is not much of a difference in the SSE after six clusters, it implies that six clusters is a preferred choice in this k -means model.

Combining information from both methods, we infer the optimum number of clusters to be six.

5.1.2. Clustering and visualization. Let us build the k -means model with six clusters and visualize the results:

```

nclust=6
#Fit with k-means
k_means = cluster.KMeans(n_clusters=nclust)
k_means.fit(X)
#Extracting labels
target_labels = k_means.predict(X)

```

Visualizing how clusters are formed is no easy task when the number of variables in the dataset is very large. A basic scatterplot is one method for visualizing a cluster in a two-dimensional space. We create one below to identify the relationships inherent in our data:

```

centroids = k_means.cluster_centers_
fig = plt.figure(figsize=(16,10))
ax = fig.add_subplot(111)

```

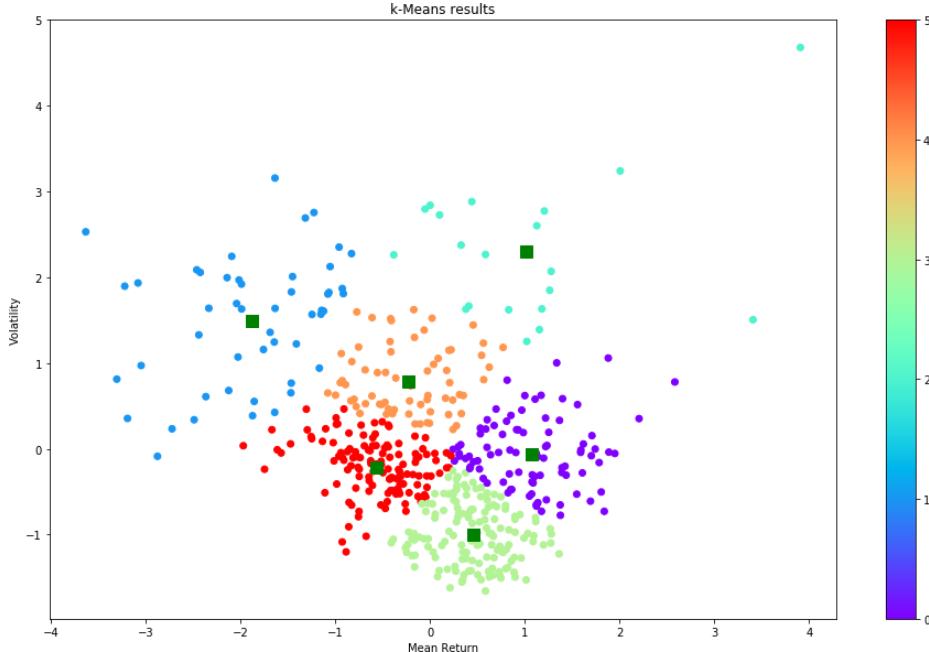
```

scatter = ax.scatter(X.iloc[:,0],X.iloc[:,1], c=k_means.labels_, \
    cmap="rainbow", label = X.index)
ax.set_title('k-means results')
ax.set_xlabel('Mean Return')
ax.set_ylabel('Volatility')
plt.colorbar(scatter)

plt.plot(centroids[:,0],centroids[:,1],'sg',markersize=11)

```

Output



In the preceding plot, we can somewhat see that there are distinct clusters separated by different colors (full-color version available on [GitHub](#)). The grouping of data in the plot seems to be separated quite well. There is also a degree of separation in the centroids of the clusters, represented by square dots.

Let us look at the number of stocks in each of the clusters:

```

# show number of stocks in each cluster
clustered_series = pd.Series(index=X.index, data=k_means.labels_.flatten())
# clustered stock with its cluster label
clustered_series_all = pd.Series(index=X.index, data=k_means.labels_.flatten())
clustered_series = clustered_series[clustered_series != -1]

plt.figure(figsize=(12,7))
plt.barh(

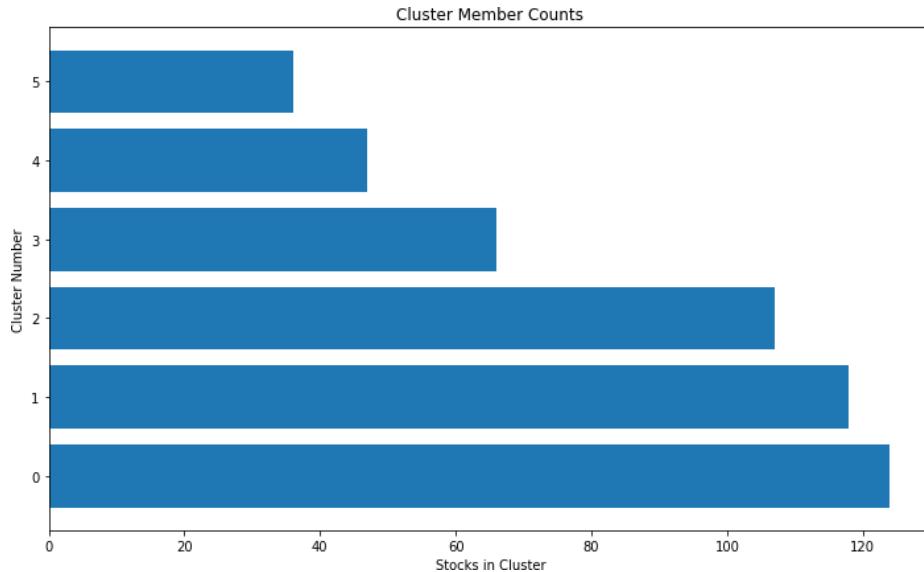
```

```

        range(len(clustered_series.value_counts()), # cluster labels, y axis
              clustered_series.value_counts())
    )
plt.title('Cluster Member Counts')
plt.xlabel('Stocks in Cluster')
plt.ylabel('Cluster Number')
plt.show()

```

Output



The number of stocks per cluster ranges from around 40 to 120. Although the distribution is not equal, we have a significant number of stocks in each cluster.

Let's look at the hierarchical clustering.

5.2. Hierarchical clustering (agglomerative clustering). In the first step, we look at the hierarchy graph and check for the number of clusters.

5.2.1. Building hierarchy graph/dendrogram. The hierarchy class has a dendrogram method that takes the value returned by the *linkage method* of the same class. The linkage method takes the dataset and the method to minimize distances as parameters. We use *ward* as the method since it minimizes the variance of distances between the clusters:

```
from scipy.cluster.hierarchy import dendrogram, linkage, ward

#Calculate linkage
Z= linkage(X, method='ward')
Z[0]
```

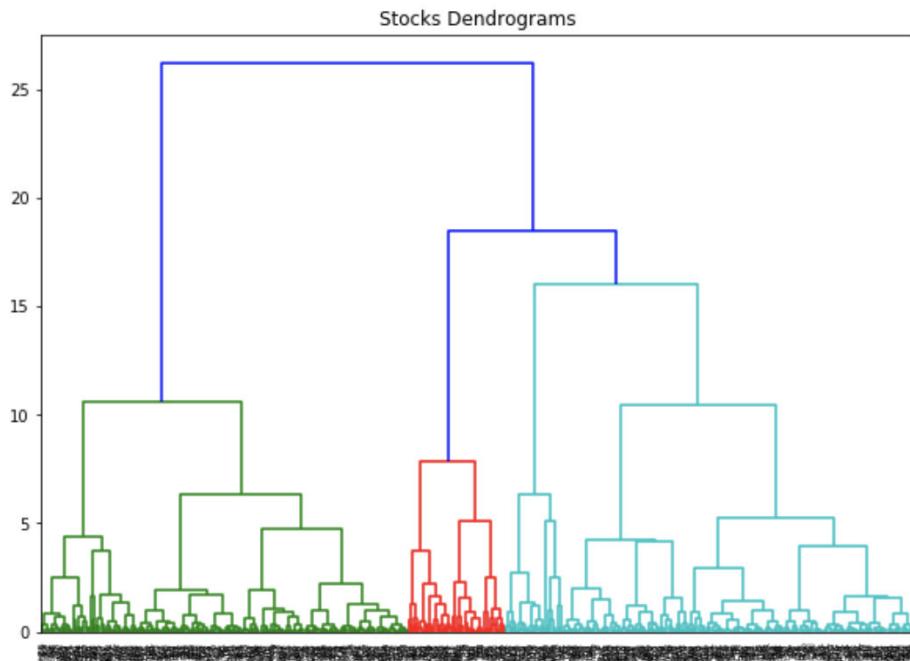
Output

```
array([3.3000000e+01, 3.1400000e+02, 3.62580431e-03, 2.0000000e+00])
```

The best way to visualize an agglomerative clustering algorithm is through a dendrogram, which displays a cluster tree, the leaves being the individual stocks and the root being the final single cluster. The distance between each cluster is shown on the y-axis. The longer the branches are, the less correlated the two clusters are:

```
#Plot Dendrogram
plt.figure(figsize=(10, 7))
plt.title("Stocks Dendograms")
dendrogram(Z,labels = X.index)
plt.show()
```

Output



This chart can be used to visually inspect the number of clusters that would be created for a selected distance threshold (although the names of the stocks on the horizontal axis are not very clear, we can see that they are grouped into several clusters). The number of vertical lines a hypothetical straight, horizontal line will pass through is the number of clusters created for that distance threshold value. For example, at a value of 20, the horizontal line would pass through two vertical branches of the dendrogram, implying two clusters at that distance threshold. All data points (leaves) from that branch would be labeled as that cluster that the horizontal line passed through.

Choosing a threshold cut at 13 yields four clusters, as confirmed in the following Python code:

```
distance_threshold = 13
clusters = fcluster(Z, distance_threshold, criterion='distance')
chosen_clusters = pd.DataFrame(data=clusters, columns=['cluster'])
chosen_clusters['cluster'].unique()
```

Output

```
array([1, 4, 3, 2], dtype=int64)
```

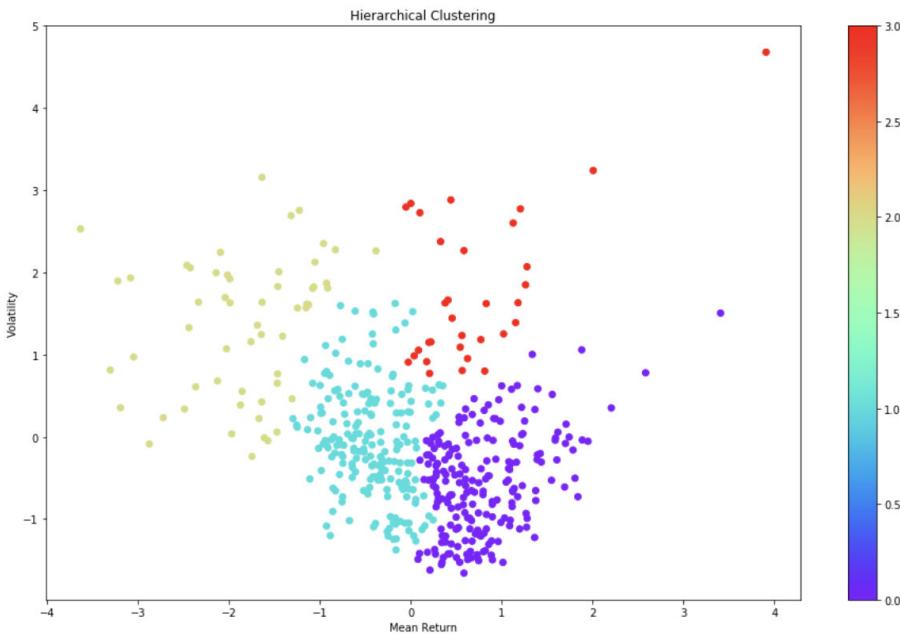
5.2.2. Clustering and visualization. Let us build the hierarchical clustering model with four clusters and visualize the results:

```
nclust = 4
hc = AgglomerativeClustering(n_clusters=nclust, affinity='euclidean', \
linkage='ward')
clust_labels1 = hc.fit_predict(X)

fig = plt.figure(figsize=(16,10))
ax = fig.add_subplot(111)
scatter = ax.scatter(X.iloc[:,0],X.iloc[:,1], c=clust_labels1, cmap="rainbow")
ax.set_title('Hierarchical Clustering')
ax.set_xlabel('Mean Return')
ax.set_ylabel('Volatility')
plt.colorbar(scatter)
```

Similar to the plot of k -means clustering, we see that there are some distinct clusters separated by different colors (full-size version available on [GitHub](#)).

Output



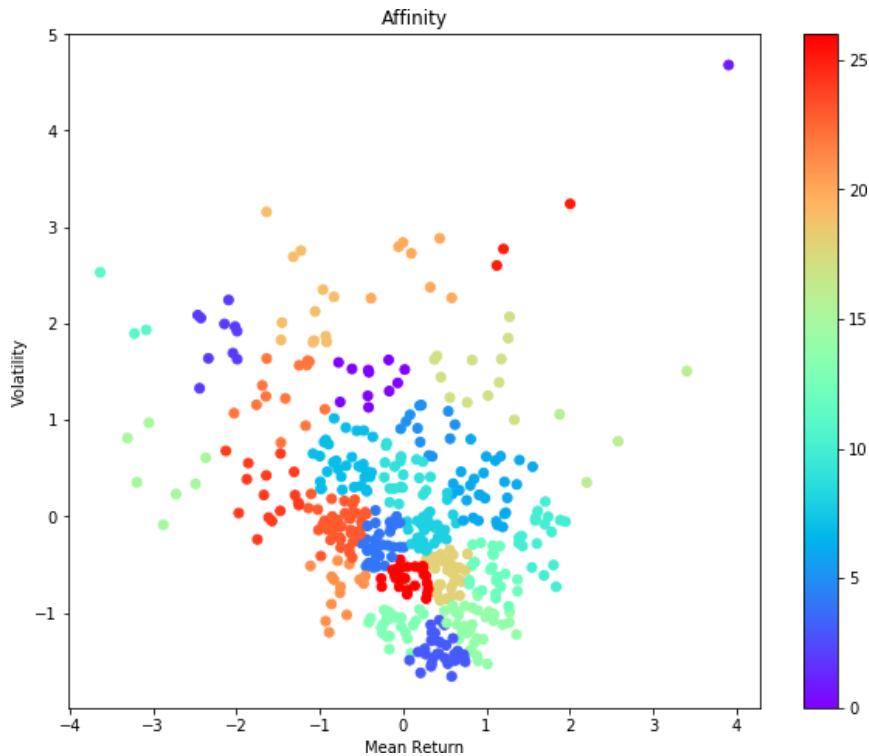
Now let us look at affinity propagation clustering.

5.3. Affinity propagation. Let us build the affinity propagation model and visualize the results:

```
ap = AffinityPropagation()
ap.fit(X)
clust_labels2 = ap.predict(X)

fig = plt.figure(figsize=(10,8))
ax = fig.add_subplot(111)
scatter = ax.scatter(X.iloc[:,0],X.iloc[:,1], c=clust_labels2, cmap="rainbow")
ax.set_title('Affinity')
ax.set_xlabel('Mean Return')
ax.set_ylabel('Volatility')
plt.colorbar(scatter)
```

Output



The affinity propagation model with the chosen hyperparameters produced many more clusters than k -means and hierarchical clustering. There is some clear grouping, but also more overlap due to the larger number of clusters (full-size version available on [GitHub](#)). In the next step, we will evaluate the clustering techniques.

5.4. Cluster evaluation. If the ground truth labels are not known, evaluation must be performed using the model itself. The silhouette coefficient (`sklearn.metrics.silhouette_score`) is one example we can use. A higher silhouette coefficient score implies a model with better defined clusters. The silhouette coefficient is computed for each of the clustering methods defined above:

```
from sklearn import metrics
print("km", metrics.silhouette_score(X, k_means.labels_, metric='euclidean'))
print("hc", metrics.silhouette_score(X, hc.fit_predict(X), metric='euclidean'))
print("ap", metrics.silhouette_score(X, ap.labels_, metric='euclidean'))
```

Output

```
km 0.3350720873411941
hc 0.3432149515640865
ap 0.3450647315156527
```

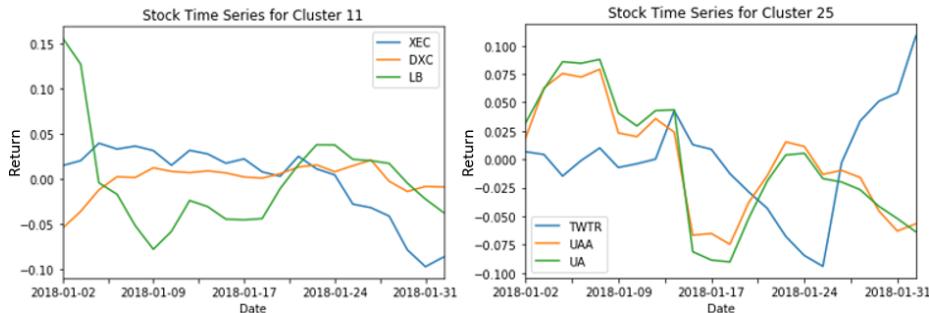
Given that affinity propagation performs the best, we proceed with affinity propagation and use 27 clusters as specified by this clustering method.

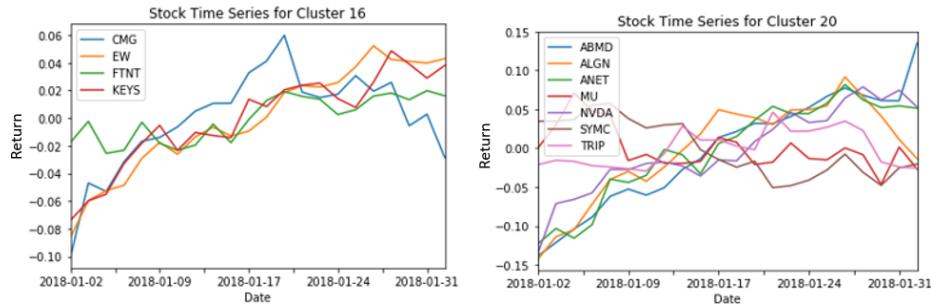
Visualizing the return within a cluster. We have the clustering technique and the number of clusters finalized, but we need to check whether the clustering leads to a sensible output. To do this, we visualize the historical behavior of the stocks in a few clusters:

```
# all stock with its cluster label (including -1)
clustered_series = pd.Series(index=X.index, data=ap.fit_predict(X).flatten())
# clustered stock with its cluster label
clustered_series_all = pd.Series(index=X.index, data=ap.fit_predict(X).flatten())
clustered_series = clustered_series[clustered_series != -1]
# get the number of stocks in each cluster
counts = clustered_series_ap.value_counts()
# let's visualize some clusters
cluster_vis_list = list(counts[(counts<25) & (counts>1)].index)[::-1]
cluster_vis_list
# plot a handful of the smallest clusters
plt.figure(figsize=(12, 7))
cluster_vis_list[0:min(len(cluster_vis_list), 4)]

for clust in cluster_vis_list[0:min(len(cluster_vis_list), 4)]:
    tickers = list(clustered_series[clustered_series==clust].index)
    # calculate the return (lognormal) of the stocks
    means = np.log(dataset.loc[:, "2018-02-01", tickers].mean())
    data = np.log(dataset.loc[:, "2018-02-01", tickers]).sub(means)
    data.plot(title='Stock Time Series for Cluster %d' % clust)
plt.show()
```

Output





Looking at the charts above, across all the clusters with small number of stocks, we see similar movement of the stocks under different clusters, which corroborates the effectiveness of the clustering technique.

6. Pairs selection

Once the clusters are created, several cointegration-based statistical techniques can be applied on the stocks within a cluster to create the pairs. Two or more time series are considered to be cointegrated if they are nonstationary and tend to move together.² The presence of cointegration between time series can be validated through several statistical techniques, including the **Augmented Dickey-Fuller test** and the **Johansen test**.

In this step, we scan through a list of securities within a cluster and test for cointegration between the pairs. First, we write a function that returns a cointegration test score matrix, a p-value matrix, and any pairs for which the p-value was less than 0.05.

Cointegration and pair selection function.

```
def find_cointegrated_pairs(data, significance=0.05):
    # This function is from https://www.quantopian.com
    n = data.shape[1]
    score_matrix = np.zeros((n, n))
    pvalue_matrix = np.ones((n, n))
    keys = data.keys()
    pairs = []
    for i in range(1):
        for j in range(i+1, n):
            S1 = data[keys[i]]
            S2 = data[keys[j]]
            result = coint(S1, S2)
            score = result[0]
            pvalue = result[1]
            score_matrix[i, j] = score
```

² Refer to [Chapter 5](#) for more details.

```

    pvalue_matrix[i, j] = pvalue
    if pvalue < significance:
        pairs.append((keys[i], keys[j]))
return score_matrix, pvalue_matrix, pairs

```

Next, we check the cointegration of different pairs within several clusters using the function created above and return the pairs found:

```

from statsmodels.tsa.stattools import coint
cluster_dict = {}
for i, which_clust in enumerate(ticker_count_reduced.index):
    tickers = clustered_series[clustered_series == which_clust].index
    score_matrix, pvalue_matrix, pairs = find_cointegrated_pairs(
        dataset[tickers])
)
cluster_dict[which_clust] = {}
cluster_dict[which_clust]['score_matrix'] = score_matrix
cluster_dict[which_clust]['pvalue_matrix'] = pvalue_matrix
cluster_dict[which_clust]['pairs'] = pairs

pairs = []
for clust in cluster_dict.keys():
    pairs.extend(cluster_dict[clust]['pairs'])

print ("Number of pairs found : %d" % len(pairs))
print ("In those pairs, there are %d unique tickers." % len(np.unique(pairs)))

```

Output

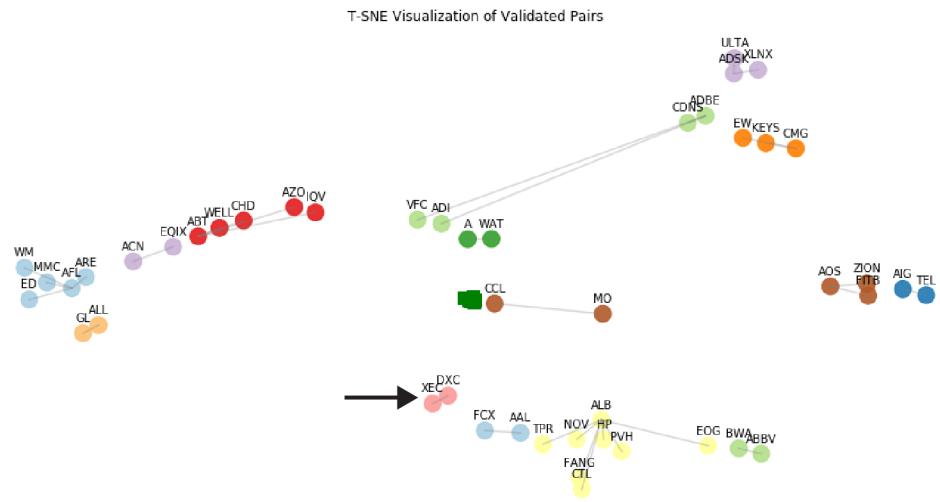
```

Number of pairs found : 32
In those pairs, there are 47 unique tickers.

```

Let us visualize the results of the pair selection process now. Refer to the Jupyter notebook of this case study for the details of the steps related to the pair visualization using the t-SNE technique.

The following chart shows the strength of k -means for finding nontraditional pairs (pointed out with an arrow in the visualization). DXC is the ticker symbol for DXC Technology, and XEC is the ticker symbol for Cimarex Energy. These two stocks are from different sectors and appear to have nothing in common on the surface, but they are identified as pairs using k -means clustering and cointegration testing. This implies that a long-run stable relationship exists between their stock price movements.



Once the pairs are created, they can be used in a pairs trading strategy. When the share prices of the pair deviate from the identified long-run relationship, an investor would seek to take a long position in the underperforming security and sell short the outperforming security. If the securities return to their historical relationship, a profit is made from the convergence of the prices.

Conclusion

In this case study, we demonstrated the efficiency of clustering techniques by finding small pools of stocks in which to identify pairs to be used in a pairs trading strategy. A next step beyond this case study would be to explore and backtest various long/short trading strategies with pairs of stocks from the groupings of stocks.

Clustering can be used for dividing stocks and other types of assets into groups with similar characteristics for several other kinds of trading strategies. It can also be effective in portfolio construction, helping to ensure we choose a pool of assets with sufficient diversification between them.

Case Study 2: Portfolio Management: Clustering Investors

Asset management and investment allocation is a tedious and time-consuming process in which investment managers often must design customized approaches for each client or investor.

What if we were able to organize these clients into particular investor profiles, or clusters, wherein each group is indicative of investors with similar characteristics?

Clustering investors based on similar characteristics can lead to simplicity and standardization in the investment management process. These algorithms can group investors based on different factors, such as age, income, and risk tolerance. It can help investment managers identify distinct groups within their investors base. Additionally, by using these techniques, managers can avoid introducing any biases that otherwise could adversely impact decision making. The factors analyzed through clustering can have a big impact on asset allocation and rebalancing, making it an invaluable tool for faster and effective investment management.

In this case study, we will use clustering methods to identify different types of investors.

The data used for this case study is from the Survey of Consumer Finances, which is conducted by the Federal Reserve Board. The same dataset was used in “[Case Study 3: Investor Risk Tolerance and Robo-Advisors](#)” on page 125 in Chapter 5.

In this case study, we focus on:

- Understanding the intuitive meaning of the groupings coming out of clustering.
- Choosing the right clustering techniques.
- Visualization of the clustering outcome and selecting the correct number of clusters in k -means.



Blueprint for Using Clustering for Grouping Investors

1. Problem definition

The goal of this case study is to build a clustering model to group individuals or investors based on parameters related to the ability and willingness to take risk. We will focus on using common demographic and financial characteristics to accomplish this.

The survey data we’re using includes responses from 10,000+ individuals in 2007 (precrisis) and 2009 (postcrisis). There are over 500 features. Since the data has many variables, we will first reduce the number of variables and select the most intuitive features directly linked to an investor’s ability and willingness to take risk.

2. Getting started—loading the data and Python packages

2.1. Loading the Python packages. The packages loaded for this case study are similar to those loaded in the case study presented in [Chapter 5](#). However, some additional packages related to the clustering techniques are shown in the following code snippet:

```
#Import packages for clustering techniques
from sklearn.cluster import KMeans, AgglomerativeClustering,AffinityPropagation
from sklearn.metrics import adjusted_mutual_info_score
from sklearn import cluster, covariance, manifold
```

2.2. Loading the data. The data (again, previously used in [Chapter 5](#)) is further processed to give the following attributes that represent an individual's ability and willingness to take risk. This preprocessed data is for the 2007 survey and is loaded below:

```
# load dataset
dataset = pd.read_excel('ProcessedData.xlsx')
```

3. Exploratory data analysis

Next, we take a closer look at the different columns and features found in the data.

3.1. Descriptive statistics. First, looking at the shape of the data:

```
dataset.shape
```

Output

```
(3866, 13)
```

The data has information for 3,886 individuals across 13 columns:

```
# peek at data
set_option('display.width', 100)
dataset.head(5)
```

ID	AGE	EDUC	MARRIED	KIDS	LIFECL	OCCAT	RISK	HHOUSES	WSAVED	SPENDMOR	NWCAT	INCCL	
0	1	3	2	1	0	2	1	3	1	1	5	3	4
1	2	4	4	1	2	5	2	3	0	2	5	5	5
2	3	3	1	1	2	3	2	2	1	2	4	4	4
3	4	3	1	1	2	3	2	2	1	2	4	3	4
4	5	4	3	1	1	5	1	2	1	3	3	5	5

As we can see in the table above, there are 12 attributes for each of the individuals. These attributes can be categorized as demographic, financial, and behavioral attributes. They are summarized in [Figure 8-2](#).

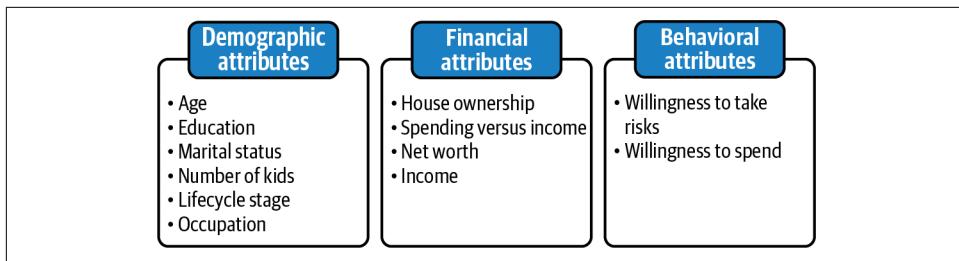


Figure 8-2. Attributes for clustering individuals

Many of these were previously used and defined in the [Chapter 5](#) case study. A few additional attributes (LIFECYCL, HHOUSES, and SPENDMOR) are used in this case study and are defined below:

LIFECYCL

This is a lifecycle variable, used to approximate a person's ability to take on risk. There are six categories in increasing level of ability to take risk. A value of 1 represents "age under 55, not married, and no kids," and a value of 6 represents "age over 55 and not working."

HHOUSES

This is a flag indicating whether the individual is a homeowner. A value of 1 (0) implies the individual does (does not) own a home.

SPENDMOR

This represents higher spending preference if assets appreciated on a scale of 1 to 5.

3.2. Data visualization. We will take a detailed look into the visualization postclustering.

4. Data preparation

Here, we perform any necessary changes to the data in preparation for modeling.

4.1. Data cleaning. In this step, we check for NAs in the rows and either drop them or fill them with the mean of the column:

```
print('Null Values =', dataset.isnull().values.any())
```

Output

```
Null Values = False
```

Given that there is not any missing data, and the data is already in categorical format, no further data cleaning was performed. The *ID* column is unnecessary and is dropped:

```
X=X.drop(['ID'], axis=1)
```

4.2. Data transformation. As we saw in Section 3.1, all the columns represent categorical data with similar numeric scale, with no outliers. Hence, no data transformation will be required for clustering.

5. Evaluate algorithms and models

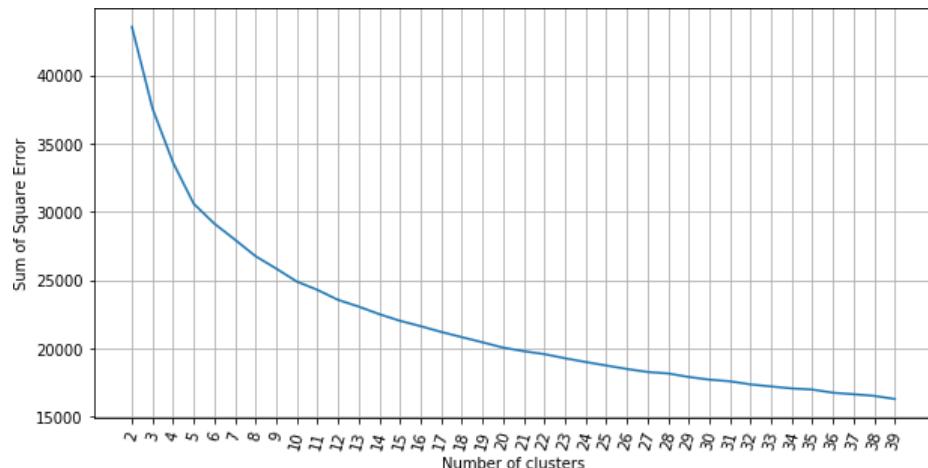
We will analyze the performance of k -means and affinity propagation.

5.1. k -means clustering. We look at the details of the k -means clustering in this step. First, we find the optimal number of clusters, followed by the creation of a model.

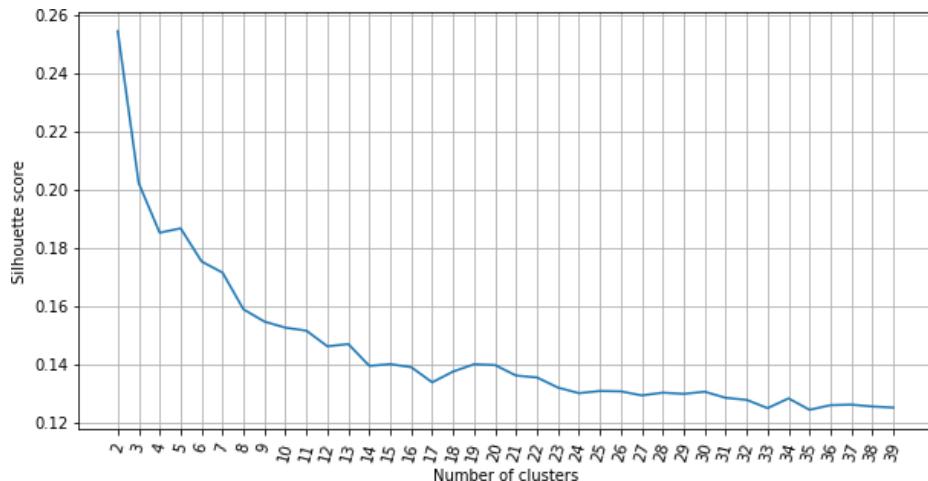
5.1.1. Finding the optimal number of clusters. We look at the following two metrics to evaluate the number of clusters in the k -means model. The Python code to get these two metrics is the same as in case study 1:

1. Sum of squared errors (SSE)
2. Silhouette score

Sum of squared errors (SSE) within clusters



Silhouette score



Looking at both of the preceding charts, the optimum number of clusters seems to be around 7. We can see that as the number of clusters increases past 6, the SSE within clusters begins to plateau. From the second graph, we can see that there are various parts of the graph where a kink can be seen. Since there is not much of a difference in the SSE after 7 clusters, we proceed with using 7 clusters in the k -means model below.

5.1.2. Clustering and visualization. Let us create a k -means model with 7 clusters:

```
nclust=7

#Fit with k-means
k_means = cluster.KMeans(n_clusters=nclust)
k_means.fit(X)
```

Let us assign a target cluster to each individual in the dataset. This assignment is used further for exploratory data analysis to understand the behavior of each cluster:

```
#Extracting labels
target_labels = k_means.predict(X)
```

5.2. Affinity propagation. Here, we build an affinity propagation model and look at the number of clusters:

```
ap = AffinityPropagation()
ap.fit(X)
clust_labels2 = ap.predict(X)

cluster_centers_indices = ap.cluster_centers_indices_
labels = ap.labels_
```

```
n_clusters_ = len(cluster_centers_indices)
print('Estimated number of clusters: %d' % n_clusters_)
```

Output

```
Estimated number of clusters: 161
```

The affinity propagation resulted in over 150 clusters. Such a large number will likely make it difficult to ascertain proper differentiation between them.

5.3. Cluster evaluation. In this step, we check the performance of the clusters using silhouette coefficient (`sklearn.metrics.silhouette_score`). Recall that a higher silhouette coefficient score relates to a model with better defined clusters:

```
from sklearn import metrics
print("km", metrics.silhouette_score(X, k_means.labels_))
print("ap", metrics.silhouette_score(X, ap.labels_))
```

Output

```
km 0.170585217843582
ap 0.09736878398868973
```

The k -means model has a much higher silhouette coefficient compared to the affinity propagation. Additionally, the large number of clusters resulting from the affinity propagation is untenable. In the context of the problem at hand, having fewer clusters, or categorizations of investors, helps build simplicity and standardization in the investment management process. It gives the users of this information (e.g., financial advisors) some manageable intuition around the representation of the clusters. Comprehending and being able to speak to six to eight investor types is much more practical than maintaining a meaningful understanding of over 100 different profiles. With this in mind, we proceed with k -means as the preferred clustering technique.

6. Cluster intuition

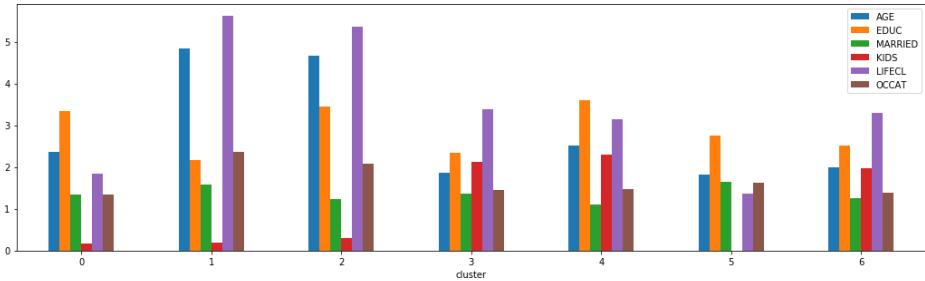
In the next step, we will analyze the clusters and attempt to draw conclusions from them. We do that by plotting the average of each variable of the cluster and summarizing the findings:

```
cluster_output= pd.concat([pd.DataFrame(X), pd.DataFrame(k_means.labels_, \
                                         columns = ['cluster'])],axis=1)
output=cluster_output.groupby('cluster').mean()
```

Demographics Features: Plot for each of the clusters

```
output[['AGE','EDUC','MARRIED','KIDS','LIFECL','OCCAT']].\
plot.bar(rot=0, figsize=(18,5));
```

Output

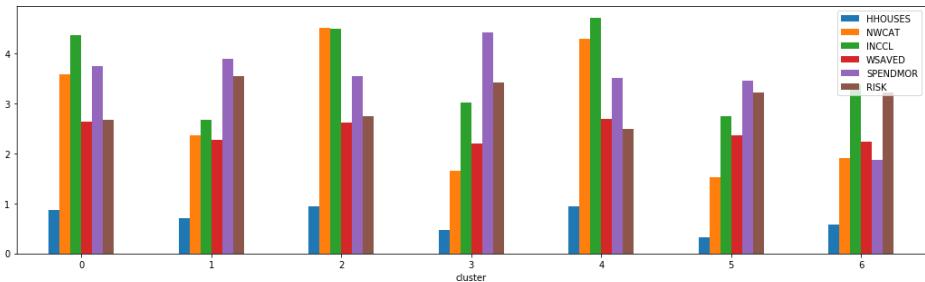


The plot here shows the average values of the attributes for each of the clusters (full size version available on [GitHub](#)). For example, in comparing clusters 0 and 1, cluster 0 has *lower* average age, yet *higher* average education. However, these two clusters are more similar in marital status and number of children. So, based on the demographic attributes, the individuals in cluster 0 will, on average, have higher risk tolerance compared to those in cluster 1.

Financial and Behavioral Attributes: Plot for each of the clusters

```
output[['HHOUSES','NWCAT','INCLL','WSAVED','SPENDMOR','RISK']].\nplot.bar(rot=0, figsize=(18,5));
```

Output



The plot here shows the average values of the financial and behavior attributes for each of the clusters (full size version available on [GitHub](#)). Again, comparing clusters 0 and 1, the former has higher average house ownership, higher average net worth and income, and a lower willingness to take risk compared to the latter. In terms of saving versus income comparison and willingness to save, the two clusters are comparable. Therefore, we can posit that the individuals in cluster 0 will, on average, have a higher ability and yet a lower willingness to take risks compared to the individuals in cluster 1.

Combining the information from the demographics, financial, and behavioral attributes for these two clusters, the overall ability to take risks for an individual in cluster 0 is higher than someone in cluster 1. Performing similar analyses across all other clusters, we summarize the results in the table below. The risk tolerance column represents the subjective assessment of the risk tolerance of each cluster.

Cluster	Features	Risk capacity
Cluster 0	Low age, high net worth and income, less risky life category, willingness to spend more	High
Cluster 1	High age, low net worth and income, highly risky life category, willingness to take risk, low education	Low
Cluster 2	High age, high net worth and income, highly risky life category, willingness to take risk, owns home	Medium
Cluster 3	Low age, very low income and net worth, high willingness to take risk, many kids	Low
Cluster 4	Medium age, very high income and net worth, high willingness to take risk, many kids, owns home	High
Cluster 5	Low age, very low income and net worth, high willingness to take risk, no kids	Medium
Cluster 6	Low age, medium income and net worth, high willingness to take risk, many kids, owns home	Low

Conclusion

One of the key takeaways from this case study is the approach to understanding the cluster intuition. We used visualization techniques to understand the expected behavior of a cluster member by qualitatively interpreting mean values of the variables in each cluster. We demonstrated the efficiency of clustering in discovering the natural groups of different investors based on their risk tolerance.

Given that clustering algorithms can successfully group investors based on different factors (such as age, income, and risk tolerance), they can be further used by portfolio managers to standardize portfolio allocation and rebalance strategies across the clusters, making the investment management process faster and more effective.

Case Study 3: Hierarchical Risk Parity

Markowitz's *mean-variance portfolio optimization* is the most commonly used technique for portfolio construction and asset allocation. In this technique, we need to estimate the covariance matrix and expected returns of assets to be used as inputs. As discussed in “[Case Study 1: Portfolio Management: Finding an Eigen Portfolio](#)” on [page 202](#) in [Chapter 7](#), the erratic nature of financial returns causes estimation errors in the expected returns and the covariance matrix, especially when the number of assets is large compared to the sample size. These errors greatly jeopardize the optimality of the resulting portfolios, which leads to erroneous and unstable results. Additionally, small changes in the assumed asset returns, volatilities, or covariances

can lead to large effects on the output of the optimization procedure. In this sense, the Markowitz mean-variance optimization is an ill-posed (or ill-conditioned) inverse problem.

In “[Building Diversified Portfolios That Outperform Out-of-Sample](#)” by Marcos López de Prado (2016), the author proposes a portfolio allocation method based on clustering called *hierarchical risk parity*. The main idea of hierarchical risk parity is to run hierarchical clustering on the covariance matrix of stock returns and then find a diversified weighting by distributing capital equally to each cluster hierarchy (so that many correlated strategies will receive the same total allocation as a single uncorrelated one). This alleviates some of the issues (highlighted above) found in Markowitz’s mean-variance optimization and improves numerical stability.

In this case study, we will implement hierarchical risk parity based on clustering methods and compare it against Markowitz’s mean-variance optimization method.

The dataset used for this case study is price data for stocks in the S&P 500 from 2018 onwards. The dataset can be downloaded from Yahoo Finance. It is the same dataset as was used in case study 1.

In this case study, we will focus on:

- Application of clustering-based techniques for portfolio allocation.
- Developing a framework for comparing portfolio allocation methods.



Blueprint for Using Clustering to Implement Hierarchical Risk Parity

1. Problem definition

Our goal in this case study is to use a clustering-based algorithm on a dataset of stocks to allocate capital into different asset classes. In order to backtest and compare the portfolio allocation against the traditional Markowitz mean-variance optimization, we will perform visualization and use performance metrics, such as the Sharpe ratio.

2. Getting started—loading the data and Python packages

2.1. Loading the Python packages. The packages loaded for this case study are similar to those loaded in the previous case study. However, some additional packages related to the clustering techniques are shown in the following code snippet:

```
#Import Model Packages
import scipy.cluster.hierarchy as sch
from sklearn.cluster import AgglomerativeClustering
from scipy.cluster.hierarchy import fcluster
from scipy.cluster.hierarchy import dendrogram, linkage, cophenet
from sklearn.metrics import adjusted_mutual_info_score
from sklearn import cluster, covariance, manifold
import ffn

#Package for optimization of mean variance optimization
import cvxopt as opt
from cvxopt import blas, solvers
```

Since this case study uses the same data as case study 1, some of the next steps (i.e., loading the data) have been skipped to avoid repetition. As a reminder, the data contains around 500 stocks and 448 observations.

3. Exploratory data analysis

We will take a detailed look into the visualization postclustering later in this case study.

4. Data preparation

4.1. Data cleaning. Refer to case study 1 for data cleaning steps.

4.2. Data transformation. We will be using annual returns for clustering. Additionally, we will train the data and then test the data. Here, we prepare the dataset for training and testing by separating 20% of the dataset for testing, and we generate the return series:

```
X= dataset.copy('deep')
row= len(X)
train_len = int(row*.8)

X_train = X.head(train_len)
X_test = X.tail(row-train_len)

#Calculate percentage return
returns = X_train.to_returns().dropna()
returns_test=X_test.to_returns().dropna()
```

5. Evaluate algorithms and models

In this step, we will look at hierarchical clustering and perform further analysis and visualization.

5.1. Building a hierarchy graph/dendrogram. The first step is to look for clusters of correlations using the agglomerative hierarchical clustering technique. The hierarchy class has a dendrogram method that takes the value returned by the linkage method of the same class. The linkage method takes the dataset and the method to minimize distances as parameters. There are different options for measurement of the distance. The option we will choose is ward, since it minimizes the variance of distances between the clusters. Other possible measures of distance include single and centroid.

Linkage does the actual clustering in one line of code and returns a list of the clusters joined in the format:

```
Z= [stock_1, stock_2, distance, sample_count]
```

As a precursor, we define a function to convert correlation into distances:

```
def correldist(corr):
    # A distance matrix based on correlation, where 0<=d[i,j]<=1
    # This is a proper distance metric
    dist = ((1 - corr) / 2.) ** .5 # distance matrix
    return dist
```

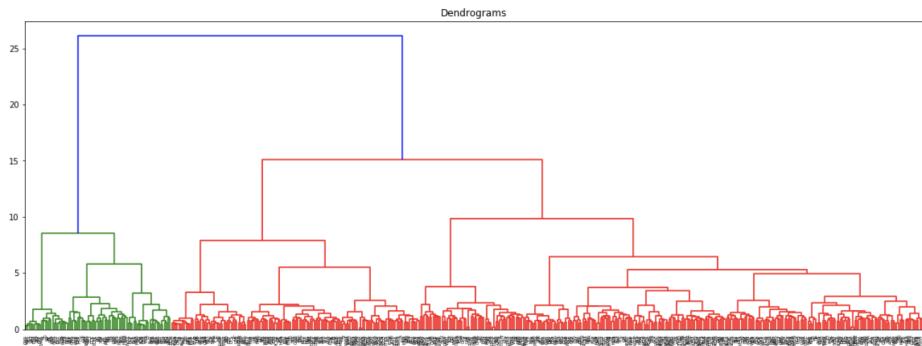
Now we convert the correlation of the returns of the stocks into distances, followed by the computation of linkages in the step below. Computation of linkages is followed by the visualization of the clusters through a dendrogram. Again, the leaves are the individual stocks, and the root is the final single cluster. The distance between each cluster is shown on the y-axis; the longer the branches are, the less correlated two clusters are.

```
#Calculate linkage
dist = correldist(returns.corr())
link = linkage(dist, 'ward')

#Plot Dendrogram
plt.figure(figsize=(20, 7))
plt.title("Dendograms")
dendrogram(link, labels = X.columns)
plt.show()
```

In the following chart, the horizontal axis represents the clusters. Although the names of the stocks on the horizontal axis are not very clear (not surprising, given that there are 500 stocks), we can see that they are grouped into several clusters. The appropriate number of clusters appears to be 2, 3, or 6, depending on the desired distance threshold level. Next, we will leverage the linkages computed from this step to compute the asset allocation based on hierarchical risk parity.

Output



5.2. Steps for hierarchical risk parity. The hierarchical risk parity (HRP) algorithm works in three stages, as outlined in Prado's paper:

Tree clustering

Grouping similar investments into clusters based on their correlation matrix. Having a hierarchical structure helps us improve stability issues of quadratic optimizers when inverting the covariance matrix.

Quasi-diagonalization

Reorganizing the covariance matrix so similar investments will be placed together. This matrix diagonalization allows us to distribute weights optimally following an inverse-variance allocation.

Recursive bisection

Distributing the allocation through recursive bisection based on cluster covariance.

Having performed the first stage in the previous section, where we identified clusters based on the distance metrics, we proceed to quasi-diagonalization.

5.2.1. Quasi-diagonalization. Quasi-diagonalization is a process known as *matrix seriation*, which reorganizes the rows and columns of a covariance matrix so that the largest values lie along the diagonal. As shown in the following code, the process reorganizes the covariance matrix so similar investments are placed together. This matrix diagonalization allows us to distribute weights optimally following an inverse-variance allocation:

```
def getQuasiDiag(link):
    # Sort clustered items by distance
    link = link.astype(int)
    sortIx = pd.Series([link[-1, 0], link[-1, 1]])
    numItems = link[-1, 3] # number of original items
```

```

while sortIx.max() >= numItems:
    sortIx.index = range(0, sortIx.shape[0] * 2, 2) # make space
    df0 = sortIx[sortIx >= numItems] # find clusters
    i = df0.index
    j = df0.values - numItems
    sortIx[i] = link[j, 0] # item 1
    df0 = pd.Series(link[j, 1], index=i + 1)
    sortIx = sortIx.append(df0) # item 2
    sortIx = sortIx.sort_index() # re-sort
    sortIx.index = range(sortIx.shape[0]) # re-index
return sortIx.tolist()

```

5.2.2. Recursive bisection. In the next step, we perform recursive bisection, which is a top-down approach to splitting portfolio weights between subsets based on the inverse proportion to their aggregated variances. The function `getClusterVar` computes the cluster variance, and in this process, it requires the inverse-variance portfolio from the function `getIVP`. The output of the function `getClusterVar` is used by the function `getRecBipart` to compute the final allocation through recursive bisection based on cluster covariance:

```

def getIVP(cov, **kargs):
    # Compute the inverse-variance portfolio
    ivp = 1. / np.diag(cov)
    ivp /= ivp.sum()
    return ivp

def getClusterVar(cov,cItems):
    # Compute variance per cluster
    cov_=cov.loc[cItems,cItems] # matrix slice
    w_=getIVP(cov_).reshape(-1, 1)
    cVar=np.dot(np.dot(w_.T,cov_),w_)[0, 0]
    return cVar

def getRecBipart(cov, sortIx):
    # Compute HRP alloc
    w = pd.Series(1, index=sortIx)
    cItems = [sortIx] # initialize all items in one cluster
    while len(cItems) > 0:
        cItems = [i[j:k] for i in cItems for j, k in ((0,
            len(i) // 2), (len(i) // 2, len(i))) if len(i) > 1] # bi-section
        for i in range(0, len(cItems), 2): # parse in pairs
            cItems0 = cItems[i] # cluster 1
            cItems1 = cItems[i + 1] # cluster 2
            cVar0 = getClusterVar(cov, cItems0)
            cVar1 = getClusterVar(cov, cItems1)
            alpha = 1 - cVar0 / (cVar0 + cVar1)
            w[cItems0] *= alpha # weight 1
            w[cItems1] *= 1 - alpha # weight 2
    return w

```

The following function `getHRP` combines the three stages—clustering, quasi-diagonalization, and recursive bisection—to produce the final weights:

```
def getHRP(cov, corr):
    # Construct a hierarchical portfolio
    dist = correlDist(corr)
    link = sch.linkage(dist, 'single')
    #plt.figure(figsize=(20, 10))
    #dn = sch.dendrogram(link, labels=cov.index.values)
    #plt.show()
    sortIx = getQuasiDiag(link)
    sortIx = corr.index[sortIx].tolist()
    hrp = getRecBipart(cov, sortIx)
    return hrp.sort_index()
```

5.3. Comparison against other asset allocation methods. A main focus of this case study is to develop an alternative to Markowitz's mean-variance portfolio optimization using clustering. In this step, we define a function to compute the allocation of a portfolio based on Markowitz's mean-variance technique. This function (`getMVP`) takes the covariance matrix of the assets as an input, performs the mean-variance optimization, and produces the portfolio allocations:

```
def getMVP(cov):
    cov = cov.T.values
    n = len(cov)
    N = 100
    mus = [10 ** (5.0 * t / N - 1.0) for t in range(N)]

    # Convert to cvxopt matrices
    S = opt.matrix(cov)
    #pbar = opt.matrix(np.mean(returns, axis=1))
    pbar = opt.matrix(np.ones(cov.shape[0]))

    # Create constraint matrices
    G = -opt.matrix(np.eye(n)) # negative n x n identity matrix
    h = opt.matrix(0.0, (n, 1))
    A = opt.matrix(1.0, (1, n))
    b = opt.matrix(1.0)

    # Calculate efficient frontier weights using quadratic programming
    solvers.options['show_progress'] = False
    portfolios = [solvers.qp(mu * S, -pbar, G, h, A, b)['x']
                 for mu in mus]
    ## Calculate risk and return of the frontier
    returns = [blas.dot(pbar, x) for x in portfolios]
    risks = [np.sqrt(bla.dot(x, S * x)) for x in portfolios]
    ## Calculate the 2nd degree polynomial of the frontier curve.
    m1 = np.polyfit(returns, risks, 2)
    x1 = np.sqrt(m1[2] / m1[0])
    # CALCULATE THE OPTIMAL PORTFOLIO
    wt = solvers.qp(opt.matrix(x1 * S), -pbar, G, h, A, b)['x']
```

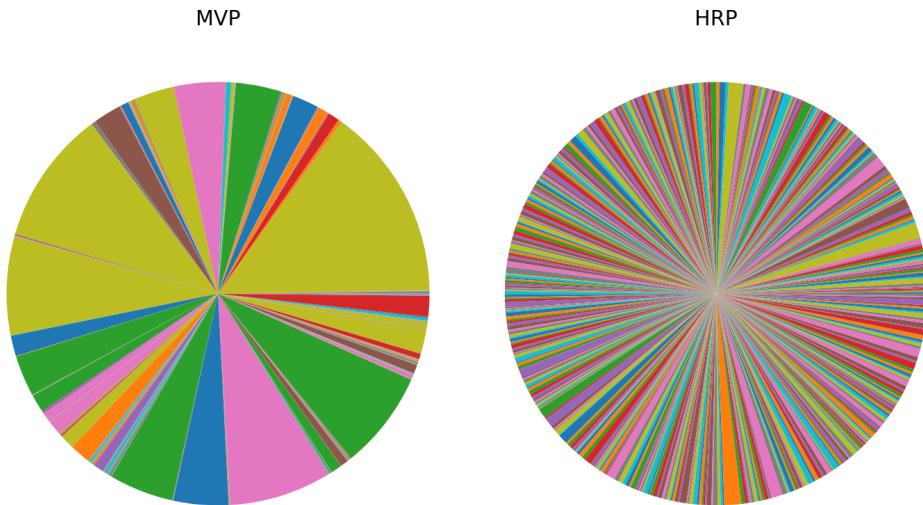
```
return list(wt)
```

5.4. Getting the portfolio weights for all types of asset allocation. In this step, we use the functions above to compute the asset allocation using the two asset allocation methods. We then visualize the asset allocation results:

```
def get_all_portfolios(returns):  
  
    cov, corr = returns.cov(), returns.corr()  
    hrp = getHRP(cov, corr)  
    mvp = getMVP(cov)  
    mvp = pd.Series(mvp, index=cov.index)  
    portfolios = pd.DataFrame([mvp, hrp], index=['MVP', 'HRP']).T  
    return portfolios  
  
#Now getting the portfolios and plotting the pie chart  
portfolios = get_all_portfolios(returns)  
  
portfolios.plot.pie(subplots=True, figsize=(20, 10), legend = False);  
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(30,20))  
ax1.pie(portfolios.iloc[:, 0], );  
ax1.set_title('MVP', fontsize=30)  
ax2.pie(portfolios.iloc[:, 1]);  
ax2.set_title('HRP', fontsize=30)
```

The following pie charts show the asset allocation of MVP versus HRP. We clearly see more diversification in HRP. Now let us look at the backtesting results.

Output



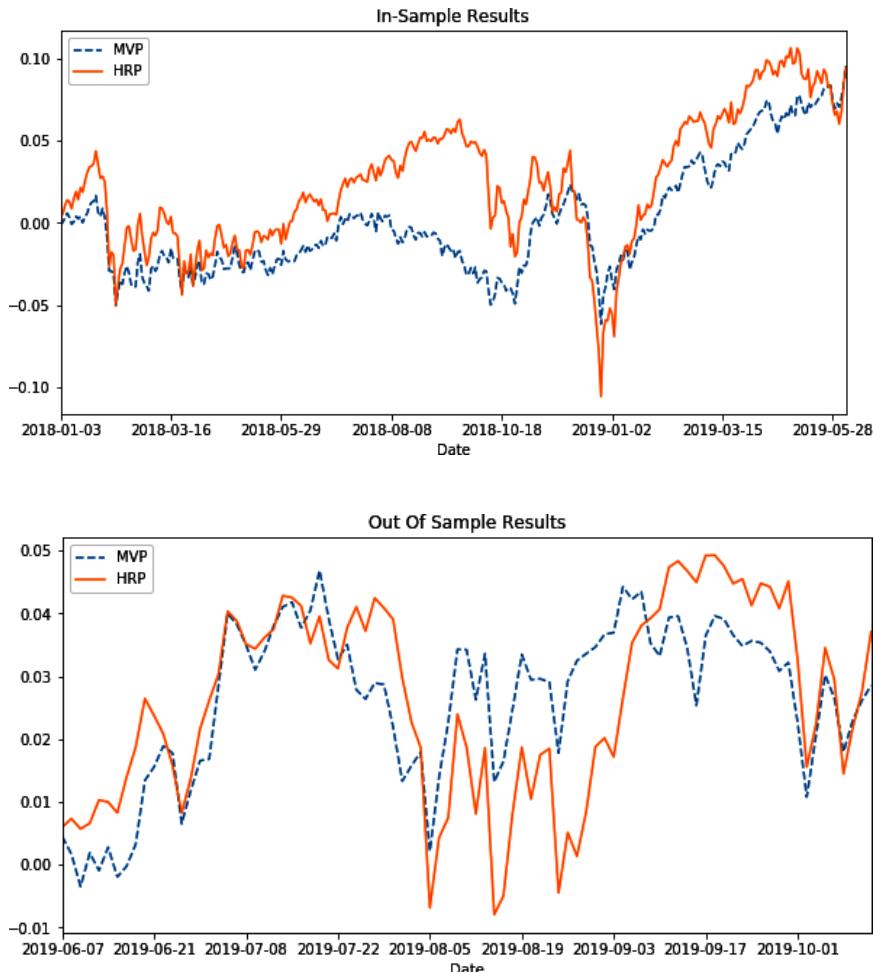
6. Backtesting

We will now backtest the performance of portfolios produced by the algorithms, looking at both in-sample and out-of-sample results:

```
Insample_Result=pd.DataFrame(np.dot(returns,np.array(portfolios)), \
'MVP','HRP'], index = returns.index)
OutOfSample_Result=pd.DataFrame(np.dot(returns_test,np.array(portfolios)), \
columns=['MVP', 'HRP'], index = returns_test.index)

Insample_Result.cumsum().plot(figsize=(10, 5), title ="In-Sample Results",\
style=['-', '-'])
OutOfSample_Result.cumsum().plot(figsize=(10, 5), title ="Out Of Sample Results",\
style=['-', '-'])
```

Output



Looking at the charts, MVP underperforms for a significant amount of time in the in-sample test. In the out-of-sample test, MVP performed better than HRP for a brief period of time from August 2019 to mid-September 2019. In the next step, we examine the Sharpe ratio for the two allocation methods:

In-sample results.

```
#In_sample Results
stddev = Insample_Result.std() * np.sqrt(252)
sharp_ratio = (Insample_Result.mean()*np.sqrt(252))/(Insample_Result).std()
Results = pd.DataFrame(dict(stddev=stddev, sharp_ratio = sharp_ratio))
Results
```

Output

	stdev	sharp_ratio
MVP	0.086	0.785
HRP	0.127	0.524

Out-of-sample results.

```
#OutOf_sample Results
stddev_oos = OutOfSample_Result.std() * np.sqrt(252)
sharp_ratio_oos = (OutOfSample_Result.mean()*np.sqrt(252))/(OutOfSample_Result).\\
std()
Results_oos = pd.DataFrame(dict(stdev_oos=stddev_oos, sharp_ratio_oos = \
sharp_ratio_oos))
Results_oos
```

Output

	stdev_oos	sharp_ratio_oos
MVP	0.103	0.787
HRP	0.126	0.836

Although the in-sample results of MVP look promising, the out-of-sample Sharpe ratio and overall return of the portfolio constructed using the hierarchical clustering approach are better. The diversification that HRP achieves across uncorrelated assets makes the methodology more robust against shocks.

Conclusion

In this case study, we saw that portfolio allocation based on hierarchical clustering offers better separation of assets into clusters with similar characteristics without relying on classical correlation analysis used in Markowitz's mean-variance portfolio optimization.

Using Markowitz's technique yields a less diverse portfolio, concentrated in a few stocks. The HRP approach, leveraging hierarchical clustering-based allocation, results in a more diverse and distributed portfolio. This approach presented the best out-of-sample performance and offers better tail risk management due to the diversification.

Indeed, the corresponding hierarchical risk parity strategies address the shortcomings of minimum-variance-based portfolio allocation. It is visual and flexible, and it seems to offer a robust methodology for portfolio allocation and portfolio management.

Chapter Summary

In this chapter, we learned about different clustering techniques and used them to capture the natural structure of data to enhance decision making across several areas of finance. Through the case studies, we demonstrated that clustering techniques can be useful in enhancing trading strategies and portfolio management.

In addition to offering an approach to different finance problems, the case studies focused on understanding the concepts of clustering models, developing intuition, and visualizing clusters. Overall, the concepts in Python, machine learning, and finance presented in this chapter through the case studies can be used as a blueprint for any other clustering-based problem in finance.

Having covered supervised and unsupervised learning, we will explore another type of machine learning, reinforcement learning, in the next chapter.

Exercises

- Use hierarchical clustering to form clusters of investments in a different asset class, such as forex or commodities.
- Apply clustering analysis for pairs trading in the interest rate market on the universe of bonds.

PART IV

Reinforcement Learning and Natural Language Processing

Reinforcement Learning

Incentives drive nearly everything, and finance is not an exception. Humans do not learn from millions of labeled examples. Instead, we often learn from positive or negative experiences that we associate with our actions. Learning from experiences and the associated rewards or punishments is the core idea behind reinforcement learning (RL).¹

Reinforcement learning is an approach toward training a machine to find the best course of action through optimal policies that maximize rewards and minimize punishments.

The RL algorithms that empowered *AlphaGo* (the first computer program to defeat a professional human Go player) are also finding inroads into finance. Reinforcement learning's main idea of *maximizing the rewards* aligns beautifully with several areas in finance, including algorithmic trading and portfolio management. Reinforcement learning is particularly suitable for algorithmic trading, because the concept of a *return-maximizing agent* in an uncertain, dynamic environment has much in common with an investor or a trading strategy that interacts with financial markets. Reinforcement learning-based models go one step further than the price prediction-based trading strategies discussed in previous chapters and determine rule-based policies for actions (i.e., place an order, do nothing, cancel an order, and so on).

Similarly, in portfolio management and asset allocation, reinforcement learning-based algorithms do not yield predictions and do not learn the structure of the market implicitly. They do more. They directly learn the policy of changing the portfolio allocation weights dynamically in the continuously changing market. Reinforcement learning models are also useful for order execution problems, which involve the

¹ Reinforcement learning is also referred to as RL throughout this chapter.

process of completing a buy or sell order for a market instrument. Here, the algorithms learn through trial and error, figuring out the optimal path of execution on their own.

Reinforcement learning algorithms, with their ability to tackle more nuances and parameters within the operational environment, can also produce derivatives hedging strategies. Unlike traditional finance-based hedging strategies, these hedging strategies are optimal and valid under real-world market frictions, such as transaction costs, market impact, liquidity constraints, and risk limits.

In this chapter, we cover three reinforcement learning–based case studies covering major finance applications: algorithmic trading, derivatives hedging, and portfolio allocation. In terms of the model development steps, the case studies follow a standardized seven-step model development process presented in [Chapter 2](#). Model development and evaluation are key steps for reinforcement learning, and these steps will be emphasized. With multiple concepts in machine learning and finance implemented, these case studies can be used as a blueprint for any other reinforcement learning–based problem in finance.

In “[Case Study 1: Reinforcement Learning–Based Trading Strategy](#)” on page 298, we demonstrate the use of RL to develop an algorithmic trading strategy.

In “[Case Study 2: Derivatives Hedging](#)” on page 316, we implement and analyze reinforcement learning–based techniques to calculate the optimal hedging strategies for portfolios of derivatives under market frictions.

In “[Case Study 3: Portfolio Allocation](#)” on page 334, we illustrate the use of a reinforcement learning–based technique on a dataset of cryptocurrency in order to allocate capital into different cryptocurrencies to maximize risk-adjusted returns. We also introduce a reinforcement learning–based *simulation environment* to train and test the model.

In addition to the points mentioned above, readers will understand the following points by the end of this chapter:

- Key components of reinforcement learning (i.e., reward, agent, environment, action, and policy).
- Model-based and model-free algorithms for reinforcement learning along with policy and value-based models.
- Fundamental approaches to solving reinforcement learning problems, such as Markov decision processes (MDP), temporal difference (TD) learning, and artificial neural networks (ANNs).
- Methods to train and test value-based and policy-based reinforcement learning algorithms using artificial neural networks and deep learning.

- How to set up an agent or simulation environment for reinforcement learning problems using Python.
- How to design and implement a problem statement related to algorithmic trading strategy, portfolio management, and instrument hedging in a classification-based machine learning framework.



This Chapter's Code Repository

A Python-based Jupyter notebook for all the case studies presented in this chapter is included under the folder [Chapter 9 - Reinforcement Learning](#) in the code repository for this book. To work through any machine learning problems in Python involving RL models (such as DQN or policy gradient) presented in this chapter, readers need to modify the template slightly to align with their problem statement.

Reinforcement Learning—Theory and Concepts

Reinforcement learning is an extensive topic covering a wide range of concepts and terminology. The theory section of this chapter covers the items and topics listed in [Figure 9-1](#).²

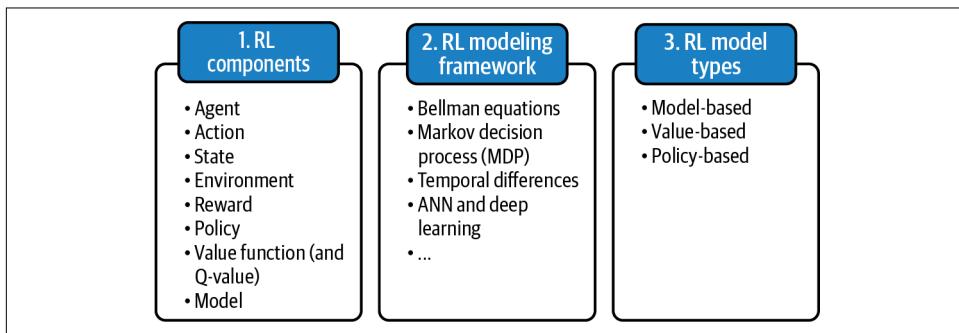


Figure 9-1. RL summary of concepts

In order to solve any problem using RL, it is important to first understand and define the RL components.

² For more details, be sure to check out *Reinforcement Learning: An Introduction* by Richard Sutton and Andrew Barto (MIT Press), or David Silver's free online RL course at [University College London](#).

RL Components

The main components of an RL system are agent, actions, environment, state, and reward.

Agent

The entity that performs actions.

Actions

The things an agent can do within its environment.

Environment

The world in which the agent resides.

State

The current situation.

Reward

The immediate return sent by the environment to evaluate the last action by the agent.

The goal of reinforcement learning is to learn an optimal strategy through experimental trials and relatively simple feedback loops. With the optimal strategy, the agent is capable of actively adapting to the environment to maximize the rewards. Unlike in supervised learning, these reward signals are not given to the model immediately. Instead, they are returned as a consequence of a sequence of actions that the agent makes.

An agent's actions are usually conditioned on what the agent perceives from the environment. What the agent perceives is referred to as the observation or the state of the environment. [Figure 9-2](#) summarizes the components of a reinforcement learning system.

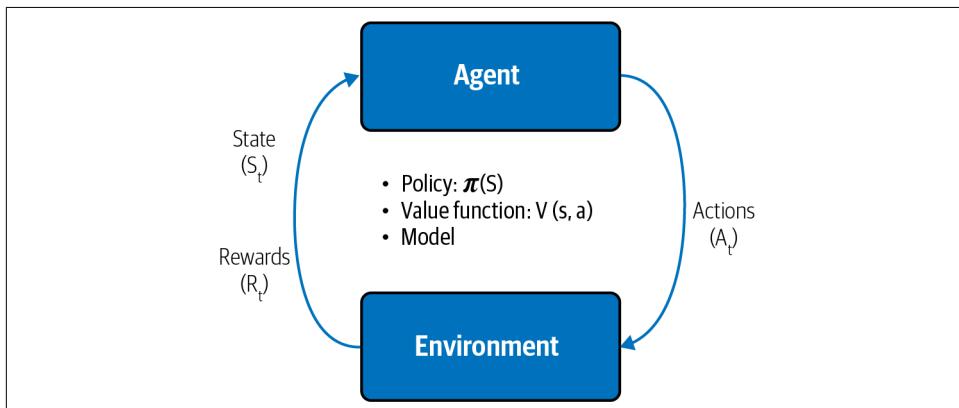


Figure 9-2. RL components

The interaction between the agent and the environment involves a sequence of actions and observed rewards in time, $t = 1, 2...T$. During the process, the agent accumulates knowledge about the environment, learns the optimal policy, and makes decisions on which action to take next so as to efficiently learn the best policy. Let's label the state, action, and reward at time step t as $S_t, A_t...R_t$, respectively. Thus, the interaction sequence is fully described by one episode (also known as "trial" or "trajectory"), and the sequence ends at the terminal state $S_T : S_1, A_1, R_2, S_2, A_2...A_T$.

In addition to the five components of reinforcement learning mentioned so far, there are three additional components of reinforcement learning: policy, value function (and Q-value), and model of the environment. Let us discuss the components in detail.

Policy

A policy is an algorithm or a set of rules that describes how an agent makes its decisions. More formally, a policy is a function, usually denoted as π , that maps a state (s) and an action (a):

$$a_t = \pi(s_t)$$

This means that an agent decides its action given its current state. The policy can be can be either deterministic or stochastic. A deterministic policy maps a state to actions. On the other hand, a stochastic policy outputs a probability distribution over actions. It means that instead of being sure of taking action a , there is a probability assigned to the action given a state.

Our goal in reinforcement learning is to learn an optimal policy (which is also referred to as π^*). An optimal policy tells us how to act to maximize return in every state.

Value function (and Q-value)

The goal of a reinforcement learning agent is to learn to perform a task well in an environment. Mathematically, this means maximizing the future reward, or cumulative discounted reward, G , which can be expressed in the following equation as a function of reward function R at different times:

$$G_t = R_{t+1} + \gamma R_{t+2} + \dots = \sum_0^{\infty} \gamma^k R_{t+k+1}$$

The discounting factor γ is a value between 0 and 1 to penalize the rewards in the future, as future rewards do not provide immediate benefits and may have higher uncertainty. Future reward is an important input to the value function.

The value function (or state value) measures the attractiveness of a state through a prediction of future reward G_t . The value function of a state s is the expected return, with a policy π if we are in this state at time t :

$$V(s) = E[G_t \mid S_t = s]$$

Similarly, we define the action-value function (Q-value) of a state-action pair (s, a) as:

$$Q(s, a) = E[G_t \mid S_t = s, A_t = a]$$

So the value function is the expected return for a state following a policy π . The Q-value is the expected reward for the state-action pair following a policy π .

The value function and the Q-value are interconnected as well. Since we follow the target policy π , we can make use of the probability distribution over possible actions and the Q-values to recover the value function:

$$V(s) = \sum_{a \in A} Q(s, a)\pi(a \mid s)$$

The preceding equation represents the relationship between the value function and Q-value.

The relationship between reward function (R), future rewards (G), value function, and Q-value is used to derive the Bellman equations (discussed later in this chapter), which are one of the key components of many reinforcement learning models.

Model

The model is a descriptor of the environment. With the model, we can learn or infer how the environment would interact with and provide feedback to the agent. Models are used for *planning*, by which we mean any way of deciding on a course of action by considering possible future situations. A model of the stock market, for example, is tasked with predicting what the prices will look like in the future. The model has two major parts: *transition probability function (P)* and *reward function*. We already discussed the reward function. The transition function (P) records the probability of transitioning from one state to another after taking an action.

Overall, an RL agent may be directly or indirectly trying to learn a policy or value function shown in [Figure 9-3](#). The approach to learning a policy varies depending on the RL model type. When we fully know the environment, we can find the optimal

solution by using *model-based approaches*.³ When we do not know the environment, we follow a *model-free approach* and try to learn the model explicitly as part of the algorithm.

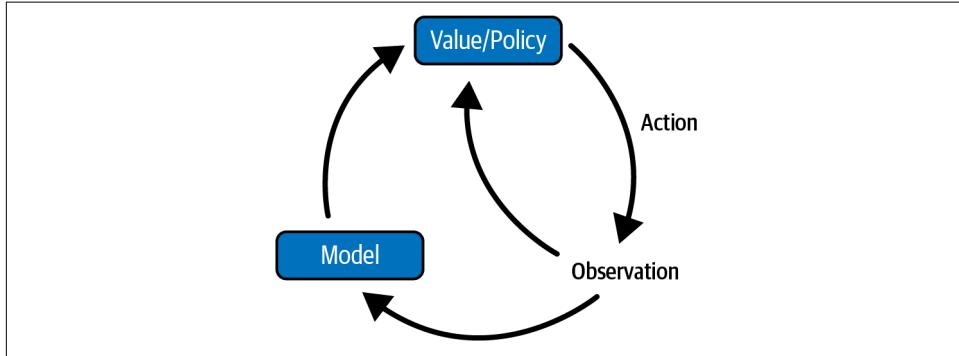


Figure 9-3. Model, value, and policy

RL components in a trading context

Let's try to understand what the RL components correspond to in a trading setting:

Agent

The agent is our trading agent. We can think of the agent as a human trader who makes trading decisions based on the current state of the exchange and their account.

Action

There would be three actions: *Buy*, *Hold*, and *Sell*.

Reward function

An obvious reward function would be the *realized PnL (Profit and Loss)*. Other reward functions can be *Sharpe ratio* or *maximum drawdown*.⁴ There can be a wide range of complex reward functions that offer a trade-off between profit and risk.

Environment

The environment in a trading context would be the *exchange*. In the case of trading on an exchange, we do not observe the complete state of the environment. Specifically, we are unaware of the other agents, and what an agent observes is not the true state of the environment but some derivation of it.

³ See “Reinforcement Learning Models” on page 293 for more details on model-based and model-free approaches.

⁴ A maximum drawdown is the maximum observed loss from peak to trough of a portfolio before a new peak is attained; it is an indicator of downside risk over a specified time period.

This is referred to as a *partially observable Markov decision process* (POMDP). This is the most common type of environment that we encounter in finance.

RL Modeling Framework

In this section, we describe the core framework of reinforcement learning used across several RL models.

Bellman equations

Bellman equations refer to a set of equations that decompose the value function and Q-value into the immediate reward plus the discounted future values.

In RL, the main aim of an agent is to get the most expected sum of rewards from every state it lands in. To achieve that, we must try to get the optimal value function and Q-value; the Bellman equations help us to do so.

We use the relationship between reward function (R), future rewards (G), value function, and Q-value to derive the Bellman equation for value function, as shown in [Equation 9-1](#).

Equation 9-1. Bellman equation for value function

$$V(s) = E[R_{t+1} + \gamma V(S_{t+1}) \mid S_t = s]$$

Here, the value function is decomposed into two parts; an immediate reward, R_{t+1} , and the discounted value of the successor state, $\gamma V(S_{t+1})$, as shown in the preceding equation. Hence, we have broken down the problem into the immediate reward and the discounted successor state. The state value $V(s)$ for the state s at time t can be computed using the current reward R_{t+1} and the value function at the time $t+1$. This is the Bellman equation for value function. This equation can be maximized to get an equation called Bellman Optimality Equation for value function, represented by $V^*(s)$.

We follow a very similar algorithm to estimate the optimal state-action values (Q-values). The simplified iteration algorithms for value function and Q-value are shown in [Equations 9-2](#) and [9-3](#).

Equation 9-2. Iteration algorithm for value function

$$V_{k+1}(s) = \max_a \sum_{s'} P_{ss'}^a (R_{ss'}^a + \gamma V_k(s'))$$

Equation 9-3. Iteration algorithm for Q-value

$$Q_{k+1}(s, a) = \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma * \max_{a'} Q_k(s', a')]$$

where

- $P_{ss'}^a$ is the transition probability from state s to state s' , given that action a was chosen.
- $R_{ss'}^a$ is the reward that the agent gets when it goes from state s to state s' , given that action a was chosen.

Bellman equations are important because they let us express values of states as values of other states. This means that if we know the value function or Q-value of s_{t+1} , we can very easily calculate the value of s_t . This opens a lot of doors for iterative approaches for calculating the value for each state, since if we know the value of the next state, we can know the value of the current state.

If we have complete information about the environment, the iteration algorithms shown in Equations 9-2 and 9-3 turn into a planning problem, solvable by dynamic programming that we will demonstrate in the next section. Unfortunately, in most scenarios, we do not know $R_{ss'}$ or $P_{ss'}$, and thus cannot apply the Bellman equations directly, but they lay the theoretical foundation for many RL algorithms.

Markov decision processes

Almost all RL problems can be framed as Markov decision processes (MDPs). MDPs formally describe an environment for reinforcement learning. A Markov decision process consists of five elements: $M = S, A, P, R, \gamma$, where the symbols carry the same meanings as defined in the previous section:

- S : a set of states
- A : a set of actions
- P : transition probability
- R : reward function
- γ : discounting factor for future rewards

MDPs frame the agent–environment interaction as a sequential decision problem over a series of time steps $t = 1, \dots, T$. The agent and the environment interact continually, the agent selecting actions and the environment responding to these actions and presenting new situations to the agent, with the aim of coming up with an optimal policy or strategy. Bellman equations form the basis for the overall algorithm.

All states in MDP have the Markov property, referring to the fact that the future depends only on the current state, not on the history.

Let us look into an example of MDP in a financial context and analyze the Bellman equation. Trading in the market can be formalized as an MDP, which is a process

that has specified transition probabilities from state to state. **Figure 9-4** shows an example of MDP in the financial market, with a set of states, transition probability, action, and reward.

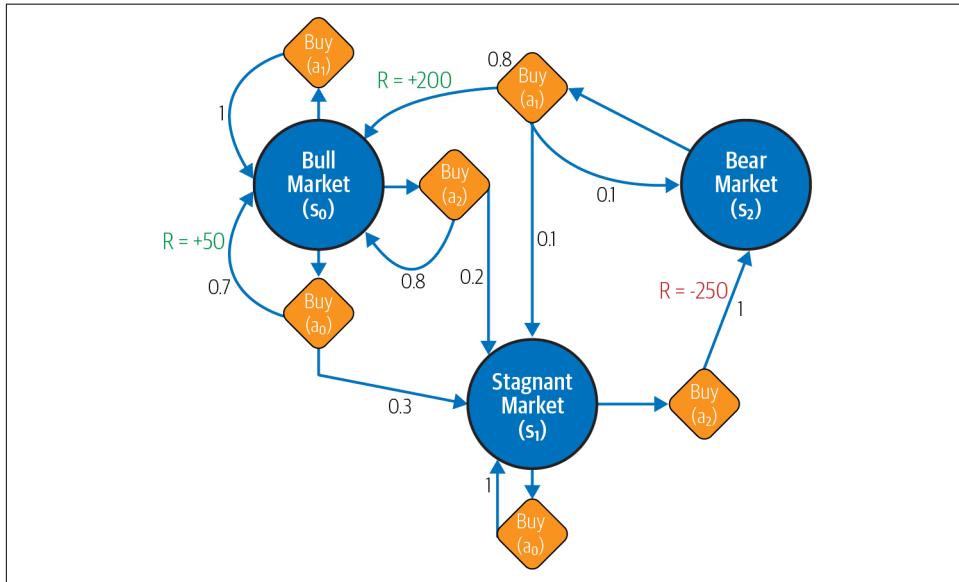


Figure 9-4. Markov decision process

The MDP presented here has three states: bull, bear, and stagnant market, represented by three states (s_0 , s_1 , s_2). The three actions of a trader are hold, buy, and sell, represented by a_0 , a_1 , a_2 , respectively. This is a hypothetical setup in which we assume that transition probabilities are known and the action of the trader leads to a change in the state of the market. In the subsequent sections we will look at approaches for solving RL problems without making such assumptions. The chart also shows the transition probabilities and the rewards for different actions. If we start in state s_0 (bull market), the agent can choose between actions a_0 , a_1 , a_2 (sell, buy, or hold). If it chooses action buy (a_1), it remains in state s_0 with certainty, and without any reward. It can thus decide to stay there forever if it wants. But if it chooses action hold (a_0), it has a 70% probability of gaining a reward of +50, and remaining in state s_0 . It can then try again to gain as much reward as possible. But at some point, it is going to end up instead in state s_1 (stagnant market). In state s_1 it has only two possible actions: hold (a_0) or buy (a_1). It can choose to stay put by repeatedly choosing action a_1 , or it can choose to move on to state s_2 (bear market) and get a negative reward of -250. In state s_3 it has no other choice than to take action buy (a_1), which will most likely lead it back to state s_0 (bull market), gaining a reward of +200 on the way.

Now, by looking at this MDP, it is possible to come up with an optimal policy or a strategy to achieve the most reward over time. In state s_0 it is clear that action a_0 is the

best option, and in state s_2 the agent has no choice but to take action a_1 , but in state s_1 it is not obvious whether the agent should stay put (a_0) or sell (a_2).

Let's apply the following Bellman equation as per [Equation 9-3](#) to get the optimal Q-value:

$$Q_{k+1}(s, a) = \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma * \max_{a'} Q_k(s', a')]$$

```

import numpy as np
nan=np.nan # represents impossible actions
#Array for transition probability
P = np.array([ # shape=[s, a, s']
[[0.7, 0.3, 0.0], [1.0, 0.0, 0.0], [0.8, 0.2, 0.0]],
[[0.0, 1.0, 0.0], [nan, nan, nan], [0.0, 0.0, 1.0]],
[[nan, nan, nan], [0.8, 0.1, 0.1], [nan, nan, nan]],
])

# Array for the return
R = np.array([ # shape=[s, a, s']
[[50., 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]],
[[50., 0.0, 0.0], [nan, nan, nan], [0.0, 0.0, -250.]],
[[nan, nan, nan], [200., 0.0, 0.0], [nan, nan, nan]],
])
#Actions
A = [[0, 1, 2], [0, 2], [1]]
#The data already obtained from yahoo finance is imported.

#Now let's run the Q-Value Iteration algorithm:
Q = np.full((3, 3), -np.inf) # -inf for impossible actions
for state, actions in enumerate(A):
    Q[state, actions] = 0.0 # Initial value = 0.0, for all possible actions
discount_rate = 0.95
n_iterations = 100
for iteration in range(n_iterations):
    Q_prev = Q.copy()
    for s in range(3):
        for a in A[s]:
            Q[s, a] = np.sum([
                T[s, a, sp] * (R[s, a, sp] + discount_rate * np.max(Q_prev[sp]))
            for sp in range(3)])
print(Q)

```

Output

```

[[109.43230584 103.95749333 84.274035  ]
 [ 5.5402017         -inf   5.83515676]
 [-inf 269.30353051      -inf]]
```

This gives us the optimal policy (Q-value) for this MDP, when using a discount rate of 0.95. Looking for the highest Q-value for each of the states: in a bull market (s_0)

choose action hold (a_0); in a stagnant market (s_1) choose action sell (a_2); and in a bear market (s_2) choose action buy (a_1).

The preceding example is a demonstration of a dynamic programming (DP) algorithm for obtaining optimal policy. These methods make an unrealistic assumption of complete knowledge of the environment but are the conceptual foundations for most other approaches.

Temporal difference learning

Reinforcement learning problems with discrete actions can often be modeled as Markov decision processes, as we saw in the previous example, but in most cases the agent initially has no insight into the transition probabilities. It also does not know what the rewards are going to be. This is where temporal difference (TD) learning can be useful.

A TD learning algorithm is very similar to the value iteration algorithm ([Equation 9-2](#)) based on the Bellman equation but is tweaked to take into account the fact that the agent has only partial knowledge of the MDP. In general, we assume that the agent initially knows only the possible states and actions and nothing more. For example, the agent uses an exploration policy, a purely random policy, to explore the MDP, and as it progresses, the TD learning algorithm updates the estimates of the state values based on the transitions and rewards that are actually observed.

The key idea in TD learning is to update the value function $V(S_t)$ toward an estimated return $R_{t+1} + \gamma V(S_{t+1})$ (known as the *TD target*). The extent to which we want to update the value function is controlled by the *learning rate* hyperparameter α , which defines how aggressive we want to be when updating our value. When α is close to zero, we're not updating very aggressively. When α is close to one, we're simply replacing the old value with the updated value:

$$V(s_t) \leftarrow V(s_t) + \alpha(R_{t+1} + \gamma V(s_{t+1}) - V(s_t))$$

Similarly, for Q-value estimation:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(R_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$$

Many RL models use the TD learning algorithm that we will see in the next section.

Artificial neural network and deep learning

Reinforcement learning models often leverage an artificial neural network and deep learning methods to approximate a value or policy function. That is, ANN can learn to map states to values, or state-action pairs to Q-values. ANNs use *coefficients*, or *weights*, to approximate the function relating inputs to outputs. In the context of RL,

the learning of ANNs means finding the right weights by iteratively adjusting them in such a way that the rewards are maximized. Refer to 3 and 5 for more details on methods related to ANN (including deep learning).

Reinforcement Learning Models

Reinforcement learning can be categorized into *model-based* and *model-free* algorithms, based on whether the rewards and probabilities for each step are readily accessible.

Model-based algorithms

Model-based algorithms try to understand the environment and create a model to represent it. When the RL problem includes well-defined transition probabilities and a limited number of states and actions, it can be framed as a *finite MDP* for which dynamic programming (DP) can compute an exact solution, similar to the previous example.⁵

Model-free algorithms

Model-free algorithms try to maximize the expected reward only from real experience, without a model or prior knowledge. Model-free algorithms are used when we have incomplete information about the model. The agent's policy $\pi(s)$ provides the guideline on what is the optimal action to take in a certain state with the goal of maximizing the total rewards. Each state is associated with a value function $V(s)$ predicting the expected amount of future rewards we are able to receive in this state by acting on the corresponding policy. In other words, the value function quantifies how good a state is. Model-free algorithms are further divided into *value-based* and *policy-based*. Value-based algorithms learn the state, or Q-value, by choosing the best action in a state. These algorithms are generally based upon temporal difference learning that we discussed in the RL framework section. Policy-based algorithms (also known as *direct policy search*) directly learn an optimal policy that maps state to action (or tries to approximate optimal policy, if true optimal policy is not attainable).

In most situations in finance, we do not fully know the environment, rewards, or transition probabilities, and we must fall back to model-free algorithms and related approaches.⁶ Hence, the focus of the next section and of the case studies will be the model-free methods and related algorithms.

⁵ If the state and action spaces of MDP are finite, then it is called a finite Markov decision process.

⁶ The MDP example based on dynamic programming that was discussed in the previous section was an example of a model-based algorithm. As seen there, example rewards and transition probabilities are needed for such algorithms.

Figure 9-5 shows a taxonomy of model-free reinforcement learning. We highly recommend that readers refer to *Reinforcement Learning: An Introduction* for a more in-depth understanding of the algorithms and the concepts.

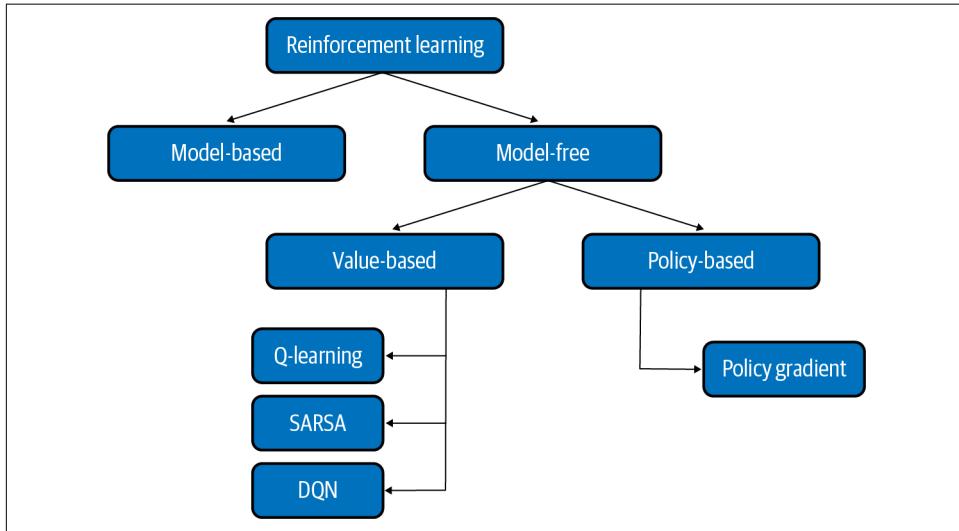


Figure 9-5. Taxonomy of RL models

In the context of model-free methods, temporal difference learning is one of the most used approaches. In TD, the algorithm refines its estimates based on its own prior estimates. The value-based algorithms *Q-learning* and *SARSA* use this approach.

Model-free methods often leverage an artificial neural network to approximate a value or policy function. *Policy gradient* and *deep Q-network (DQN)* are two commonly used model-free algorithms that use artificial neural networks. Policy gradient is a policy-based approach that directly parameterizes the policy. Deep Q-network is a value-based method that combines deep learning with *Q-learning*, which sets the learning objective to optimize the estimates of Q-value.⁷

Q-Learning

Q-learning is an adaptation of TD learning. The algorithm evaluates which action to take based on a Q-value (or action-value) function that determines the value of being in a certain state and taking a certain action at that state. For each state-action pair (s, a) , this algorithm keeps track of a running average of the rewards, R , which the agent gets upon leaving the state s with action a , plus the rewards it expects to earn later.

⁷ There are some models, such as the actor-critic model, that leverage both policy-based and value-based methods.

Since the target policy would act optimally, we take the maximum of the Q-value estimates for the next state.

The learning proceeds *off-policy*—that is, the algorithm does *not* need to select actions based on the policy that is implied by the value function alone. However, convergence requires that all state-action pairs continue to be updated throughout the training process, and a straightforward way to ensure that this occurs is to use an ϵ -*greedy* policy, which is defined further in the following section.

The steps of Q-learning are:

1. At time step t , we start from state s_t and pick an action according to Q-values, $a_t = \max_a Q(s_t, a)$.
2. We apply an ϵ -*greedy* approach that selects an action randomly with a probability of ϵ or otherwise chooses the best action according to the Q-value function. This ensures the exploration of new actions in a given state while also exploiting the learning experience.⁸
3. With action a_t , we observe reward R_{t+1} and get into the next state S_{t+1} .
4. We update the action-value function:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(R_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

5. We increment the time step, $t = t+1$, and repeat the steps.

Given enough iterations of the steps above, this algorithm will converge to the optimal Q-value.

SARSA

SARSA is also a TD learning-based algorithm. It refers to the procedure of updating the Q-value by following a sequence of $\dots, S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}, \dots$. The first two steps of SARSA are similar to the steps of Q-learning. However, unlike Q-learning, SARSA is an *on-policy* algorithm in which the agent grasps the optimal policy and uses the same to act. In this algorithm, the policies used for *updating* and for *acting* are the same. Q-learning is considered an *off-policy* algorithm.

⁸ *Off-policy*, ϵ -*greedy*, *exploration*, and *exploitation* are commonly used terms in RL and will be used in other sections and case studies as well.

Deep Q-Network

In the previous section, we saw how Q-learning allows us to learn the optimal Q-value function in an environment with discrete state actions using iterative updates based on the Bellman equation. However, Q-learning may have the following drawbacks:

- In cases where the state and action space are large, the optimal Q-value table quickly becomes computationally infeasible.
- Q-learning may suffer from instability and divergence.

To address these shortcomings, we use ANNs to approximate Q-values. For example, if we use a function with parameter θ to calculate Q-values, we can label the Q-value function as $Q(s, a; \theta)$. The deep Q-learning algorithm approximates the Q-values by learning a set of weights, θ , of a multilayered deep Q-network that maps states to actions. The algorithm aims to greatly improve and stabilize the training procedure of Q-learning through two innovative mechanisms:

Experience replay

Instead of running Q-learning on state-action pairs as they occur during simulation or actual experience, the algorithm stores the history of state, action, reward, and next state transitions that are experienced by the agent in one large *replay memory*. This can be referred to as a *mini-batch* of observations. During Q-learning updates, samples are drawn at random from the replay memory, and thus one sample could be used multiple times. Experience replay improves data efficiency, removes correlations in the observation sequences, and smooths over changes in the data distribution.

Periodically updated target

Q is optimized toward target values that are only periodically updated. The Q-network is cloned and kept frozen as the optimization targets every C step (C is a hyperparameter). This modification makes the training more stable as it overcomes the short-term oscillations. To learn the network parameters, the algorithm applies *gradient descent*⁹ to a loss function defined as the squared difference between the DQN's estimate of the target and its estimate of the Q-value of the current state-action pair, $Q(s, a; \theta)$. The loss function is as follows:

$$L(\theta_i) = \mathbb{E}_{a'}[(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i))^2]$$

⁹ Refer to [Chapter 3](#) for more details on gradient descent.

The loss function is essentially a mean squared error (MSE) function, where $(r + \gamma \max_a Q(s', a'; \theta_{i-1}))$ represents the target value and $Q[s, a; \theta_i]$ represents the predicted value. θ are the weights of the network, which are computed when the loss function is minimized. Both the target and the current estimate depend on the set of weights, underlining the distinction from supervised learning, in which targets are fixed prior to training.

An example of the DQN for the trading example containing buy, sell, and hold actions is represented in [Figure 9-6](#). Here, we provide the network only the state (s) as input, and we receive Q-values for all possible actions (i.e., buy, sell, and hold) at once. We will be using DQN in the first and third case studies of this chapter.

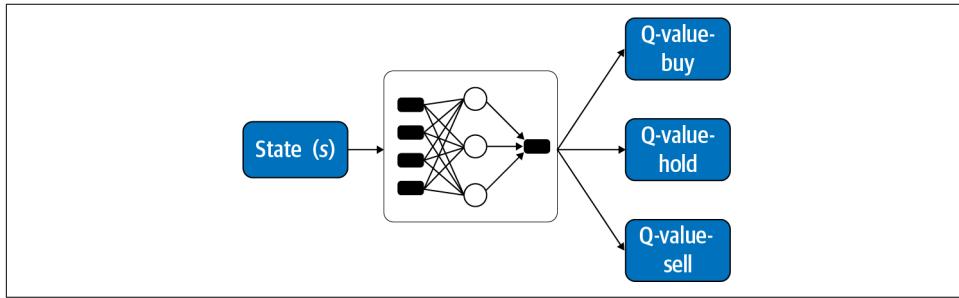


Figure 9-6. DQN

Policy gradient

Policy gradient is a policy-based method in which we learn a policy function, π , which is a direct map from each state to the best corresponding action at that state. It is a more straightforward approach than the value-based method, without the need for a Q-value function.

Policy gradient methods learn the policy directly with a parameterized function respect to θ , $\pi(a|s;\theta)$. This function can be a complex function and might require a sophisticated model. In policy gradient methods, we use ANNs to map state to action because they are efficient at learning complex functions. The loss function of the ANN is the opposite of the expected return (cumulative future rewards).

The objective function of the policy gradient method can be defined as:

$$J(\theta) = V_{\pi_\theta}(S_1) = \mathbb{E}_{\pi_\theta}[V_1]$$

where θ represents a set of weights of the ANN that maps states to actions. The idea here is to maximize the objective function and compute the weights (θ) of the ANN.

Since this is a maximization problem, we optimize the policy by taking the *gradient ascent* (as opposed to gradient descent, which is used to minimize the loss function), with the partial derivative of the objective with respect to the policy parameter θ :

$$\theta \leftarrow \theta + \frac{\partial}{\partial \theta} J(\theta)$$

Using gradient ascent, we can find the best θ that produces the highest return. Computing the gradient numerically can be done by perturbing θ by a small amount ε in the k th dimension or by using an analytical approach for deriving the gradient.

We will be using the policy gradient method for case study 2 later in this chapter.

Key Challenges in Reinforcement Learning

So far, we have covered only what reinforcement learning algorithms can do. However, several shortcomings are outlined below:

Resource efficiency

Current deep reinforcement learning algorithms require vast amounts of time, training data, and computational resources in order to reach a desirable level of proficiency. Thus, making reinforcement learning algorithms trainable under limited resources will continue to be an important issue.

Credit assignment

In RL, reward signals can occur significantly later than actions that contributed to the result, complicating the association of actions with their consequences.

Interpretability

In RL, it is relatively difficult for a model to provide any meaningful, intuitive relationships between input and their corresponding output that can be easily understood. Most advanced reinforcement learning algorithms incorporate deep neural networks, which make interpretability even more difficult due to a large number of layers and nodes inside the neural network.

Let us look at the case studies now.

Case Study 1: Reinforcement Learning-Based Trading Strategy

Algorithmic trading primarily has three components: *policy development*, *parameter optimization*, and *backtesting*. The policy determines what actions to take based on the current state of the market. Parameter optimization is performed using a search over possible values of strategy parameters, such as thresholds or coefficients. Finally,

backtesting assesses the viability of a trading strategy by exploring how it would have played out using historical data.

RL is based around coming up with a policy to maximize the reward in a given environment. Instead of needing to hand code a rule-based trading policy, RL learns one directly. There is no need to explicitly specify rules and thresholds. Their ability to decide policy on their own makes RL models very suitable machine learning algorithms to create automated algorithmic trading models, or *trading bots*.

In terms of *parameter optimization* and *backtesting* steps, RL allows for end-to-end optimization and maximizes (potentially delayed) rewards. Reinforcement learning agents are trained in a simulation, which can be as complex as desired. Taking into account latencies, liquidity, and fees, we can seamlessly combine the backtesting and parameter optimization steps without needing to go through separate stages.

Additionally, RL algorithms learn powerful policies parameterized by artificial neural networks. RL algorithms can also learn to adapt to various market conditions by experiencing them in historical data, given that they are trained over a long-time horizon and have sufficient memory. This allows them to be much more robust to changing markets than supervised learning-based trading strategies, which, due to the simplistic nature of the policy, may not have a parameterization powerful enough to learn to adapt to changing market conditions.

Reinforcement learning, with its capability to easily handle policy, parameter optimization, and backtesting, is ideal for the next wave of algorithmic trading. Anecdotally, it seems that several of the more sophisticated algorithmic execution desks at large investment banks and hedge funds are beginning to use reinforcement learning to optimize their decision making.

In this case study, we will create an end-to-end trading strategy based on reinforcement learning. We will use the Q-learning approach with deep Q-network (DQN) to come up with a policy and an implementation of the trading strategy. As discussed before, the name “Q-learning” is in reference to the $Q(s, a)$ function, which returns the expected reward based on the state s and provided action a . In addition to developing a specific trading strategy, this case study will discuss the general framework and components of a reinforcement learning-based trading strategy.

In this case study, we will focus on:

- Understanding the key components of an RL framework from a trading strategy standpoint.
- Evaluating the Q-learning method of RL in Python by defining an agent, followed by training and testing setup.

- Implementing a deep neural network to be used for RL problems in Python using the Keras package.
- Understanding the class structure of Python programming while implementing an RL-based model.
- Understanding the intuition and interpretation of RL-based algorithms.



Blueprint for Creating a Reinforcement Learning–Based Trading Strategy

1. Problem definition

In the reinforcement learning framework for this case study, the algorithm takes an action (buy, sell, or hold) depending on the current state of the stock price. The algorithm is trained using a deep Q-learning model to perform the best action. The key components of the reinforcement learning framework for this case study are:

Agent

Trading agent.

Action

Buy, sell, or hold.

Reward function

Realized profit and loss (PnL) is used as the reward function for this case study. The reward depends on the action: sell (realized profit and loss), buy (no reward), or hold (no reward).

State

A sigmoid function¹⁰ of the differences of past stock prices for a given time window is used as the state. State S_t is described as $(d_{t-\tau+1}, d_{t-1}, d_t)$, where $d_T = \text{sigmoid}(p_t - p_{t-1})$, p_t is price at time t , and τ is the time window size. A sigmoid function converts the differences of the past stock prices into a number between zero and one, which helps to normalize the values to probabilities and makes the state simpler to interpret.

Environment

Stock exchange or the stock market.

¹⁰ Refer to [Chapter 3](#) for more details on the sigmoid function.



Selecting the RL Components for a Trading Strategy

Formulating an intelligent behavior for a reinforcement learning-based trading strategy begins with identification of the correct components of the RL model. Hence, before we go into the model development, we should carefully identify the following RL components:

Reward function

This is an important parameter, as it decides whether the RL algorithm will learn to optimize the appropriate metric. In addition to the return or PnL, the reward function can incorporate risk embedded in the underlying instrument or include other parameters such as volatility or maximum drawdown. It can also include the transaction costs of the buy/sell actions.

State

State determines the observations that the agent receives from the environment for taking a decision. The state should be representative of current market behavior as compared to the past and can also include values of any signals that are believed to be predictive or items related to market micro-structure, such as volume traded.

The data that we will use will be the S&P 500 closing prices. The data is extracted from Yahoo Finance and contains ten years of daily data from 2010 to 2019.

2. Getting started—loading the data and Python packages

2.1. Loading the Python packages. The list of libraries used for all of the steps of model implementation, from *data loading* to *model evaluation*, including deep learning-based model development, are included here. The details of most of these packages and functions have been provided in Chapters 2, 3, and 4. The packages used for different purposes have been separated in the Python code here, and their usage will be demonstrated in different steps of the model development process.

Packages for reinforcement learning

```
import keras
from keras import layers, models, optimizers
from keras import backend as K
from collections import namedtuple, deque
from keras.models import Sequential
from keras.models import load_model
from keras.layers import Dense
from keras.optimizers import Adam
```

Packages/modules for data processing and visualization

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from pandas import read_csv, set_option
import datetime
import math
from numpy.random import choice
import random
from collections import deque
```

2.2. Loading the data. The fetched data for the time period of 2010 to 2019 is loaded:

```
dataset = read_csv('data/SP500.csv', index_col=0)
```

3. Exploratory data analysis

We will look at descriptive statistics and data visualization in this section. Let us have a look at the dataset we have:

```
# shape
dataset.shape
```

Output

```
(2515, 6)

# peek at data
set_option('display.width', 100)
dataset.head(5)
```

Output

Date	Open	High	Low	Close	Adj Close	Volume
2001-01-02	1320.28	1320.28	1276.05	1283.27	1283.27	1129400000
2001-01-03	1283.27	1347.76	1274.62	1347.56	1347.56	1880700000
2001-01-04	1347.56	1350.24	1329.14	1333.34	1333.34	2131000000
2001-01-05	1333.34	1334.77	1294.95	1298.35	1298.35	1430800000
2001-01-08	1298.35	1298.35	1276.29	1295.86	1295.86	1115500000

The data has a total of 2,515 rows and six columns, which contain the categories *open*, *high*, *low*, *close*, *adjusted close price*, and *total volume*. The adjusted close price is the closing price adjusted for the split and dividends. For the purpose of this case study, we will be focusing on the closing price.



The chart shows that S&P 500 has been in an upward-trending series between 2010 and 2019. Let us perform the data preparation.

4. Data preparation

This step is important in order to create a meaningful, reliable, and clean dataset that can be used without any errors in the reinforcement learning algorithm.

4.1. Data cleaning. In this step, we check for NAs in the rows and either drop them or fill them with the mean of the column:

```
#Checking for any null values and removing the null values!!!
print('Null Values =', dataset.isnull().values.any())
```

Output

```
Null Values = False
```

As there are no null values in the data, there is no need to perform any further data cleaning.

5. Evaluate algorithms and models

This is the key step of the reinforcement learning model development, where we will define all the relevant functions and classes and train the algorithm. In the first step, we prepare the data for the training set and the test set.

5.1. Train-test split. In this step, we partition the original dataset into training set and test set. We use the test set to confirm the performance of our final model and to understand if there is any overfitting. We will use 80% of the dataset for modeling and 20% for testing:

```

X=list(dataset["Close"])
X=[float(x) for x in X]
validation_size = 0.2
train_size = int(len(X) * (1-validation_size))
X_train, X_test = X[0:train_size], X[train_size:len(X)]

```

5.2. Implementation steps and modules. The overall algorithm of this case study (and of reinforcement learning in general) is a bit complex as it requires building *class-based code structure* and the simultaneous use of many modules and functions. This additional section was added for this case study to provide a functional explanation of what is happening in the program.

The algorithm, in simple terms, decides whether to buy, sell, or hold when provided with the current market price.

Figure 9-7 provides an overview of the training of the Q-learning-based algorithm in the context of this case study. The algorithm evaluates which action to take based on a Q-value, which determines the value of being in a certain state and taking a certain action at that state.

As per Figure 9-7, the state (s) is decided on the basis of the current and historical behavior of the price (P_t, P_{t-1}, \dots). Based on the current state, the action is “buy.” With this action, we observe a reward of \$10 (i.e., the PnL associated with the action) and move into the next state. Using the current reward and the next state’s Q-value, the algorithm updates the Q-value function. The algorithm keeps on moving through the next time steps. Given sufficient iterations of the steps above, this algorithm will converge to the optimal Q-value.

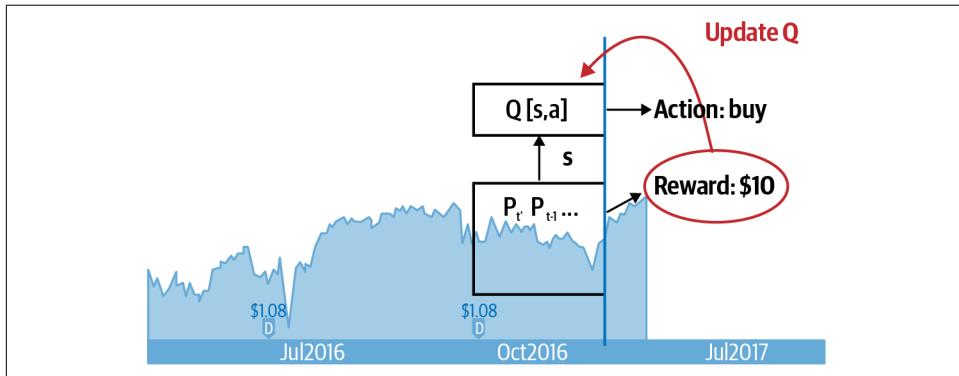


Figure 9-7. Reinforcement learning for trading

The deep Q-network that we use in this case study uses an ANN to approximate Q-values; hence, the action value function is defined as $Q(s,a;\theta)$. The deep Q-learning algorithm approximates the Q-value function by learning a set of weights, θ , of a multilayered DQN that maps states to actions.

Modules and functions

Implementing this DQN algorithm requires implementation of several functions and modules that interact with each other during the model training. Here is a summary of the modules and functions:

Agent class

The agent is defined as “Agent” class. This holds the variables and member functions that perform the Q-learning. An object of the Agent class is created using the training phase and is used for training the model.

Helper functions

In this module, we create additional functions that are helpful for training.

Training module

In this step, we perform the training of the data using the variables and the functions defined in the agent and helper methods. During training, the prescribed action for each day is predicted, the rewards are computed, and the deep learning-based Q-learning model weights are updated iteratively over a number of episodes. Additionally, the profit and loss of each action is summed to determine whether an overall profit has occurred. The aim is to maximize the total profit.

We provide a deep dive into the interaction between the different modules and functions in “[5.5. Training the model](#)” on page 308.

Let us look at each of these in detail.

5.3. Agent class. The agent class consists of the following components:

- Constructor
- Function `model`
- Function `act`
- Function `expReplay`

The `Constructor` is defined as `init` function and contains important parameters such as `discount factor` for reward function, `epsilon` for the ϵ -greedy approach, `state size`, and `action size`. The number of actions is set at three (i.e., buy, sell, and hold). The `memory` variable defines the `replay memory` size. The input parameter of this function also consists of `is_eval` parameter, which defines whether training is ongoing. This variable is changed to `True` during the evaluation/testing phase. Also, if the pretrained model has to be used in the evaluation/training phase, it is passed using the `model_name` variable:

```

class Agent:
    def __init__(self, state_size, is_eval=False, model_name=""):
        self.state_size = state_size # normalized previous days
        self.action_size = 3 # hold, buy, sell
        self.memory = deque(maxlen=1000)
        self.inventory = []
        self.model_name = model_name
        self.is_eval = is_eval

        self.gamma = 0.95
        self.epsilon = 1.0
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.995

    self.model = load_model("models/" + model_name) if is_eval \
        else self._model()

```

The function `model` is a deep learning model that maps the states to actions. This function takes in the state of the environment and returns a *Q-value* table or a policy that refers to a probability distribution over actions. This function is built using the Keras Python library.¹¹ The architecture for the deep learning model used is:

- The model expects rows of data with number of variables equal to the *state size*, which comes as an input.
- The first, second, and third hidden layers have 64, 32, and 8 nodes, respectively, and all of these layers use the ReLU activation function.
- The output layer has the number of nodes equal to the action size (i.e., three), and the node uses a linear activation function:¹²

```

def _model(self):
    model = Sequential()
    model.add(Dense(units=64, input_dim=self.state_size, activation="relu"))
    model.add(Dense(units=32, activation="relu"))
    model.add(Dense(units=8, activation="relu"))
    model.add(Dense(self.action_size, activation="linear"))
    model.compile(loss="mse", optimizer=Adam(lr=0.001))

    return model

```

The function `act` returns an action given a state. The function uses the `model` function and returns a buy, sell, or hold action:

¹¹ The details of the Keras-based implementation of deep learning models are shown in [Chapter 3](#).

¹² Refer to [Chapter 3](#) for more details on the linear and ReLU activation functions.

```

def act(self, state):
    if not self.is_eval and random.random() <= self.epsilon:
        return random.randrange(self.action_size)

    options = self.model.predict(state)
    return np.argmax(options[0])

```

The function `expReplay` is the key function, where the neural network is trained based on the observed experience. This function implements the *Experience replay* mechanism as previously discussed. Experience replay stores a history of state, action, reward, and next state transitions that are experienced by the agent. It takes a mini-batch of the observations (*replay memory*) as an input and updates the deep learning-based Q-learning model weights by minimizing the loss function. The *epsilon greedy* approach implemented in this function prevents overfitting. In order to explain the function, different steps are numbered in the comments of the following Python code, along with an outline of the steps:

1. Prepare the replay buffer memory, which is the set of observation used for training. New experiences are added to the replay buffer memory using a for loop.
2. *Loop* across all the observations of state, action, reward, and next state transitions in the mini-batch.
3. The target variable for the Q-table is updated based on the Bellman equation. The update happens if the current state is the terminal state or the end of the episode. This is represented by the variable `done` and is defined further in the training function. If it is not done, the target is just set to reward.
4. Predict the Q-value of the next state using a deep learning model.
5. The Q-value of this state for the action in the current replay buffer is set to the target.
6. The deep learning model weights are updated by using the `model.fit` function.
7. The epsilon greedy approach is implemented. Recall that this approach selects an action randomly with a probability of ϵ or the best action, according to the Q-value function, with probability $1-\epsilon$.

```

def expReplay(self, batch_size):
    mini_batch = []
    l = len(self.memory)
    #1: prepare replay memory
    for i in range(l - batch_size + 1, l):
        mini_batch.append(self.memory[i])

    #2: Loop across the replay memory batch.
    for state, action, reward, next_state, done in mini_batch:
        target = reward # reward or Q at time t
        #3: update the target for Q table. table equation
        if not done:

```

```

        target = reward + self.gamma * \
            np.amax(self.model.predict(next_state)[0])
    #set_trace()

    # 4: Q-value of the state currently from the table
    target_f = self.model.predict(state)
    # 5: Update the output Q table for the given action in the table
    target_f[0][action] = target
    # 6. train and fit the model.
    self.model.fit(state, target_f, epochs=1, verbose=0)

#7. Implement epsilon greedy algorithm
if self.epsilon > self.epsilon_min:
    self.epsilon *= self.epsilon_decay

```

5.4. Helper functions. In this module, we create additional functions that are helpful for training. Some of the important helper functions are discussed here. For details about other helper functions, refer to the Jupyter notebook in the GitHub repository for this book.

The function `getState` generates the states given the stock data, time t (the day of prediction), and window n (number of days to go back in time). First, the vector of price difference is computed, followed by scaling this vector from zero to one with a `sigmoid` function. This is returned as the state.

```

def getState(data, t, n):
    d = t - n + 1
    block = data[d:t + 1] if d >= 0 else -d * [data[0]] + data[0:t + 1]
    res = []
    for i in range(n - 1):
        res.append(sigmoid(block[i + 1] - block[i]))
    return np.array([res])

```

The function `plot_behavior` returns the plot of the market price along with indicators for the buy and sell actions. It is used for the overall evaluation of the algorithm during the training and testing phase.

```

def plot_behavior(data_input, states_buy, states_sell, profit):
    fig = plt.figure(figsize = (15, 5))
    plt.plot(data_input, color='r', lw=2.)
    plt.plot(data_input, '^', markersize=10, color='m', label='Buying signal',\
        markevery=states_buy)
    plt.plot(data_input, 'v', markersize=10, color='k', label='Selling signal',\
        markevery = states_sell)
    plt.title('Total gains: %f'%(profit))
    plt.legend()
    plt.show()

```

5.5. Training the model. We will proceed to train the data. Based on our agent, we define the following variables and instantiate the stock agent:

Episode

The number of times the code is trained through the entire data. In this case study, we use 10 episodes.

Windows size

Number of market days to consider to evaluate the state.

Batch size

Size of the replay buffer or memory use during training.

Once these variables are defined, we train the model iterating through the episodes. [Figure 9-8](#) provides a deep dive into the training steps and brings together all the elements discussed so far. The upper section showing steps 1 to 7 describes the steps in the *training* module, and the lower section describes the steps in the *replay buffer* function (i.e., `exeReplay` function).

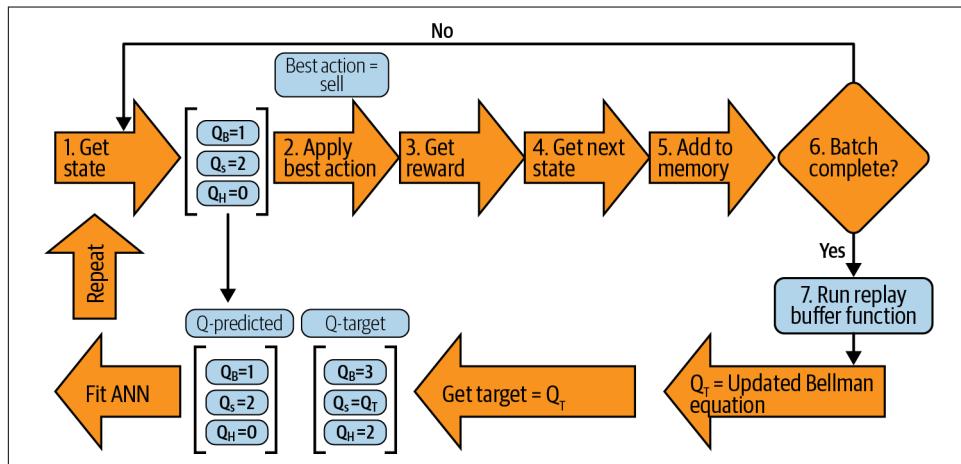


Figure 9-8. Training steps of Q-trading

Steps 1 to 6 shown in [Figure 9-8](#) are numbered in the following Python code and are described as follows:

1. Get the current state using the helper function `getState`. It returns a vector of states, where the length of the vector is defined by windows size and the values of the states are between zero and one.
2. Get the action for the given state using the `act` function of the agent class.
3. Get the reward for the given action. The mapping of the action and reward is described in the problem definition section of this case study.
4. Get the next state using the `getstate` function. The detail of the next state is further used in the Bellman equation for updating the Q-function.

5. The details of the state, next state, action, etc., are saved in the memory of the agent object, which is used further by the `exeReply` function. A sample mini-batch is as follows:

State	action	reward	next_state	done
0.5000	2	0	1.0000	False
1.0000	0	0	0.0000	False
0.0000	1	0	0.0000	False
0.0000	0	0	0.0766	False
0.0766	1	0	0.9929	False
0.9929	2	0	1.0000	False
1.0000	2	17.410	1.0000	False
1.0000	2	0	0.0003	False
0.0003	2	0	0.9997	False
0.9997	1	0	0.9437	False

6. Check if the batch is complete. The size of a batch is defined by the batch size variable. If the batch is complete, then we move to the `Replay buffer` function and update the Q-function by minimizing the MSE between the Q-predicted and the Q-target. If not, then we move to the next time step.

The code produces the final results of each episode, along with the plot showing the buy and sell actions and the total profit for each episode of the training phase.

```
window_size = 1
agent = Agent(window_size)
l = len(data) - 1
batch_size = 10
states_sell = []
states_buy = []
episode_count = 3

for e in range(episode_count + 1):
    print("Episode " + str(e) + "/" + str(episode_count))
    # 1-get state
    state = getState(data, 0, window_size + 1)

    total_profit = 0
    agent.inventory = []

    for t in range(l):
        # 2-apply best action
        action = agent.act(state)

        # sit
        next_state = getState(data, t + 1, window_size + 1)
        reward = 0

        if action == 1: # buy
```

```

agent.inventory.append(data[t])
states_buy.append(t)
print("Buy: " + formatPrice(data[t]))

elif action == 2 and len(agent.inventory) > 0: # sell
    bought_price = agent.inventory.pop(0)
    #3: Get Reward

    reward = max(data[t] - bought_price, 0)
    total_profit += data[t] - bought_price
    states_sell.append(t)
    print("Sell: " + formatPrice(data[t]) + " | Profit: " \
        + formatPrice(data[t] - bought_price))

done = True if t == l - 1 else False
# 4: get next state to be used in bellman's equation
next_state = getState(data, t + 1, window_size + 1)

# 5: add to the memory
agent.memory.append((state, action, reward, next_state, done))
state = next_state

if done:

    print("-----")
    print("Total Profit: " + formatPrice(total_profit))
    print("-----")

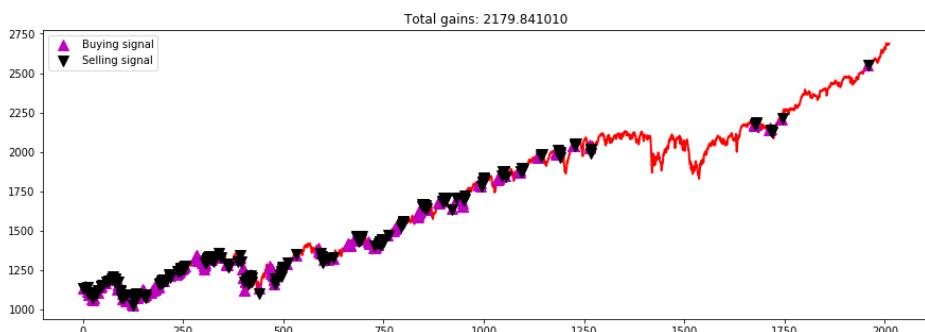
# 6: Run replay buffer function
if len(agent.memory) > batch_size:
    agent.expReplay(batch_size)

if e % 10 == 0:
    agent.model.save("models/model_ep" + str(e))

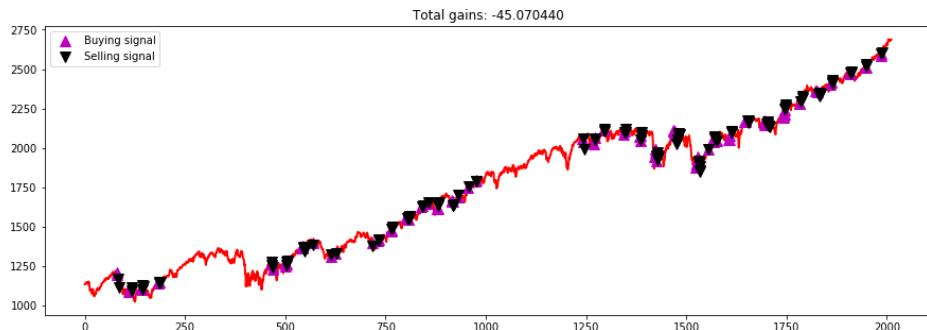
```

Output

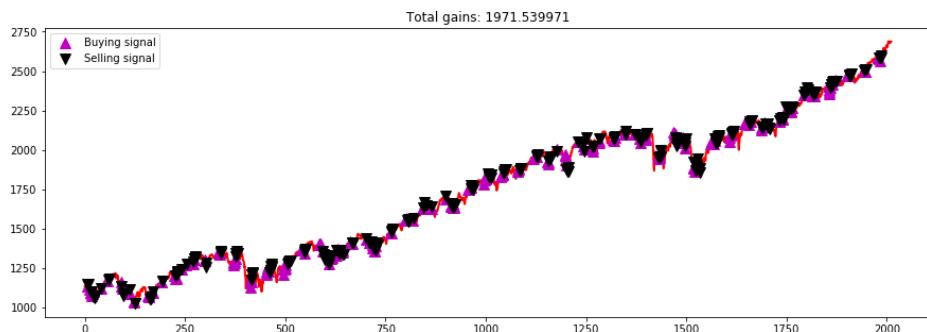
Running episode 0/10
 Total Profit: \$6738.87



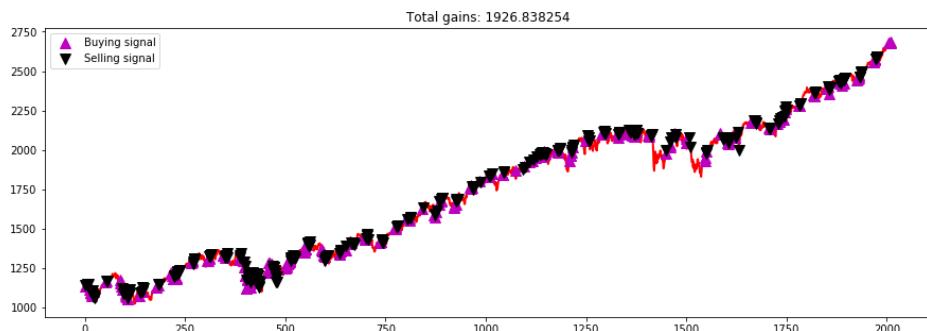
Running episode 1/10
Total Profit: -\$45.07



Running episode 9/10
Total Profit: \$1971.54



Running episode 10/10
Total Profit: \$1926.84



The charts show the details of the buy/sell pattern and the total gains of the first two (zero and one) and last two (9 and 10) episodes. The details of other episodes can be seen in Jupyter notebook under the GitHub repository for this book.

As we can see, in the beginning of episodes 0 and 1, since the agent has no preconception of the consequences of its actions, it takes randomized actions to observe the rewards associated with it. In episode zero, there is an overall profit of \$6,738, a strong result indeed, but in episode one we experience an overall loss of \$45. The fact that the cumulative reward per episode fluctuates substantially in the beginning illustrates the exploration process the algorithm is going through. Looking at episodes 9 and 10, it seems as though the agent begins learning from its training. It discovers the strategy and starts to exploit it consistently. The buy and sell actions of these last two episodes lead a PnL that is perhaps less than that of episode zero, but far more robust. The buy and sell actions in the later episodes have been performed uniformly over the entire time period, and the overall profit is stable.

Ideally, the number of training episodes should be higher than the number used in this case study. A higher number of training episodes will lead to a better training performance. Before we move on to the testing, let us go through the details about model tuning.

5.6. Model tuning. Similar to other machine learning techniques, we can find the best combination of model hyperparameters in RL by using techniques such as grid search. The grid search for RL-based problems are computationally intensive. Hence, in this section, rather than performing the grid search, we present the key hyperparameters to consider, along with their intuition and potential impact on the model output.

Gamma (discount factor)

Decaying gamma will have the agent prioritize short-term rewards as it learns what those rewards are, and place less emphasis on long-term rewards. Lowering the discount factor in this case study may cause the algorithm to focus on the long-term rewards.

Epsilon

The epsilon variable drives the *exploration versus exploitation* property of the model. The more we get to know our environment, the less random exploration we want to do. When we reduce epsilon, the likelihood of a random action becomes smaller, and we take more opportunities to benefit from the high-valued actions that we already discovered. However, in the trading setup, we do not want the algorithm to *overfit* to the training data, and the epsilon should be modified accordingly.

Episodes and batch size

A higher number of episodes and larger batch size in the training set will lead to better training and a more optimal Q-value. However, there is a trade-off, as increasing the number of episodes and batch size increases the total training time.

Window size

Window size determines the number of market days to consider to evaluate the state. This can be increased in case we want the state to be determined by a greater number of days in the past.

Number of layers and nodes of the deep learning model

This can be modified for better training and a more optimal Q-value. The details about the impact of changing the layers and nodes of ANN models are discussed in [Chapter 3](#), and the grid search for a deep learning model is discussed in [Chapter 5](#).

6. Testing the data

After training the data, it is evaluated against the test dataset. This is an important step, especially for reinforcement learning, as the agent may mistakenly correlate reward with certain spurious features from the data, or it may overfit a particular chart pattern. In the testing step, we look at the performance of the already trained model (`model_ep10`) from the training step on the test data. The Python code looks similar to the training set we saw before. However, the `is_eval` flag is set to `true`, the `reply_buffer` function is not called, and there is no training. Let us look at the results:

```
#agent is already defined in the training set above.
test_data = X_test
l_test = len(test_data) - 1
state = getState(test_data, 0, window_size + 1)
total_profit = 0
is_eval = True
done = False
states_sell_test = []
states_buy_test = []
model_name = "model_ep10"
agent = Agent(window_size, is_eval, model_name)
state = getState(data, 0, window_size + 1)
total_profit = 0
agent.inventory = []

for t in range(l_test):
    action = agent.act(state)

    next_state = getState(test_data, t + 1, window_size + 1)
    reward = 0
```

```

if action == 1:

    agent.inventory.append(test_data[t])
    print("Buy: " + formatPrice(test_data[t]))

elif action == 2 and len(agent.inventory) > 0:
    bought_price = agent.inventory.pop(0)
    reward = max(test_data[t] - bought_price, 0)
    total_profit += test_data[t] - bought_price
    print("Sell: " + formatPrice(test_data[t]) + " | profit: " + \
        formatPrice(test_data[t] - bought_price))

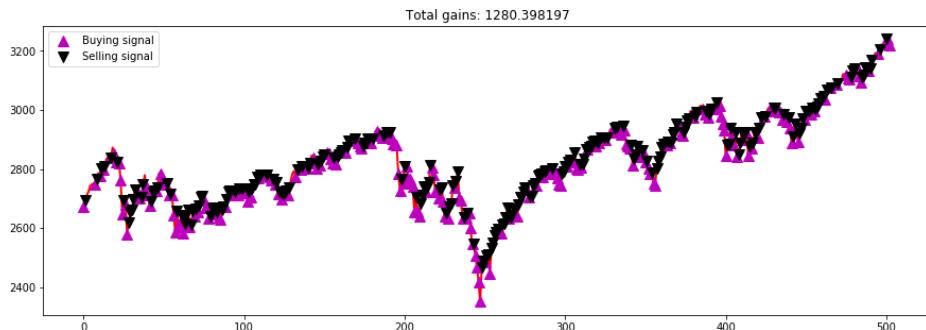
if t == l_test - 1:
    done = True
    agent.memory.append((state, action, reward, next_state, done))
    state = next_state

if done:
    print("-----")
    print("Total Profit: " + formatPrice(total_profit))
    print("-----")

```

Output

Total Profit: \$1280.40



Looking at the results above, our model resulted in an overall profit of \$1,280, and we can say that our DQN agent performs quite well on the test set.

Conclusion

In this case study, we created an automated trading strategy, or a *trading bot*, that simply needs to be fed running stock market data to produce a trading signal. We saw that the algorithm decides the policy by itself, and the overall approach is much simpler and more principled than the supervised learning-based approach. The trained

model was profitable in the test set, corroborating the effectiveness of the RL-based trading strategy.

In using a reinforcement learning model such as DQN, which is based on a deep neural network, we can learn policies that are more complex and powerful than what a human trader could learn.

Given the high complexity and low interpretability of the RL-based model, visualization and testing steps become quite important. For interpretability, we used the plots of the training episodes of the training algorithm and found that the model starts to learn over a period of time, discovers the strategy, and starts to exploit it. A sufficient number of tests should be conducted on different time periods before deploying the model for live trading.

While using RL-based models, we should carefully select the RL components, such as the reward function and state, and ensure understanding of their impact on the overall model results. Before implementing or training the model, it is important to think of questions, such as “How can we engineer the reward function or the state so that the RL algorithm has the potential to learn to optimize the right metric?”

Overall, these RL-based models can enable financial practitioners to create trading strategies with a very flexible approach. The framework provided in this case study can be a great starting point to develop more powerful models for algorithmic trading.

Case Study 2: Derivatives Hedging

Much of traditional finance theory for handling derivatives pricing and risk management is based on the idealized complete markets assumption of perfect hedgability, without trading restrictions, transaction costs, market impact, or liquidity constraints. In practice, however, these frictions are very real. As a consequence, practical risk management using derivatives requires human oversight and maintenance; the models themselves are insufficient. Implementation is still partially driven by the trader’s intuitive understanding of the shortcomings of the existing tools.

Reinforcement learning algorithms, with their ability to tackle more nuances and parameters within the operational environment, are inherently aligned with the objective of hedging. These models can produce dynamic strategies that are optimal, even in a world with frictions. The model-free RL approaches demand very few theoretical assumptions. This allows for automation of hedging without requiring frequent human intervention, making the overall hedging process significantly faster. These models can learn from large amounts of historical data and can consider many variables to make more precise and accurate hedging decisions. Moreover, the availa-

bility of vast amounts of data makes RL-based models more useful and effective than ever before.

In this case study, we implement a reinforcement learning-based hedging strategy that adopts the ideas presented in the paper “[Deep Hedging](#)” by Hans Bühler et al. We will build an optimal hedging strategy for a specific type of derivative (call options) by minimizing the risk-adjusted PnL. We use the measure *CVaR* (conditional value at risk), which quantifies the amount of tail risk of a position or portfolio as a risk assessment measure.

In this case study, we will focus on:

- Using policy-based (or direct policy search-based) reinforcement learning and implementing it using a deep neural network.
- Comparing the effectiveness of an RL-based trading strategy to the traditional Black-Scholes model.
- Setting up an agent for an RL problem using class structure in Python.
- Implementing and evaluating a policy gradient-based RL method.
- Introducing the basic concept of functions in the TensorFlow Python package.
- Implementing a Monte Carlo simulation of stock price and the Black-Scholes pricing model, and computing option Greeks.



Blueprint for Implementing a Reinforcement Learning-Based Hedging Strategy

1. Problem definition

In the reinforcement learning framework for this case study, the algorithm decides the best hedging strategy for call options using market prices of the underlying asset. A direct policy search reinforcement learning strategy is used. The overall idea, derived from the “Deep Hedging” paper, is based on minimizing the hedge error under a risk assessment measure. The overall PnL of a call option hedging strategy over a period of time, from $t=1$ to $t=T$, can be written as:

$$PnL_T(Z, \delta) = -Z_T + \sum_{t=1}^T \delta_{t-1}(S_t - S_{t-1}) - \sum_{t=1}^T C_t$$

where

- Z_T is the payoff of a call option at maturity.
- $\delta_{t-1}(S_t - S_{t-1})$ is the cash flow from the hedging instruments on day t , where δ is the hedge and S_t is the spot price on day t .
- C_t is the transaction cost at time t and may be constant or proportional to the hedge size.

The individual components in the equation are the components of the cash flow. However, it would be preferable to take into account the risk arising from any position while designing the reward function. We use the measure CVaR as the risk assessment measure. CVaR quantifies the amount of tail risk and is the **expected shortfall** (risk aversion parameter)¹³ for the confidence level α . Now the reward function is modified to the following:

$$V_T = f \left(-Z_T + \sum_{t=1}^T \delta_{t-1}(S_t - S_{t-1}) - \sum_{t=1}^T C_t \right)$$

where f represents the CVaR.

We will train an *RNN-based* network to learn the optimal hedging strategy (i.e., $\delta_1, \delta_2, \dots, \delta_T$) given the stock price, strike price, and risk aversion parameter, (α), by minimizing CVaR. We assume transaction costs to be zero for simplicity. The model can easily be extended to incorporate transaction costs and other market frictions.

The data used for the synthetic underlying stock price is generated using Monte Carlo simulation, assuming a lognormal price distribution. We assume an interest rate of 0% and annual volatility of 20%.

The key components of the model are:

Agent

Trader or trading agent.

Action

Hedging strategy (i.e., $\delta_1, \delta_2, \dots, \delta_T$).

Reward function

CVaR—this is a convex function and is minimized during the model training.

¹³ The expected shortfall is the expected value of an investment in the tail scenario.

State

State is the representation of the current market and relevant product variables. The state represents the model inputs, which include the simulated stock price path (i.e., S_1, S_2, \dots, S_T), strike, and risk aversion parameter (α).

Environment

Stock exchange or stock market.

2. Getting started

2.1. Loading the Python packages. The loading of Python packages is similar to the previous case studies. Please refer to the Jupyter notebook for this case study for more details.

2.2. Generating the data. In this step we generate the data for this case study using a Black-Scholes simulation.

This function generates the Monte Carlo paths for the stock price and gets the option price on each of the Monte Carlo paths. The calculation as shown is based on the log-normal assumption of stock prices:

$$S_{t+1} = S_t e^{\left(\mu - \frac{1}{2}\sigma^2\right)\Delta t + \sigma\sqrt{\Delta t}Z}$$

where S is stock price, σ is volatility, μ is the drift, t is time, and Z is a standard normal variable.

```
def monte_carlo_paths(S_0, time_to_expiry, sigma, drift, seed, n_sims, \
n_timesteps):
    """
    Create random paths of a stock price following a brownian geometric motion
    return:
        a (n_timesteps x n_sims x 1) matrix
    """
    if seed > 0:
        np.random.seed(seed)
    stdnorm_random_variates = np.random.randn(n_sims, n_timesteps)
    S = S_0
    dt = time_to_expiry / stdnorm_random_variates.shape[1]
    r = drift
    S_T = S * np.cumprod(np.exp((r-sigma**2/2)*dt+sigma*np.sqrt(dt)*\
    stdnorm_random_variates), axis=1)
    return np.reshape(np.transpose(np.c_[np.ones(n_sims)*S_0, S_T]), \
(n_timesteps+1, n_sims, 1))
```

We generate 50,000 simulations of the spot price over a period of one month. The total number of time steps is 30. Hence, for each Monte Carlo scenario, there is one observation per day. The parameters needed for the simulation are defined below:

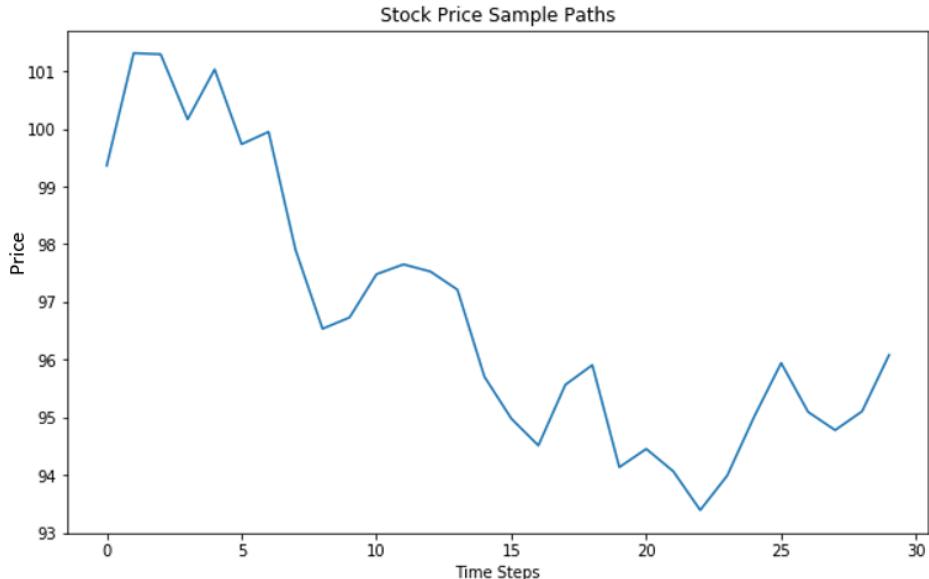
```
S_0 = 100; K = 100; r = 0; vol = 0.2; T = 1/12  
timesteps = 30; seed = 42; n_sims = 5000  
  
# Generate the monte carlo paths  
paths_train = monte_carlo_paths(S_0, T, vol, r, seed, n_sims, timesteps)
```

3. Exploratory data analysis

We will look at descriptive statistics and data visualization in this section. Given that the data was generated by the simulation, we simply inspect one path as a sanity check of the simulation algorithm:

```
#Plot Paths for one simulation  
plt.figure(figsize=(20, 10))  
plt.plot(paths_train[1])  
plt.xlabel('Time Steps')  
plt.title('Stock Price Sample Paths')  
plt.show()
```

Output



4. Evaluate algorithms and models

In this direct policy search approach, we use an artificial neural network (ANN) to map the state to action. In a traditional ANN, we assume that all inputs (and outputs) are independent of each other. However, the hedging decision at time t (represented by δ_t) is *path dependent* and is influenced by the stock price and hedging decisions at previous time steps. Hence, using a traditional ANN is not feasible. RNN is a type of ANN that can capture the time-varying dynamics of the underlying system and is more appropriate in this context. RNNs have a memory, which captures information about what has been calculated so far. We used this property of the RNN model for time series modeling in [Chapter 5](#). LSTM (also discussed in [Chapter 5](#)) is a special kind of RNN capable of learning long-term dependencies. Past state information is made available to the network when mapping to an action; the extraction of relevant past data is then learned as part of the training process. We will use an LSTM model to map the state to action and get the hedging strategy (i.e., $\delta_1, \delta_2, \dots, \delta_T$).

4.1. Policy gradient script. We will cover the implementation steps and model training in this section. We provide the input variables—stock price path (S_1, S_2, \dots, S_T), strike, and risk aversion parameter, α —to the trained model and receive the hedging strategy (i.e., $\delta_1, \delta_2, \dots, \delta_T$) as the output. [Figure 9-9](#) provides an overview of the training of the policy gradient for this case study.

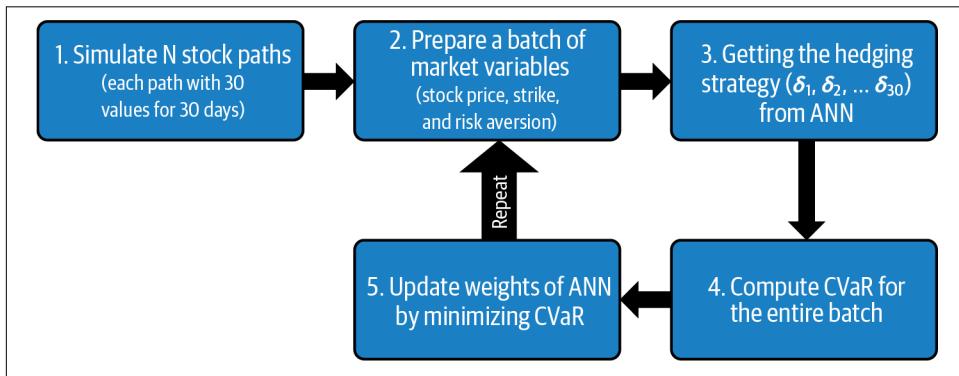


Figure 9-9. Policy gradient training for derivatives hedging

We already performed step 1 in section 2 of this case study. Steps 2 to 5 are self-explanatory and are implemented in the `agent` class defined later. The `agent` holds the variables and member functions that perform the training. An object of the `agent` class is created using the training phase and is used for training the model. After a sufficient number of iterations of steps 2 to 5, an optimal policy gradient model is generated.

The class consists of the following modules:

- Constructor
- The function `execute_graph_batchwise`
- The functions `training`, `predict`, and `restore`

Let us dig deeper into the Python code for each of the functions.

The `Constructor` is defined as an `init` function, where we define the model parameters. We can pass the `timesteps`, `batch_size`, and `number of nodes` in each layer of the LSTM model to the constructor. We define the input variables of the model (i.e., stock price path, strike, and risk aversion parameter) as *TensorFlow placeholders*. Placeholders are used to feed in data from outside the computational graph, and we feed the data of these input variables during the training phase. We implement an LSTM network in TensorFlow by using the `tf.MultiRNNCell` function. The LSTM model uses four layers with 62, 46, 46, and 1 nodes. The loss function is the CVaR, which is minimized when `tf.train` is called during the training step. We sort the negative realized PnLs of the trading strategy and calculate the mean of the $(1-\alpha)$ top losses:

```
class Agent(object):
    def __init__(self, time_steps, batch_size, features,\n        nodes = [62, 46, 46, 1], name='model'):\n\n        #1. Initialize the variables\n        tf.reset_default_graph()\n        self.batch_size = batch_size # Number of options in a batch\n        self.S_t_input = tf.placeholder(tf.float32, [time_steps, batch_size, \n            features]) #Spot\n        self.K = tf.placeholder(tf.float32, batch_size) #Strike\n        self.alpha = tf.placeholder(tf.float32) #alpha for cVaR\n\n        S_T = self.S_t_input[-1,:,:] #Spot at time T\n        # Change in the Spot\n        dS = self.S_t_input[1:, :, 0] - self.S_t_input[0:-1, :, 0]\n        #dS = tf.reshape(dS, (time_steps, batch_size))\n\n        #2. Prepare S_t for use in the RNN remove the |\n        #last time step (at T the portfolio is zero)\n        S_t = tf.unstack(self.S_t_input[:-1, :, :], axis=0)\n\n        # Build the lstm\n        lstm = tf.contrib.rnn.MultiRNNCell([tf.contrib.rnn.LSTMCell(n) \\n\n        for n in nodes])\n\n        #3. So the state is a convenient tensor that holds the last\n        #actual RNN state, ignoring the zeros.
```

```

#The strategy tensor holds the outputs of all cells.
self.strategy, state = tf.nn.static_rnn(lstm, S_t, initial_state=\
    lstm.zero_state(batch_size, tf.float32), dtype=tf.float32)

self.strategy = tf.reshape(self.strategy, (time_steps-1, batch_size))

#4. Option Price
self.option = tf.maximum(S_T-self.K, 0)

self.Hedging_PnL = - self.option + tf.reduce_sum(dS*self.strategy, \
    axis=0)

#5. Total Hedged PnL of each path
self.Hedging_PnL_Paths = - self.option + dS*self.strategy

# 6. Calculate the CVaR for a given confidence level alpha
# Take the 1-alpha largest losses (top 1-alpha negative PnLs)
#and calculate the mean
CVaR, idx = tf.nn.top_k(-self.Hedging_PnL, tf.cast((1-self.alpha)*\
batch_size, tf.int32))
CVaR = tf.reduce_mean(CVaR)
#7. Minimize the CVaR
self.train = tf.train.AdamOptimizer().minimize(CVaR)
self.saver = tf.train.Saver()
self.modelname = name

```

The function `execute_graph_batchwise` is the key function of the program, in which we train the neural network based on the observed experience. It takes a batch of the states as input and updates the policy gradient-based LSTM model weights by minimizing CVaR. This function trains the LSTM model to predict a hedging strategy by looping across the epochs and batches. First, it prepares a batch of market variables (stock price, strike, and risk aversion) and uses `sess.run` function for training. This `sess.run` is a TensorFlow function to run any operation defined within it. Here, it takes the inputs and runs the `tf.train` function that was defined in the constructor. After a sufficient number of iterations, an optimal policy gradient model is generated:

```

def _execute_graph_batchwise(self, paths, strikes, riskaversion, sess, \
    epochs=1, train_flag=False):
    #1: Initialize the variables.
    sample_size = paths.shape[1]
    batch_size=self.batch_size
    idx = np.arange(sample_size)
    start = dt.datetime.now()
    #2:Loop across all the epochs
    for epoch in range(epochs):
        # Save the hedging Pnl for each batch
        pnls = []
        strategies = []
        if train_flag:
            np.random.shuffle(idx)
    #3. Loop across the observations

```

```

for i in range(int(sample_size/batch_size)):
    indices = idx[i*batch_size : (i+1)*batch_size]
    batch = paths[:,indices,:]

    #4. Train the LSTM
    if train_flag:#runs the train, hedging PnL and strategy.
        _, pnl, strategy = sess.run([self.train, self.Hedging_PnL, \
            self.strategy], {self.S_t_input: batch,\ 
            self.K : strikes[indices],\ 
            self.alpha: riskaversion})
    #5. Evaluation and no training
    else:
        pnl, strategy = sess.run([self.Hedging_PnL, self.strategy], \ 
            {self.S_t_input: batch,\ 
            self.K : strikes[indices],\ 
            self.alpha: riskaversion}\

pnls.append(pnl)
strategies.append(strategy)
#6. Calculate the option price # given the risk aversion level alpha

CVaR = np.mean(-np.sort(np.concatenate(pnls))\
[:int((1-riskaversion)*sample_size)])
#7. Return training metrics, |
#if it is in the training phase
if train_flag:
    if epoch % 10 == 0:
        print('Time elapsed:', dt.datetime.now()-start)
        print('Epoch', epoch, 'CVaR', CVaR)
        #Saving the model
        self.saver.save(sess, "model.ckpt")
    self.saver.save(sess, "model.ckpt")

#8. return CVaR and other parameters
return CVaR, np.concatenate(pnls), np.concatenate(strategies, axis=1)

```

The `training` function simply triggers the `execute_graph_batchwise` function and provides all the inputs required for training to this function. The `predict` function returns the action (hedging strategy) given a state (market variables). The `restore` function restores the saved trained model, to be used further for training or prediction:

```

def training(self, paths, strikes, riskaversion, epochs, session, init=True):
    if init:
        sess.run(tf.global_variables_initializer())
    self._execute_graph_batchwise(paths, strikes, riskaversion, session, \
        epochs, train_flag=True)

def predict(self, paths, strikes, riskaversion, session):
    return self._execute_graph_batchwise(paths, strikes, riskaversion, \
        session, 1, train_flag=False)

```

```
def restore(self, session, checkpoint):
    self.saver.restore(session, checkpoint)
```

4.2. Training the data.

The steps of training our policy-based model are:

1. Define the risk aversion parameter for CVaR, number of features (this is total number of stocks, and in this case we just have one), strike price, and batch size. The CVaR represents the amount of loss we want to minimize. For example, a CVaR of 99% means that we want to avoid extreme loss, while a CVaR of 50% minimizes average loss. We train with a CVaR of 50% to have smaller mean loss.
2. Instantiate the policy gradient agent, which has the RNN based-policy with the loss function based on the CVaR.
3. Iterate through the batches; the strategy is defined by the policy output of the LSTM-based network.
4. Finally, the trained model is saved.

```
batch_size = 1000
features = 1
K = 100
alpha = 0.50 #risk aversion parameter for CVaR
epoch = 101 #It is set to 11, but should ideally be a high number
model_1 = Agent(paths_train.shape[0], batch_size, features, name='rnn_final')
# Training the model takes a few minutes
start = dt.datetime.now()
with tf.Session() as sess:
    # Train Model
    model_1.training(paths_train, np.ones(paths_train.shape[1])*K, alpha,
                      epoch, sess)
print('Training finished, Time elapsed:', dt.datetime.now()-start)
```

Output

```
Time elapsed: 0:00:03.326560
Epoch 0 CVaR 4.0718956
Epoch 100 CVaR 2.853285
Training finished, Time elapsed: 0:01:56.299444
```

5. Testing the data

Testing is an important step, especially for RL, as it is difficult for a model to provide any meaningful, intuitive relationships between input and their corresponding output that is easily understood. In the testing step, we will compare the effectiveness of the hedging strategy and compare it to a delta hedging strategy based on the Black-Scholes model. We first define the helper functions used in this step.

5.1. Helper functions for comparison against Black-Scholes. In this module, we create additional functions that are used for comparison against the traditional Black-Scholes model.

5.1.1. Black-Scholes price and delta. The function `BlackScholes_price` implements the analytical formula for the call option price, and `BS_delta` implements the analytical formula for the delta of a call option:

```
def BS_d1(S, dt, r, sigma, K):
    return (np.log(S/K) + (r+sigma**2/2)*dt) / (sigma*np.sqrt(dt))

def BlackScholes_price(S, T, r, sigma, K, t=0):
    dt = T-t
    Phi = stats.norm(loc=0, scale=1).cdf
    d1 = BS_d1(S, dt, r, sigma, K)
    d2 = d1 - sigma*np.sqrt(dt)
    return S*Phi(d1) - K*np.exp(-r*dt)*Phi(d2)

def BS_delta(S, T, r, sigma, K, t=0):
    dt = T-t
    d1 = BS_d1(S, dt, r, sigma, K)
    Phi = stats.norm(loc=0, scale=1).cdf
    return Phi(d1)
```

5.1.2. Test results and plotting. The following functions are used to compute the key metrics and related plots for evaluating the effectiveness of the hedge. The function `test_hedging_strategy` computes different types of PnL, including CVaR, PnL, and Hedge PnL. The function `plot_deltas` plots the comparison of the RL delta versus Black-Scholes hedging at different time points. The function `plot_strategy_pnl` is used to plot the total PnL of the RL-based strategy versus Black-Scholes hedging:

```
def test_hedging_strategy(deltas, paths, K, price, alpha, output=True):
    S_returns = paths[1,:,:]-paths[:-1,:,:]
    hedge_pnl = np.sum(deltas * S_returns, axis=0)
    option_payoff = np.maximum(paths[-1,:,:] - K, 0)
    replication_portfolio_pnls = -option_payoff + hedge_pnl + price
    mean_pnl = np.mean(replication_portfolio_pnls)
    cvar_pnl = -np.mean(np.sort(replication_portfolio_pnls)\n    [:int((1-alpha)*replication_portfolio_pnls.shape[0])])
    if output:
        plt.hist(replication_portfolio_pnls)
        print('BS price at t0:', price)
        print('Mean Hedging PnL:', mean_pnl)
        print('CVaR Hedging PnL:', cvar_pnl)
    return (mean_pnl, cvar_pnl, hedge_pnl, replication_portfolio_pnls, deltas)

def plot_deltas(paths, deltas_bs, deltas_rnn, times=[0, 1, 5, 10, 15, 29]):
    fig = plt.figure(figsize=(10,6))
    for i, t in enumerate(times):
```

```

plt.subplot(2,3,i+1)
xs = paths[t,:,:]
ys_bs = deltas_bs[t,:]
ys_rnn = deltas_rnn[t,:]
df = pd.DataFrame([xs, ys_bs, ys_rnn]).T

plt.plot(df[0], df[1], df[0], df[2], linestyle='', marker='x' )
plt.legend(['BS delta', 'RNN Delta'])
plt.title('Delta at Time %i' % t)
plt.xlabel('Spot')
plt.ylabel('$\Delta$')
plt.tight_layout()

def plot_strategy_pnl(portfolio_pnl_bs, portfolio_pnl_rnn):
    fig = plt.figure(figsize=(10,6))
    sns.boxplot(x=['Black-Scholes', 'RNN-LSTM-v1'], y=[portfolio_pnl_bs, \
    portfolio_pnl_rnn])
    plt.title('Compare PnL Replication Strategy')
    plt.ylabel('PnL')

```

5.1.3. Hedging error for Black-Scholes replication. The following function is used to get the hedging strategy based on the traditional Black-Scholes model, which is further used for comparison against the RL-based hedging strategy:

```

def black_scholes_hedge_strategy(S_0, K, r, vol, T, paths, alpha, output):
    bs_price = BlackScholes_price(S_0, T, r, vol, K, 0)
    times = np.zeros(paths.shape[0])
    times[1:] = T / (paths.shape[0]-1)
    times = np.cumsum(times)
    bs_deltas = np.zeros((paths.shape[0]-1, paths.shape[1]))
    for i in range(paths.shape[0]-1):
        t = times[i]
        bs_deltas[i,:] = BS_delta(paths[i,:,:], T, r, vol, K, t)
    return test_hedging_strategy(bs_deltas, paths, K, bs_price, alpha, output)

```

5.2. Comparison between Black-Scholes and reinforcement learning. We will compare the effectiveness of the hedging strategy by looking at the influence of the CVaR risk aversion parameter and inspect how well the RL-based model can generalize the hedging strategy if we change the moneyness of the option, the drift, and the volatility of the underlying process.

5.2.1. Test at 99% risk aversion. As mentioned before, the CVaR represents the amount of loss we want to minimize. We trained the model using a risk aversion of 50% to minimize average loss. However, for testing purposes we increase the risk aversion to 99%, meaning that we want to avoid extreme loss. These results are compared against the Black-Scholes model:

```

n_sims_test = 1000
# Monte Carlo Path for the test set
alpha = 0.99

```

```
paths_test = monte_carlo_paths(S_0, T, vol, r, seed_test, n_sims_test, \
timesteps)
```

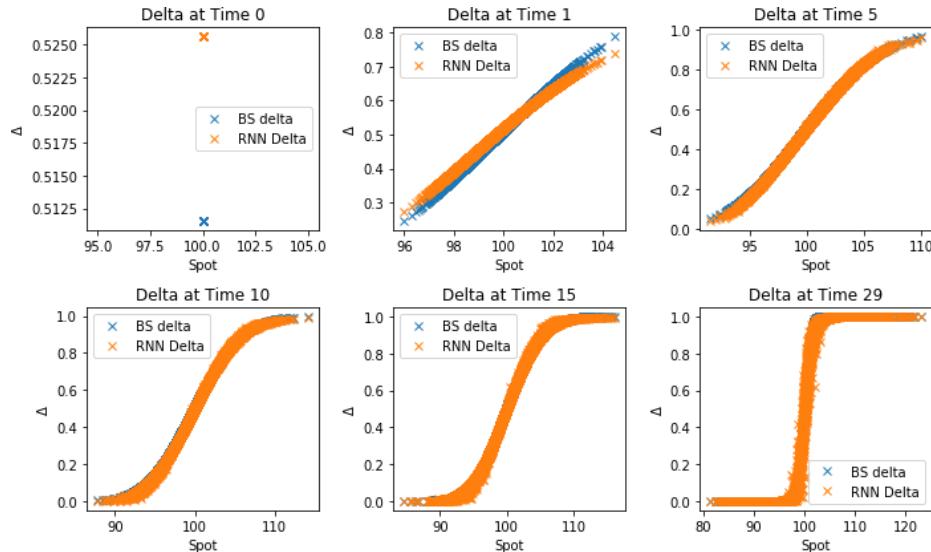
We use the trained function and compare the Black-Scholes and RL models in the following code:

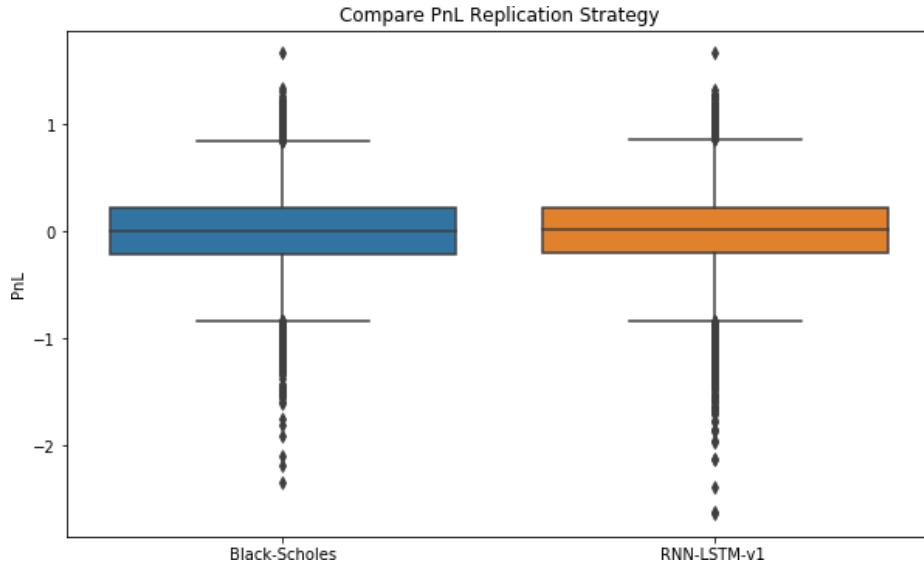
```
with tf.Session() as sess:
    model_1.restore(sess, 'model.ckpt')
    #Using the model_1 trained in the section above
    test1_results = model_1.predict(paths_test, np.ones(paths_test.shape[1])*K, \
alpha, sess)

    _,_,_,portfolio_pnl_bs, deltas_bs = black_scholes_hedge_strategy\
(S_0,K, r, vol, T, paths_test, alpha, True)
    plt.figure()
    _,_,_,portfolio_pnl_rnn, deltas_rnn = test_hedging_strategy\
(test1_results[2], paths_test, K, 2.302974467802428, alpha, True)
    plot_deltas(paths_test, deltas_bs, deltas_rnn)
    plot_strategy_pnl(portfolio_pnl_bs, portfolio_pnl_rnn)
```

Output

```
BS price at t0: 2.3029744678024286
Mean Hedging PnL: -0.0010458505607415178
CVaR Hedging PnL: 1.2447953011695538
RL based BS price at t0: 2.302974467802428
RL based Mean Hedging PnL: -0.0019250998451393934
RL based CVaR Hedging PnL: 1.3832611348053374
```





For the first test set (strike 100, same drift, same vol) with a risk aversion of 99%, the results look quite good. We see that the delta from both Black-Scholes and the RL-based approach converge over time from day 1 to 30. The CVaRs of both strategies are similar and lower in magnitude, with values of 1.24 and 1.38 for Black-Scholes and RL, respectively. Also, the volatility of the two strategies is similar, as illustrated in the second chart.

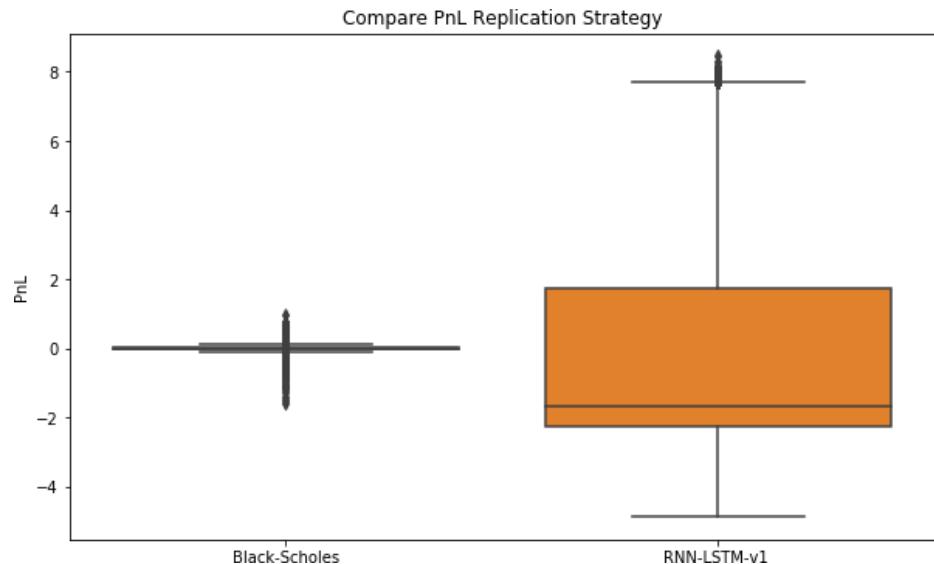
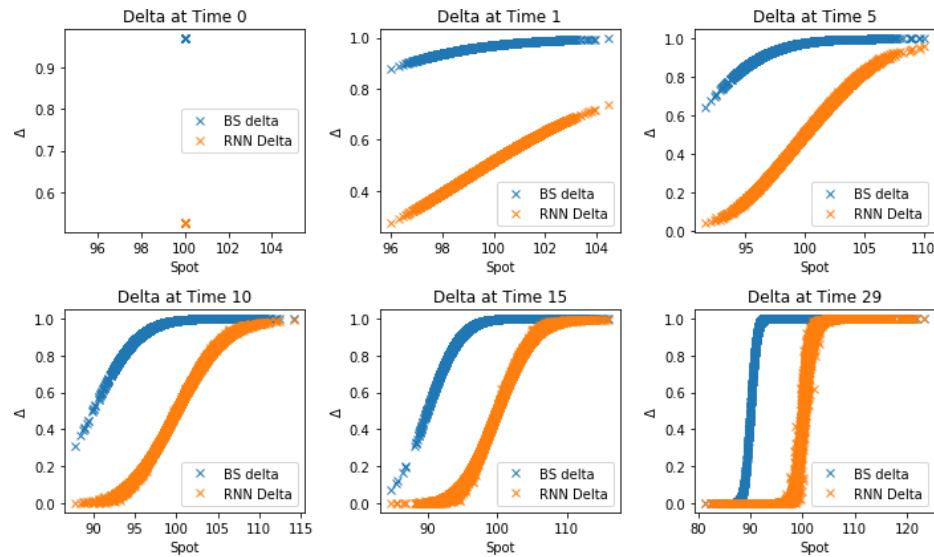
5.2.2. Changing moneyness. Let us now look at the comparison of the strategies, when the moneyness, defined as the ratio of strike to spot price, is changed. In order to change the moneyness, we decrease the strike price by 10. The code snippet is similar to the previous case, and the output is shown below:

```

BS price at t0: 10.07339936955367
Mean Hedging PnL: 0.0007508571761945107
CVaR Hedging PnL: 0.6977526775080665
RL based BS price at t0: 10.073
RL based Mean Hedging PnL: -0.038571546628968216
RL based CVaR Hedging PnL: 3.4732447615593975

```

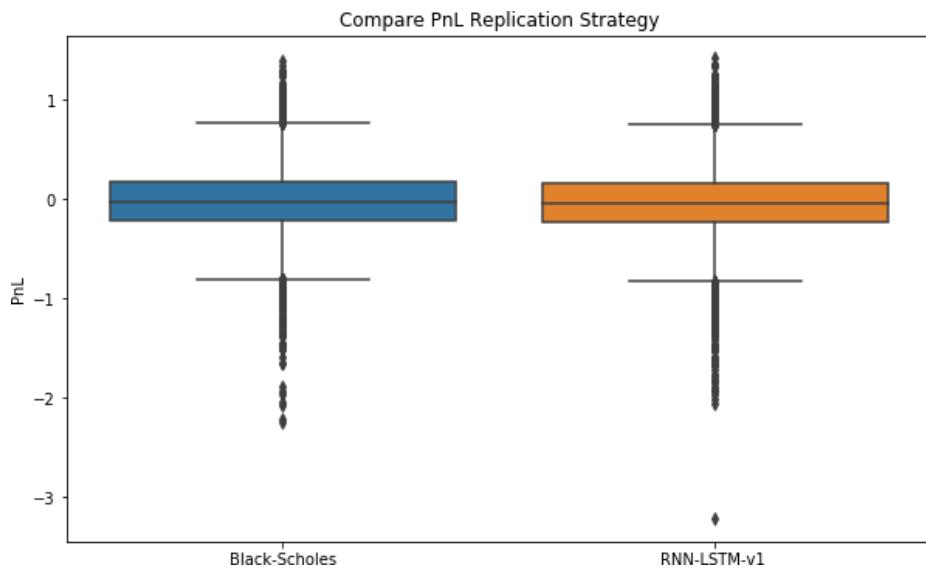
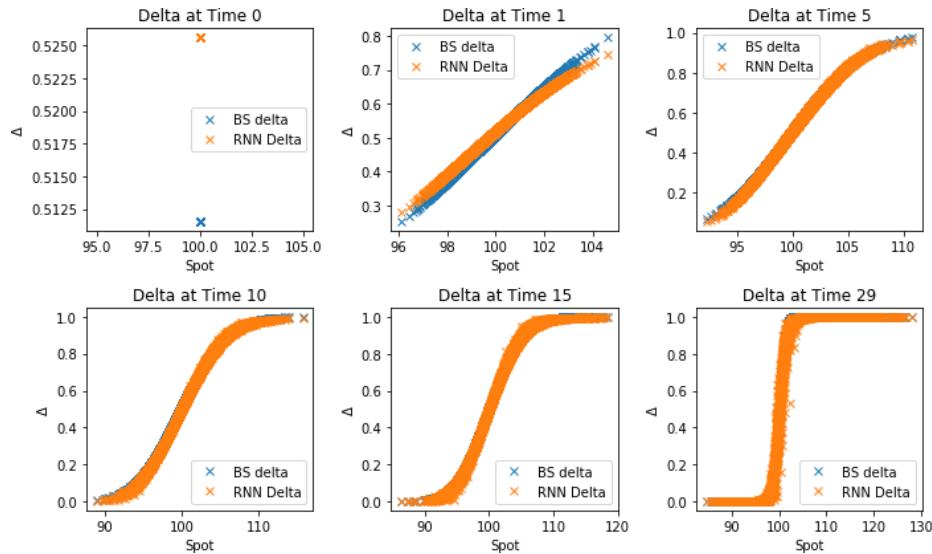
With the change in the moneyness, we see that the PnL of the RL strategy is significantly worse than that of the Black-Scholes strategy. We see a significant deviation of the delta between the two across all the days. The CVaR and the volatility of the RL-based strategy is much higher. The results indicate that we should be careful while generalizing the model to different levels of moneyness and should train the model with the option of using a variety of strikes before implementing it in a production environment.



5.2.3. Changing drift. Let us now look at the comparison of the strategies when the drift is changed. In order to change the drift, we assume the drift of the stock price is 4% per month, or 48% annualized. The output is shown below:

Output

```
BS price at t0: 2.3029744678024286
Mean Hedging PnL: -0.01723902964827388
CVaR Hedging PnL: 1.2141220199385756
RL based BS price at t0: 2.3029
RL based Mean Hedging PnL: -0.037668804359885316
RL based CVaR Hedging PnL: 1.357201635552361
```

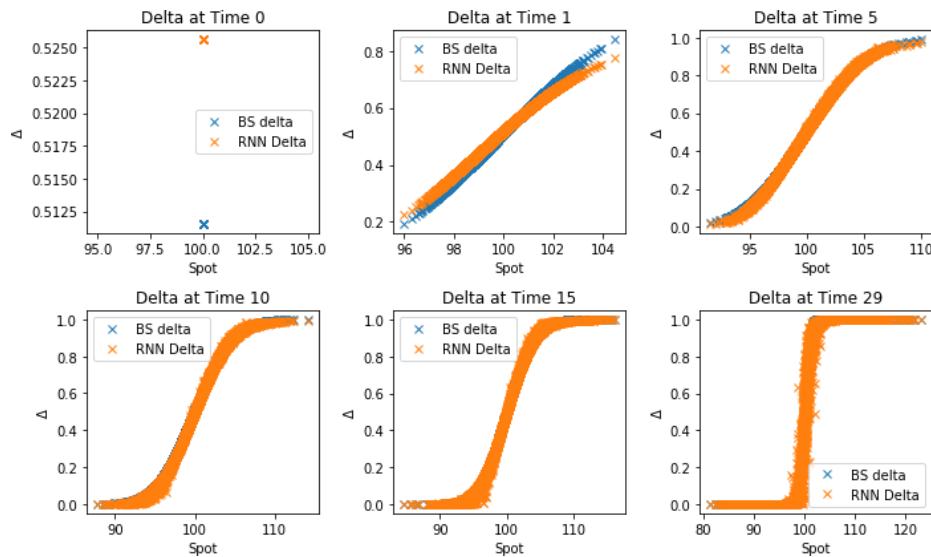


The overall results look good for the change in drift. The conclusion is similar to results when the risk aversion was changed, with the deltas for the two approaches converging over time. Again, the CVaRs are similar in magnitude, with Black-Scholes producing a value of 1.21, and RL a value of 1.357.

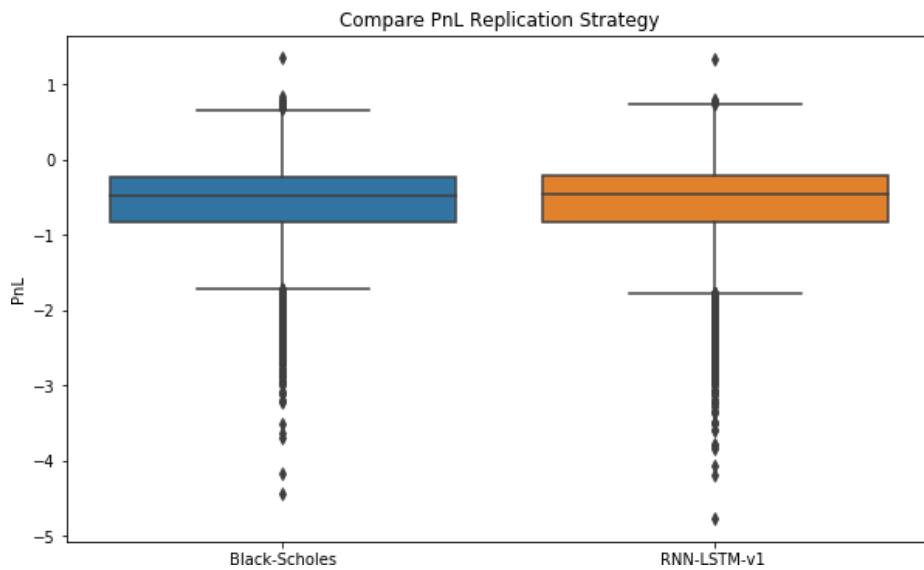
5.2.4. Shifted volatility. Finally, we look at the impact of shifting the volatility. In order to change the volatility, we increase it by 5%:

Output

```
BS price at t0: 2.3029744678024286
Mean Hedging PnL: -0.5787493248269506
CVaR Hedging PnL: 2.5583922824407566
RL based BS price at t0: 2.309
RL based Mean Hedging PnL: -0.5735181045192523
RL based CVaR Hedging PnL: 2.835487824499669
```



Looking at the results, the delta, CVaR, and overall volatility of both models are similar. Hence looking at the different comparisons overall, the performance of this RL-based hedging is on par with Black-Scholes-based hedging.



Conclusion

In this case study, we compared the effectiveness of a call option hedging strategy using RL. The RL-based hedging strategy did quite well even when certain input parameters were modified. However, this strategy was not able to generalize the strategy for options at different moneyness levels. It underscores the fact that RL is a data-intensive approach, and it is important to train the model with different scenarios, which becomes more important if the model is intended to be used across a wide variety of derivatives.

Although we found the RL and traditional Black-Scholes strategies comparable, the RL approach offers a much higher ceiling for improvement. The RL model can be further trained using a wide variety of instruments with different hyperparameters, leading to performance enhancements. It would be interesting to explore the comparison of these two hedging models for more exotic derivatives, given the trade-off between these approaches.

Overall, the RL-based approach is model independent and scalable, and it offers efficiency boosts for many classical problems.

Case Study 3: Portfolio Allocation

As discussed in prior case studies, the most commonly used technique for portfolio allocation, *mean-variance portfolio optimization*, suffers from several weaknesses, including:

- Estimation errors in the expected returns and covariance matrix caused by the erratic nature of financial returns.
- Unstable quadratic optimization that greatly jeopardizes the optimality of the resulting portfolios.

We addressed some of these weaknesses in “[Case Study 1: Portfolio Management: Finding an Eigen Portfolio](#)” on page 202 in [Chapter 7](#), and in “[Case Study 3: Hierarchical Risk Parity](#)” on page 267 in [Chapter 8](#). Here, we approach this problem from an RL perspective.

Reinforcement learning algorithms, with the ability to decide the policy on their own, are strong models for performing portfolio allocation in an automated manner, without the need for continuous supervision. Automation of the manual steps involved in portfolio allocation can prove to be immensely useful, specifically for robo-advisors.

In an RL-based framework, we treat portfolio allocation not just as a one-step optimization problem but as *continuous control* of the portfolio with delayed rewards. We move from discrete optimal allocation to continuous control territory, and in the environment of a continuously changing market, RL algorithms can be leveraged to solve complex and dynamic portfolio allocation problems.

In this case study, we will use a Q-learning-based approach and DQN to come up with a policy for optimal portfolio allocation among a set of cryptocurrencies. Overall, the approach and framework in terms of the Python-based implementation is similar to that in case study 1. Therefore, some repetitive sections or code explanation is skipped in this case study.

In this case study, we will focus on:

- Defining the components of RL in a portfolio allocation problem.
- Evaluating Q-learning in the context of portfolio allocation.
- Creating a simulation environment to be used in the RL framework.
- Extending the Q-learning framework used for trading strategy development to portfolio management.



Blueprint for Creating a Reinforcement Learning–Based Algorithm for Portfolio Allocation

1. Problem definition

In the reinforcement learning framework defined for this case study, the algorithm performs an action, which is *optimal portfolio allocation*, depending on the current state of the portfolio. The algorithm is trained using a deep Q-learning framework, and the components of the model are as follows:

Agent

A portfolio manager, a robo-advisor, or an individual investor.

Action

Assignment and rebalancing of the portfolio weights. The DQN model provides the Q-values, which are converted into portfolio weights.

Reward function

The Sharpe ratio. Although there can be a wide range of complex reward functions that provide a trade-off between profit and risk, such as percentage return or maximum drawdown.

State

The state is the correlation matrix of the instruments based on a specific time window. The correlation matrix is a suitable state variable for the portfolio allocation, as it contains the information about the relationships between different instruments and can be useful in performing portfolio allocation.

Environment

The cryptocurrency exchange.

The dataset used in this case study is from the [Kaggle](#) platform. It contains the daily prices of cryptocurrencies in 2018. The data contains some of the most liquid cryptocurrencies, including Bitcoin, Ethereum, Ripple, Litecoin, and Dash.

2. Getting started—loading the data and Python packages

2.1. Loading the Python packages. The standard Python packages are loaded in this step. The details have already been presented in the previous case studies. Refer to the Jupyter notebook for this case study for more details.

2.2. Loading the data. The fetched data is loaded in this step:

```
dataset = read_csv('data/crypto_portfolio.csv',index_col=0)
```

3. Exploratory data analysis

3.1. Descriptive statistics. We will look at descriptive statistics and data visualizations of the data in this section:

```
# shape  
dataset.shape
```

Output

```
(375, 15)
```

```
# peek at data  
set_option('display.width', 100)  
dataset.head(5)
```

Output

Date	ADA	BCH	BNB	BTC	DASH	EOS	ETH	IOT	LINK	LTC	TRX	USDT	XLM	XMR	XRP
2018-01-01	0.7022	2319.120117	8.480	13444.879883	1019.419983	7.64	756.200012	3.90	0.7199	224.339996	0.05078	1.01	0.4840	338.170013	2.05
2018-01-02	0.7620	2555.489990	8.749	14754.129883	1162.469971	8.30	861.969971	3.98	0.6650	251.809998	0.07834	1.02	0.5560	364.440002	2.19
2018-01-03	1.1000	2557.520020	9.488	15156.620117	1129.890015	9.43	941.099976	4.13	0.6790	244.630005	0.09430	1.01	0.8848	385.820007	2.73
2018-01-04	1.1300	2355.780029	9.143	15180.080078	1120.119995	9.47	944.830017	4.10	0.9694	238.300003	0.21010	1.02	0.6950	372.230011	2.73
2018-01-05	1.0100	2390.040039	14.850	16954.779297	1080.880005	9.29	967.130005	3.76	0.9669	244.509995	0.22400	1.01	0.6400	357.299988	2.51

The data has a total of 375 rows and 15 columns. These columns hold the daily prices of 15 different cryptocurrencies in 2018.

4. Evaluate algorithms and models

This is the key step of the reinforcement learning model development, where we will define all the functions and classes and train the algorithm.

4.1. Agent and cryptocurrency environment script. We have an Agent class that holds the variables and member functions that perform the Q-learning. This is similar to the Agent class defined in case study 1, with an additional function to convert the Q-value output from the deep neural network to portfolio weights and vice versa. The training module implements iteration through several episodes and batches and saves the information of the state, action, reward, and next state to be used in training. We skip the detailed description of the Python code of Agent class and the training module in this case study. Readers can refer to the Jupyter notebook in the code repository for this book for more details.

We implement a simulation environment for cryptocurrencies using a class called `CryptoEnvironment`. The concept of a simulation environment, or *gym*, is quite common in RL problems. One of the challenges of reinforcement learning is the lack of available simulation environments on which to experiment. *OpenAI gym* is a toolkit that provides a wide variety of simulated environments (e.g., Atari games, 2D/3D physical simulations), so we can train agents, compare them, or develop new RL algorithms. Additionally, it was developed with the aim of becoming a standardized environment and benchmark for RL research. We introduce a similar concept in the `CryptoEnvironment` class, where we create a simulation environment for cryptocurrencies. This class has the following key functions:

`getState`

This function returns the state as well as the historical return or raw historical data depending on the `is_cov_matrix` or `is_raw_time_series` flag

`getReward`

This function returns the reward (i.e., Sharpe ratio) of the portfolio, given the portfolio weights and lookback period

```
class CryptoEnvironment:

    def __init__(self, prices = './data/crypto_portfolio.csv', capital = 1e6):
        self.prices = prices
        self.capital = capital
        self.data = self.load_data()

    def load_data(self):
        data = pd.read_csv(self.prices)
        try:
            data.index = data['Date']
            data = data.drop(columns = ['Date'])
        except:
            data.index = data['date']
            data = data.drop(columns = ['date'])
        return data

    def preprocess_state(self, state):
        return state

    def get_state(self, t, lookback, is_cov_matrix=True\
                is_raw_time_series=False):

        assert lookback <= t

        decision_making_state = self.data.iloc[t-lookback:t]
        decision_making_state = decision_making_state.pct_change().dropna()

        if is_cov_matrix:
            x = decision_making_state.cov()
```

```

        return x
    else:
        if is_raw_time_series:
            decision_making_state = self.data.iloc[t-lookback:t]
        return self.preprocess_state(decision_making_state)

def get_reward(self, action, action_t, reward_t, alpha = 0.01):

    def local_portfolio(returns, weights):
        weights = np.array(weights)
        rets = returns.mean() # * 252
        covs = returns.cov() # * 252
        P_ret = np.sum(rets * weights)
        P_vol = np.sqrt(np.dot(weights.T, np.dot(covs, weights)))
        P_sharpe = P_ret / P_vol
        return np.array([P_ret, P_vol, P_sharpe])

    data_period = self.data[action_t:reward_t]
    weights = action
    returns = data_period.pct_change().dropna()

    sharpe = local_portfolio(returns, weights)[-1]
    sharpe = np.array([sharpe] * len(self.data.columns))
    ret = (data_period.values[-1] - data_period.values[0]) / \
    data_period.values[0]

    return np.dot(returns, weights), ret

```

Let's explore the training of the RL model in the next step.

4.3. Training the data. As a first step, we initialize the `Agent` class and `CryptoEnvironment` class. Then, we set the number of episodes and batch size for the training purpose. Given the volatility of cryptocurrencies, we set the state window size to 180 and rebalancing frequency to 90 days:

```

N_ASSETS = 15
agent = Agent(N_ASSETS)
env = CryptoEnvironment()
window_size = 180
episode_count = 50
batch_size = 32
rebalance_period = 90

```

Figure 9-10 provides a deep dive into the training of the DQN algorithm used for developing the RL-based portfolio allocation strategy. If we look carefully, the chart is similar to the steps defined in Figure 9-8 in case study 1, with minor differences in the `Q-Matrix`, `reward function`, and `action`. Steps 1 to 7 describe the training and `CryptoEnvironment` module; steps 8 to 10 show what happens in the `replay buffer` function (i.e., `exeReplay` function) in the `Agent` module.

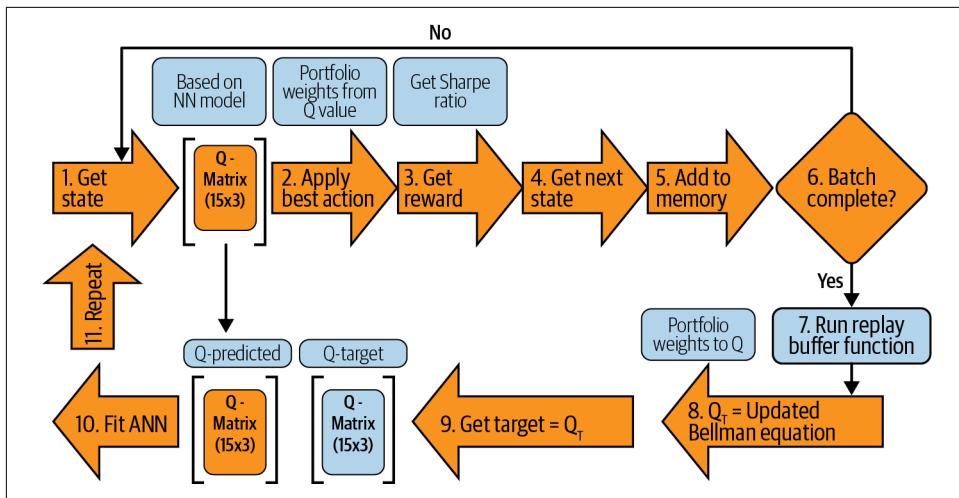


Figure 9-10. DQN training for portfolio optimization

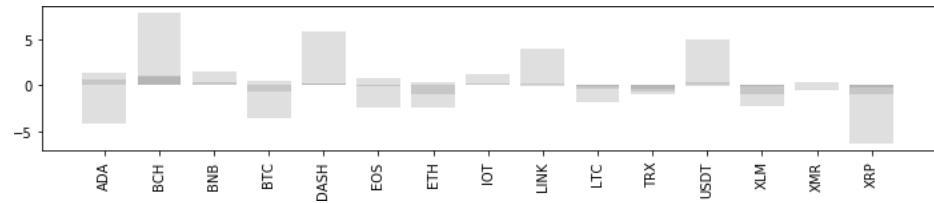
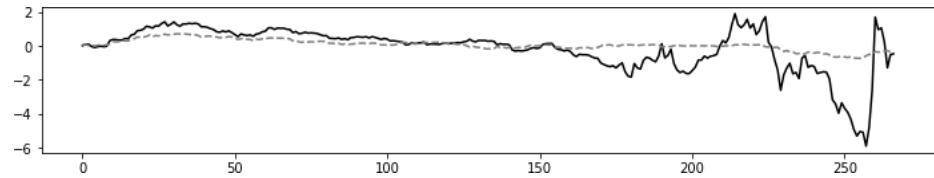
The details of steps 1 to 6 are:

1. Get the *current state* using the helper function `getState` defined in the `CryptoEnvironment` module. It returns a correlation matrix of the cryptocurrencies based on the window size.
2. Get the *action* for the given state using the `act` function of the `Agent` class. The action is the weight of the cryptocurrency portfolio.
3. Get the *reward* for the given action using the `getReward` function in the `CryptoEnvironment` module.
4. Get the next state using the `getState` function. The detail of the next state is further used in the Bellman equation for updating the Q-function.
5. The details of the state, next state, and action are saved in the memory of the `Agent` object. This memory is used further by the `exeReply` function.
6. Check if the batch is complete. The size of a batch is defined by the batch size variable. If the batch is not complete, we move to the next time iteration. If the batch is complete, then we move to the `Replay buffer` function and update the Q-function by minimizing the MSE between the `Q-predicted` and the `Q-target` in steps 8, 9, and 10.

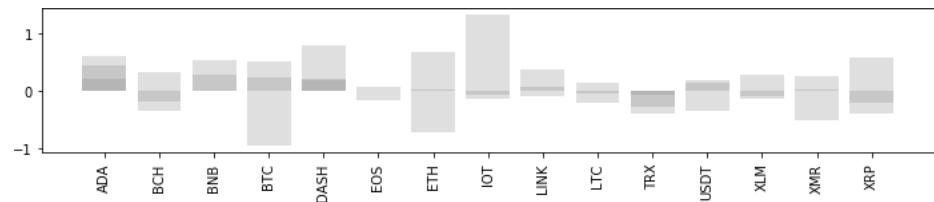
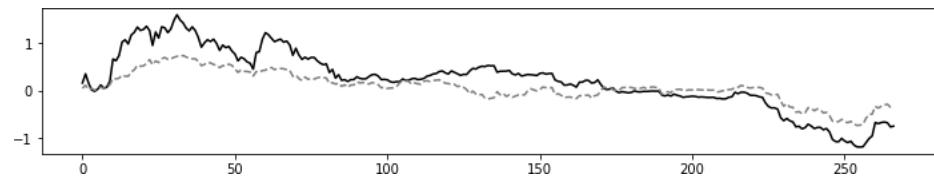
As shown in the following charts, the code produces the final results along with two charts for each episode. The first chart shows the total cumulative return over time, while the second chart shows the percentage of each cryptocurrency in the portfolio.

Output

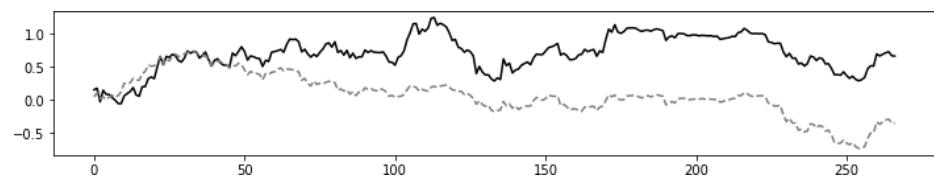
Episode 0/50 epsilon 1.0

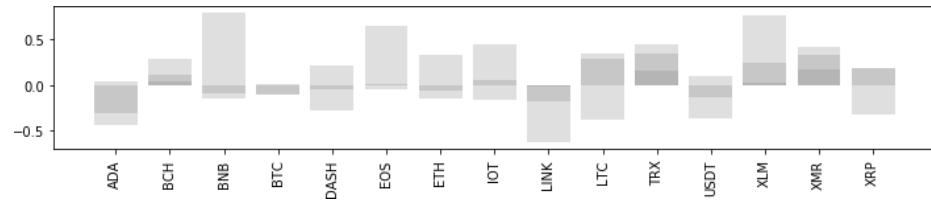


Episode 1/50 epsilon 1.0

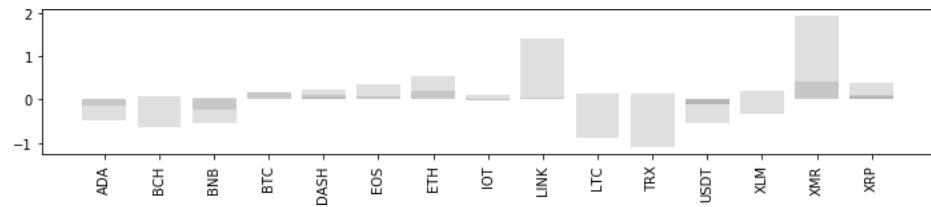
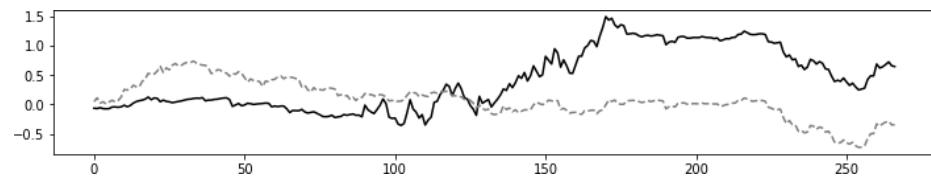


Episode 48/50 epsilon 1.0





Episode 49/50 epsilon 1.0



The charts outline the details of the portfolio allocation of the first two and last two episodes. The details of other episodes can be seen in the Jupyter notebook under the GitHub repository for this book. The black line shows the performance of the portfolio, and the dotted grey line shows the performance of the benchmark, which is an equally weighted portfolio of cryptocurrencies.

In the beginning of episodes zero and one, the agent has no preconception of the consequences of its actions, and it takes randomized actions to observe the returns, which are quite volatile. Episode zero shows a clear example of erratic performance behavior. Episode one displays more stable movement but ultimately underperforms the benchmark. This is evidence that the cumulative reward per episode fluctuates significantly in the beginning of training.

The last two charts of episodes 48 and 49 show the agent starting to learn from its training and discovering the optimal strategy. Overall returns are relatively stable and outperform the benchmark. However, the overall portfolio weights are still quite volatile due to the short time series and high volatility of the underlying cryptocurrency assets. Ideally, we would be able to increase the number of training episodes and the length of historical data to enhance the training performance.

Let us look at the testing results.

5. Testing the data

Recall that the black line shows the performance of the portfolio, and the dotted grey line is that of an equally weighted portfolio of cryptocurrencies:

```
agent.is_eval = True

actions_equal, actions_rl = [], []
result_equal, result_rl = [], []

for t in range(window_size, len(env.data), rebalance_period):

    date1 = t-rebalance_period
    s_ = env.get_state(t, window_size)
    action = agent.act(s_)

    weighted_returns, reward = env.get_reward(action[0], date1, t)
    weighted_returns_equal, reward_equal = env.get_reward(
        np.ones(agent.portfolio_size) / agent.portfolio_size, date1, t)

    result_equal.append(weighted_returns_equal.tolist())
    actions_equal.append(np.ones(agent.portfolio_size) / agent.portfolio_size)

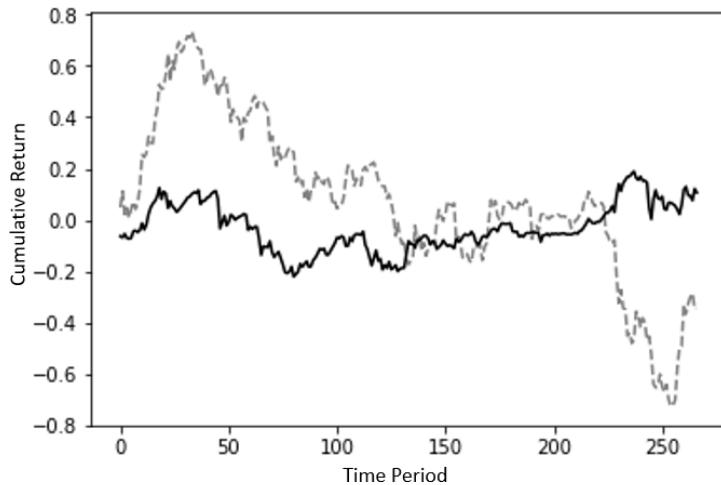
    result_rl.append(weighted_returns.tolist())
    actions_rl.append(action[0])

result_equal_vis = [item for sublist in result_equal for item in sublist]
result_rl_vis = [item for sublist in result_rl for item in sublist]

plt.figure()
plt.plot(np.array(result_equal_vis).cumsum(), label = 'Benchmark', \
color = 'grey',ls = '--')
plt.plot(np.array(result_rl_vis).cumsum(), label = 'Deep RL portfolio', \
color = 'black',ls = '-')
plt.xlabel('Time Period')
plt.ylabel('Cumulative Returnimage::images\Chapter9-b82b2.png[]')
plt.show()
```

Despite underperforming during the initial period, the model performance was better overall, primarily due to avoiding the steep decline that the benchmark portfolio experienced in the latter part of the test window. The returns appear very stable, perhaps due to rotating away from the most volatile cryptocurrencies.

Output



Let us inspect the return, volatility, Sharpe ratio, alpha, and beta of the portfolio and benchmark:

```
import statsmodels.api as sm
from statsmodels import regression
def sharpe(R):
    r = np.diff(R)
    sr = r.mean()/r.std() * np.sqrt(252)
    return sr

def print_stats(result, benchmark):
    sharpe_ratio = sharpe(np.array(result).cumsum())
    returns = np.mean(np.array(result))
    volatility = np.std(np.array(result))

    X = benchmark
    y = result
    x = sm.add_constant(X)
    model = regression.linear_model.OLS(y, x).fit()
    alpha = model.params[0]
    beta = model.params[1]

    return np.round(np.array([returns, volatility, sharpe_ratio,
                           alpha, beta]), 4).tolist()

print('EQUAL', print_stats(result_equal_vis, result_equal_vis))
print('RL AGENT', print_stats(result_rl_vis, result_equal_vis))
```

Output

```
EQUAL [-0.0013, 0.0468, -0.5016, 0.0, 1.0]  
RL AGENT [0.0004, 0.0231, 0.4445, 0.0002, -0.1202]
```

Overall, the RL portfolio performs better across the board, with a higher return, higher Sharpe ratio, lower volatility, slight alpha, and negative correlation to the benchmark.

Conclusion

In this case study, we went beyond the classic efficient frontier for portfolio optimization and directly learned a policy of dynamically changing portfolio weights. We trained an RL-based model by setting up a standardized simulation environment. This approach facilitated the training process and can be explored further for general RL-based model training.

The trained RL-based model outperformed an equal-weight benchmark in the test set. The performance of the RL-based model can be further improved by optimizing the hyperparameters or using a longer time series for training. However, given the high complexity and low interpretability of an RL-based model, testing should occur across different time periods and market cycles before deploying the model for live trading. Also, as discussed in case study 1, we should carefully select the RL components, such as the reward function and state, and ensure we understand their impact on the overall model results.

The framework provided in this case study can enable financial practitioners to perform portfolio allocation and rebalancing with a very flexible and automated approach.

Chapter Summary

Reward maximization is one of the key principles that drives algorithmic trading, portfolio management, derivative pricing, hedging, and trade execution. In this chapter, we saw that when we use RL-based approaches, explicitly defining the strategy or policy for trading, derivative hedging, or portfolio management is unnecessary. The algorithm determines the policy itself, which can lead to a much simpler and more principled approach than other machine learning techniques.

In “[Case Study 1: Reinforcement Learning-Based Trading Strategy](#)” on page 298, we saw that RL makes algorithmic trading a simple game, which may or may not involve understanding fundamental information. In “[Case Study 2: Derivatives Hedging](#)” on page 316, we explored the use of reinforcement learning for a traditional derivative hedging problem. This exercise demonstrated that we can leverage the efficient

numerical calculation of RL in derivatives hedging to address some of the drawbacks of the more traditional models. In “[Case Study 3: Portfolio Allocation](#)” on page 334, we performed portfolio allocation by learning a policy of changing portfolio weights dynamically in a continuously changing market environment, leading to further automation of the portfolio management process.

Although RL comes with some challenges, such as being computationally expensive and data intensive and lacking interpretability, it aligns perfectly with some areas in finance that are suited for policy frameworks based on reward maximization. Reinforcement learning has managed to achieve superhuman performance in finite action spaces, such as those in the games of Go, chess, and Atari. Looking ahead, with the availability of more data, refined RL algorithms, and superior infrastructure, RL will continue to prove to be immensely useful in finance.

Exercises

- Using the ideas and concepts presented in case studies 1 and 2, implement a trading strategy based on a policy gradient algorithm for FX. Vary the key components (i.e., reward function, state, etc.) for this implementation.
- Implement the hedging of a fixed income derivative using the concepts presented in case study 2.
- Incorporate a transaction cost in case study 2 and see the impact on the overall results.
- Based on the ideas presented in case study 3, implement a Q-learning-based portfolio allocation strategy on a portfolio of stocks, FX, or fixed income instruments.

CHAPTER 10

Natural Language Processing

Natural language processing (NLP) is a subfield of artificial intelligence used to aid computers in understanding natural human language. Most NLP techniques rely on machine learning to derive meaning from human languages. When text has been provided, the computer utilizes algorithms to extract meaning associated with every sentence and collect essential data from them. NLP manifests itself in different forms across many disciplines under various aliases, including (but not limited to) textual analysis, text mining, computational linguistics, and content analysis.

In the financial landscape, one of the earliest applications of NLP was implemented by the US Securities and Exchange Commission (SEC). The group used text mining and natural language processing to detect accounting fraud. The ability of NLP algorithms to scan and analyze legal and other documents at a high speed provides banks and other financial institutions with enormous efficiency gains to help them meet compliance regulations and combat fraud.

In the investment process, uncovering investment insights requires not only domain knowledge of finance but also a strong grasp of data science and machine learning principles. NLP tools may help detect, measure, predict, and anticipate important market characteristics and indicators, such as market volatility, liquidity risks, financial stress, housing prices, and unemployment.

News has always been a key factor in investment decisions. It is well established that company-specific, macroeconomic, and political news strongly influence the financial markets. As technology advances, and market participants become more connected, the volume and frequency of news will continue to grow rapidly. Even today, the volume of daily text data being produced presents an untenable task for even a large team of fundamental researchers to navigate. Fundamental analysis assisted by NLP techniques is now critical to unlock the complete picture of how experts and the masses feel about the market.

In banks and other organizations, teams of analysts are dedicated to poring over, analyzing, and attempting to quantify qualitative data from news and SEC-mandated reporting. Automation using NLP is well suited in this context. NLP can provide in-depth support in the analysis and interpretation of various reports and documents. This reduces the strain that repetitive, low-value tasks put on human employees. It also provides a level of objectivity and consistency to otherwise subjective interpretations; mistakes from human error are lessened. NLP can also allow a company to garner insights that can be used to assess a creditor's risk or gauge brand-related sentiment from content across the web.

With the rise in popularity of live chat software in banking and finance businesses, NLP-based chatbots are a natural evolution. The combination of robo-advisors with chatbots is expected to automate the entire process of wealth and portfolio management.

In this chapter, we present three NLP-based case studies that cover applications of NLP in algorithmic trading, chatbot creation, and document interpretation and automation. The case studies follow a standardized seven-step model development process presented in [Chapter 2](#). Key model steps for NLP-based problems are data preprocessing, feature representation, and inference. As such, these areas, along with the related concepts and Python-based examples, are outlined in this chapter.

[“Case Study 1: NLP and Sentiment Analysis–Based Trading Strategies” on page 362](#) demonstrates the usage of sentiment analysis and word embedding for a trading strategy. This case study highlights key focus areas for implementing an NLP-based trading strategy.

In [“Case Study 2: Chatbot Digital Assistant” on page 383](#), we create a chatbot and demonstrate how NLP enables chatbots to understand messages and respond appropriately. We leverage Python-based packages and modules to develop a chatbot in a few lines of code.

[“Case Study 3: Document Summarization” on page 393](#) illustrates the use of an NLP-based *topic modeling* technique to discover hidden topics or themes across documents. The purpose of this case study is to demonstrate the usage of NLP to automatically summarize large collections of documents to facilitate organization and management, as well as search and recommendations.

In addition to the points mentioned above, this chapter will cover:

- How to perform NLP data preprocessing, including steps such as tokenization, part-of-speech (PoS) tagging, or named entity recognition, in a few lines of code.
- How to use different supervised techniques, including LSTM, for sentiment analysis.

- Understanding the main Python packages (i.e., NLTK, spaCy and TextBlob) and how to use them for several NLP-related tasks.
- How to build a data preprocessing pipeline using the spaCy package.
- How to use pretrained models, such as word2vec, for feature representation.
- How to use models such as LDA for topic modeling.



This Chapter's Code Repository

The Python code for this chapter is included under the [Chapter 10 - Natural Language Processing](#) folder of the online GitHub repository for this chapter. For any new NLP-based case study, use the common template from the code repository and modify the elements specific to the case study. The templates are designed to run on the cloud (i.e., Kaggle, Google Colab, and AWS).

Natural Language Processing: Python Packages

Python is one of the best options to build an NLP-based expert system, and a large variety of open source NLP libraries are available for Python programmers. These libraries and packages contain ready-to-use modules and functions to incorporate complex NLP steps and algorithms, making implementation fast, easy, and efficient.

In this section, we will describe three Python-based NLP libraries we've found to be the most useful and that we will be using in this chapter.

NLTK

[NLTK](#) is the most famous Python NLP library, and it has led to incredible breakthroughs across several areas. Its modularized structure makes it excellent for learning and exploring NLP concepts. However, it has heavy functionality with a steep learning curve.

NLTK can be installed using the typical installation procedure. After installing NLTK, NLTK Data needs to be downloaded. The NLTK Data package includes a pre-trained tokenizer punkt for English, which can be downloaded as well:

```
import nltk
import nltk.data
nltk.download('punkt')
```

TextBlob

[TextBlob](#) is built on top of NLTK. This is one of the best libraries for fast prototyping or building applications with minimal performance requirements. TextBlob makes

text processing simple by providing an intuitive interface to NLTK. TextBlob can be imported using the following command:

```
from textblob import TextBlob
```

spaCy

spaCy is an NLP library designed to be fast, streamlined, and production-ready. Its philosophy is to present only one algorithm (the best one) for each purpose. We don't have to make choices and can focus on being productive. spaCy uses its own pipeline to perform multiple preprocessing steps at the same time. We will demonstrate it in a subsequent section.

spaCy's models can be installed as Python packages, just like any other module. To load a model, use `spacy.load` with the model's shortcut link or package name or a path to the data directory:

```
import spacy  
nlp = spacy.load("en_core_web_lg")
```

In addition to these, there are a few other libraries, such as gensim, that we will explore for some of the examples in this chapter.

Natural Language Processing: Theory and Concepts

As we have already established, NLP is a subfield of artificial intelligence concerned with programming computers to process textual data in order to gain useful insights. All NLP applications go through common sequential steps, which include some combination of preprocessing textual data and representing the text as predictive features before feeding them into a statistical inference algorithm. [Figure 10-1](#) outlines the major steps in an NLP-based application.

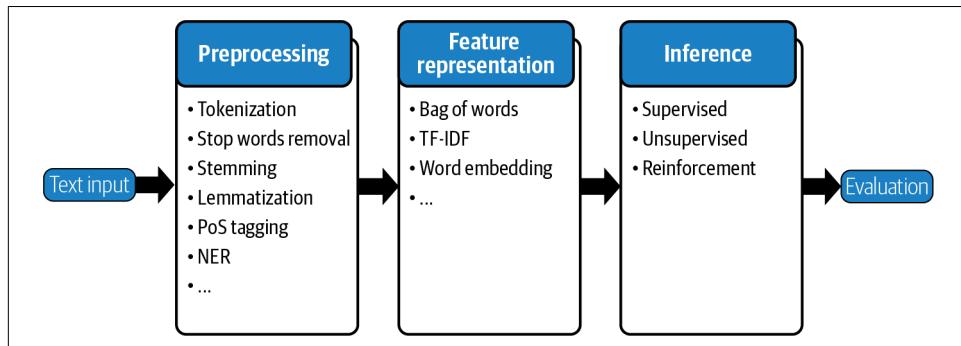


Figure 10-1. Natural language processing pipeline

The next section reviews these steps. For a thorough coverage of the topic, the reader is referred to *Natural Language Processing with Python* by Steven Bird, Ewan Klein, and Edward Loper (O'Reilly).

1. Preprocessing

There are usually multiple steps involved in preprocessing textual data for NLP. **Figure 10-1** shows the key components of the preprocessing steps for NLP. These are tokenization, stop words removal, stemming, lemmatization, PoS (part-of-speech) tagging, and NER (Name Entity Recognition).

1.1. Tokenization

Tokenization is the task of splitting a text into meaningful segments, called tokens. These segments could be words, punctuation, numbers, or other special characters that are the building blocks of a sentence. A set of predetermined rules allows us to effectively convert a sentence into a list of tokens. The following code snippets show sample word tokenization using the NLTK and TextBlob packages:

```
#Text to tokenize
text = "This is a tokenize test"
```

The NLTK data package includes a pretrained Punkt tokenizer for English, which was previously loaded:

```
from nltk.tokenize import word_tokenize
word_tokenize(text)
```

Output

```
['This', 'is', 'a', 'tokenize', 'test']
```

Let's look at tokenization using TextBlob:

```
TextBlob(text).words
```

Output

```
WordList(['This', 'is', 'a', 'tokenize', 'test'])
```

1.2. Stop words removal

At times, extremely common words that offer little value in modeling are excluded from the vocabulary. These words are called stop words. The code for removing stop words using the NLTK library is shown below:

```
text = "S&P and NASDAQ are the two most popular indices in US"

from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
stop_words = set(stopwords.words('english'))
text_tokens = word_tokenize(text)
```

```
tokens_without_sw= [word for word in text_tokens if not word in stop_words]

print(tokens_without_sw)
```

Output

```
['S', '&', 'P', 'NASDAQ', 'two', 'popular', 'indices', 'US']
```

We first load the language model and store it in the stop words variable. The `stop_words.words('english')` is a set of default stop words for the English language model in NLTK. Next, we simply iterate through each word in the input text, and if the word exists in the stop word set of the NLTK language model, the word is removed. As we can see, stop words, such as *are* and *most*, are removed from the sentence.

1.3. Stemming

Stemming is the process of reducing inflected (or sometimes derived) words to their stem, base, or root form (generally a written word form). For example, if we were to stem the words *Stems*, *Stemming*, *Stemmed*, and *Stemitzization*, the result would be a single word: *Stem*. The code for stemming using the NLTK library is shown here:

```
text = "It's a Stemming testing"

parsed_text = word_tokenize(text)

# Initialize stemmer.
from nltk.stem.snowball import SnowballStemmer
stemmer = SnowballStemmer('english')

# Stem each word.
[(word, stemmer.stem(word)) for i, word in enumerate(parsed_text)
 if word.lower() != stemmer.stem(parsed_text[i])]
```

Output

```
[('Stemming', 'stem'), ('testing', 'test')]
```

1.4. Lemmatization

A slight variant of stemming is *lemmatization*. The major difference between the two processes is that stemming can often create nonexistent words, whereas lemmas are actual words. An example of lemmatization is *run* as a base form for words like *running* and *ran*, or that the words *better* and *good* are considered the same lemma. The code for lemmatization using the TextBlob library is shown below:

```
text = "This world has a lot of faces "

from textblob import Word
parsed_data= TextBlob(text).words
```

```
[('word', word.lemmatize()) for i, word in enumerate(parsed_data)
 if word != parsed_data[i].lemmatize())]
```

Output

```
[('has', 'ha'), ('faces', 'face')]
```

1.5. PoS tagging

Part-of-speech (PoS) tagging is the process of assigning a token to its grammatical category (e.g., verb, noun, etc.) in order to understand its role within a sentence. PoS tags have been used for a variety of NLP tasks and are extremely useful since they provide a linguistic signal of how a word is being used within the scope of a phrase, sentence, or document.

After a sentence is split into tokens, a tagger, or PoS tagger, is used to assign each token to a part-of-speech category. Historically, **hidden Markov models (HMM)** were used to create such taggers. More recently, artificial neural networks have been leveraged. The code for PoS tagging using the TextBlob library is shown here:

```
text = 'Google is looking at buying U.K. startup for $1 billion'
TextBlob(text).tags
```

Output

```
[('Google', 'NNP'),
 ('is', 'VBZ'),
 ('looking', 'VBG'),
 ('at', 'IN'),
 ('buying', 'VBG'),
 ('U.K.', 'NNP'),
 ('startup', 'NN'),
 ('for', 'IN'),
 ('1', 'CD'),
 ('billion', 'CD')]
```

1.6. Named entity recognition

Named entity recognition (NER) is an optional next step in data preprocessing that seeks to locate and classify named entities in text into predefined categories. These categories can include names of persons, organizations, locations, expressions of times, quantities, monetary values, or percentages. The NER performed using spaCy is shown below:

```
text = 'Google is looking at buying U.K. startup for $1 billion'

for entity in nlp(text).ents:
    print("Entity: ", entity.text)
```

Output

```
Entity: Google  
Entity: U.K.  
Entity: $1 billion
```

Visualizing named entities in text using the `displacy` module, as shown in [Figure 10-2](#), can also be incredibly helpful in speeding up development and debugging the code and training process:

```
from spacy import displacy  
displacy.render(nlp(text), style="ent", jupyter = True)
```



Google ORG is looking at buying U.K. GPE startup for \$1 billion MONEY

Figure 10-2. NER output

1.7. spaCy: All of the above steps in one go. All the preprocessing steps shown above can be performed in one step using spaCy. When we call `nlp` on a text, spaCy first tokenizes the text to produce a `Doc` object. The `Doc` is then processed in several different steps. This is also referred to as the *processing pipeline*. The pipeline used by the default models consists of a *tagger*, a *parser*, and an *entity recognizer*. Each pipeline component returns the processed `Doc`, which is then passed on to the next component, as demonstrated in [Figure 10-3](#).

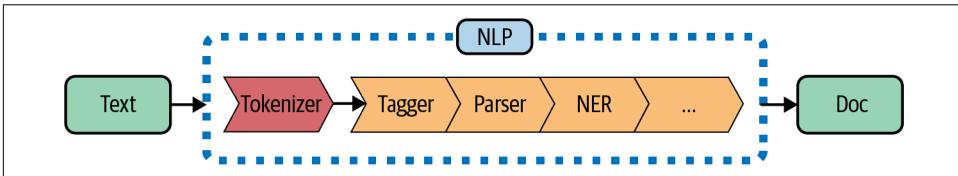


Figure 10-3. spaCy pipeline (based on an image from [the spaCy website](#)).

```
Python code text = 'Google is looking at buying U.K. startup for $1 billion'  
doc = nlp(text)  
pd.DataFrame([[t.text, t.is_stop, t.lemma_, t.pos_]  
             for t in doc],  
            columns=['Token', 'is_stop_word', 'lemma', 'POS'])
```

Output

	Token	is_stop_word	lemma	POS
0	Google	False	Google	PROPN
1	is	True	be	VERB
2	looking	False	look	VERB
3	at	True	at	ADP

Token		is_stop_word	lemma	POS
4	buying	False	buy	VERB
5	U.K.	False	U.K.	PROPN
6	startup	False	startup	NOUN
7	for	True	for	ADP
8	\$	False	\$	SYM
9	1	False	1	NUM
10	billion	False	billion	NUM

The output for each of the preprocessing steps is shown in the preceding table. Given that spaCy performs a wide range of NLP-related tasks in a single step, it is a highly recommended package. As such, we will be using spaCy extensively in our case studies.

In addition to the above preprocessing steps, there are other frequently used preprocessing steps, such as *lower casing* or *nonalphanumeric data removing*, that we can perform depending on the type of data. For example, data scraped from a website has to be cleansed further, including the removal of HTML tags. Data from a PDF report must be converted into a text format.

Other optional preprocessing steps include dependency parsing, coreference resolution, triplet extraction, and relation extraction:

Dependency parsing

Assigns a syntactic structure to sentences to make sense of how the words in the sentence relate to each other.

Coreference resolution

The process of connecting tokens that represent the same entity. It is common in languages to introduce a subject with their name in one sentence and then refer to them as him/her/it in subsequent sentences.

Triplet extraction

The process of recording subject, verb, and object triplets when available in the sentence structure.

Relation extraction

A broader form of triplet extraction in which entities can have multiple interactions.

These additional steps should be performed only if they will help with the task at hand. We will demonstrate examples of these preprocessing steps in the case studies in this chapter.

2. Feature Representation

The vast majority of NLP-related data, such as news feed articles, PDF reports, social media posts, and audio files, is created for human consumption. As such, it is often stored in an unstructured format, which cannot be readily processed by computers. In order for the preprocessed information to be conveyed to the statistical inference algorithm, the tokens need to be translated into predictive features. A model is used to embed raw text into a *vector space*.

Feature representation involves two things:

- A vocabulary of known words.
- A measure of the presence of known words.

Some of the feature representation methods are:

- Bag of words
- TF-IDF
- Word embedding
 - Pretrained models (e.g., word2vec, **GloVe**, spaCy's word embedding model)
 - Customized deep learning-based feature representation¹

Let's learn more about each of these methods.

2.1. Bag of words—word count

In natural language processing, a common technique for extracting features from text is to place all words that occur in the text in a bucket. This approach is called a *bag of words* model. It's referred to as a bag of words because any information about the structure of the sentence is lost. In this technique, we build a single matrix from a collection of texts, as shown in [Figure 10-4](#), in which each row represents a token and each column represents a document or sentence in our corpus. The values of the matrix represent the count of the number of instances of the token appearing.

¹ A customized deep learning-based feature representation model is built in case study 1 of this chapter.

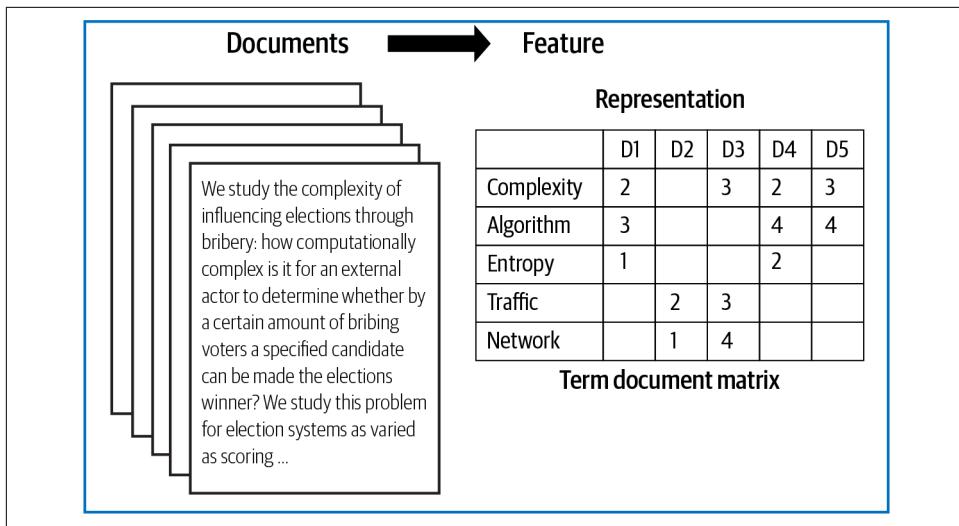


Figure 10-4. Bag of words

The CountVectorizer from sklearn provides a simple way to both tokenize a collection of text documents and encode new documents using that vocabulary. The `fit_transform` function learns the vocabulary from one or more documents and encodes each document in the word as a vector:

```

sentences = [
    'The stock price of google jumps on the earning data today',
    'Google plunge on China Data!'
]
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
print( vectorizer.fit_transform(sentences).todense() )
print( vectorizer.vocabulary_ )

```

Output

```

[[0 1 1 1 1 1 1 0 1 1 2 1]
 [1 1 0 1 0 0 1 1 0 0 0 0]]
{'the': 10, 'stock': 9, 'price': 8, 'of': 5, 'google': 3, 'jumps':\
 4, 'on': 6, 'earning': 2, 'data': 1, 'today': 11, 'plunge': 7,\n 'china': 0}

```

We can see an array version of the encoded vector showing a count of one occurrence for each word except *the* (index 10), which has an occurrence of two. Word counts are a good starting point, but they are very basic. One issue with simple counts is that some words like *the* will appear many times, and their large counts will not be very meaningful in the encoded vectors. These bag of words representations are sparse because the vocabularies are vast, and a given word or document would be represented by a large vector comprised mostly of zero values.

2.2. TF-IDF

An alternative is to calculate word frequencies, and by far the most popular method for that is *TF-IDF*, which stands for *Term Frequency-Inverse Document Frequency*:

Term Frequency

This summarizes how often a given word appears within a document.

Inverse Document Frequency

This downscales words that appear a lot across documents.

Put simply, TF-IDF is a word frequency score that tries to highlight words that are more interesting (i.e., frequent *within* a document, but not *across* documents). The *TfidfVectorizer* will tokenize documents, learn the vocabulary and the inverse document frequency weightings, and allow you to encode new documents:

```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer(max_features=1000, stop_words='english')
TFIDF = vectorizer.fit_transform(sentences)
print(vectorizer.get_feature_names()[-10:])
print(TFIDF.shape)
print(TFIDF.toarray())
```

Output

```
['china', 'data', 'earning', 'google', 'jumps', 'plunge', 'price', 'stock', \
'today']
(2, 9)
[[0.          0.29017021  0.4078241   0.29017021  0.4078241   0.\
 0.4078241   0.4078241   0.4078241 ]
 [0.57615236  0.40993715  0.          0.40993715  0.          0.57615236
 0.          0.          0.        ]]
```

In the provided code snippet, a vocabulary of nine words is learned from the documents. Each word is assigned a unique integer index in the output vector. The sentences are encoded as a nine-element sparse array, and we can review the final scorings of each word with different values from the other words in the vocabulary.

2.3. Word embedding

A *word embedding* represents words and documents using a dense vector representation. In an embedding, words are represented by dense vectors in which a vector represents the projection of the word into a continuous vector space. The position of a word within the vector space is learned from text and is based on the words that surround the word when it is used. The position of a word in the learned vector space is referred to as its *embedding*.

Some of the models of learning word embeddings from text include word2Vec, spaCy's pretrained word embedding model, and GloVe. In addition to these carefully designed methods, a word embedding can be learned as part of a deep learning

model. This can be a slower approach, but it tailors the model to a specific training dataset.

2.3.1. Pretrained model: Via spaCy

spaCy comes with built-in representation of text as vectors at different levels of word, sentence, and document. The underlying vector representations come from a word embedding model, which generally produces a dense, multidimensional semantic representation of words (as shown in the following example). The word embedding model includes 20,000 unique vectors with 300 dimensions. Using this vector representation, we can calculate similarities and dissimilarities between tokens, named entities, noun phrases, sentences, and documents.

The word embedding in spaCy is performed by first loading the model and then processing text. The vectors can be accessed directly using the `.vector` attribute of each processed token (i.e., word). The mean vector for the entire sentence is also calculated simply by using the vector, providing a very convenient input for machine learning models based on sentences:

```
doc = nlp("Apple orange cats dogs")
print("Vector representation of the sentence for first 10 features: \n", \
      doc.vector[0:10])
```

Output:\

```
Vector representation of the sentence for first 10 features:
[ -0.30732775  0.22351399 -0.110111   -0.367025   -0.13430001
  0.13790375 -0.24379876 -0.10736975   0.2715925   1.3117325 ]
```

The vector representation of the sentence for the first 10 features of the pretrained model is shown in the output.

2.3.2. Pretrained model: Word2Vec using gensim package

The Python-based implementation of the word2vec model using the `gensim` package is demonstrated here:

```
from gensim.models import Word2Vec

sentences = [
    ['The', 'stock', 'price', 'of', 'Google', 'increases'],
    ['Google', 'plunge', 'on', 'China', 'Data!']]

# train model
model = Word2Vec(sentences, min_count=1)

# summarize the loaded model
words = list(model.wv.vocab)
print(words)
print(model['Google'][1:5])
```

Output

```
['The', 'stock', 'price', 'of', 'Google', 'increases', 'plunge', 'on', 'China',\n 'Data!']\n[-1.7868265e-03 -7.6242397e-04 6.0105987e-05 3.5568199e-03\n]
```

The vector representation of the sentence for the first five features of the pretrained word2vec model is shown above.

3. Inference

As with other artificial intelligence tasks, an inference generated by an NLP application usually needs to be translated into a decision in order to be actionable. Inference falls under three machine learning categories covered in the previous chapters (i.e., supervised, unsupervised, and reinforcement learning). While the type of inference required depends on the business problem and the type of training data, the most commonly used algorithms are supervised and unsupervised.

One of the most frequently used supervised methodologies in NLP is the *Naive Bayes* model, as it can produce reasonable accuracy using simple assumptions. A more complex supervised methodology is using artificial neural network architectures. In past years, these architectures, such as recurrent neural networks (RNNs), have dominated NLP-based inference.

Most of the existing literature in NLP focuses on supervised learning. As such, unsupervised learning applications constitute a relatively less developed subdomain in which measuring *document similarity* is among the most common tasks. A popular unsupervised technique applied in NLP is *Latent Semantic Analysis* (LSA). LSA looks at relationships between a set of documents and the words they contain by producing a set of latent concepts related to the documents and terms. LSA has paved the way for a more sophisticated approach called *Latent Dirichlet Allocation* (LDA), under which documents are modeled as a finite mixture of topics. These topics in turn are modeled as a finite mixture over words in the vocabulary. LDA has been extensively used for *topic modeling*—a growing area of research in which NLP practitioners build probabilistic generative models to reveal likely topic attributions for words.

Since we have reviewed many supervised and unsupervised learning models in the previous chapters, we will provide details only on Naive Bayes and LDA models in the next sections. These are used extensively in NLP and were not covered in the previous chapters.

3.1. Supervised learning example—Naive Bayes

Naive Bayes is a family of algorithms based on applying *Bayes's theorem* with a strong (naive) assumption that every feature used to predict the category of a given sample is independent of the others. They are probabilistic classifiers and therefore will

calculate the probability of each category using Bayes's theorem. The category with the highest probability will be output.

In NLP, a Naive Bayes approach assumes that all word features are independent of each other given the class labels. Due to this simplifying assumption, Naive Bayes is very compatible with a bag-of-words word representation, and it has been demonstrated to be fast, reliable, and accurate in a number of NLP applications. Moreover, despite its simplifying assumptions, it is competitive with (and at times even outperforms) more complicated classifiers.

Let us look at the usage of Naive Bayes for the inference in a sentiment analysis problem. We take a dataframe in which there are two sentences with sentiments assigned to each. In the next step, we convert the sentences into a feature representation using `CountVectorizer`. The features and sentiments are used to train and test the model using Naive Bayes:

```
sentences = [
    'The stock price of google jumps on the earning data today',
    'Google plunge on China Data!']
sentiment = (1, 0)
data = pd.DataFrame({'Sentence':sentences,
                     'sentiment':sentiment})

# feature extraction
from sklearn.feature_extraction.text import CountVectorizer
vect = CountVectorizer().fit(data['Sentence'])
X_train_vectorized = vect.transform(data['Sentence'])

# Running naive bayes model
from sklearn.naive_bayes import MultinomialNB
clfNB = MultinomialNB(alpha=0.1)
clfNB.fit(X_train_vectorized, data['sentiment'])

#Testing the model
preds = clfNB.predict(vect.transform(['Apple price plunge', \
    'Amazon price jumps']))
preds
```

Output

```
array([0, 1])
```

As we can see, the Naive Bayes trains the model fairly well from the two sentences. The model gives a sentiment of zero and one for the test sentences “Apple price plunge” and “Amazon price jumps,” respectively, given the sentences used for training also had the keywords “plunge” and “jumps,” with corresponding sentiment assignments.

3.2. Unsupervised learning example: LDA

LDA is extensively used for *topic modeling* because it tends to produce meaningful topics that humans can interpret, assigns topics to new documents, and is extensible. It works by first making a key assumption: documents are generated by first selecting *topics*, and then, for each topic, a set of *words*. The algorithm then reverse engineers this process to find the topics in a document.

In the following code snippet, we show an implementation of LDA for topic modeling. We take two sentences and convert the sentences into a feature representation using CountVectorizer. These features and the sentiments are used to train the model and produce two smaller matrices representing the topics:

```
sentences = [  
    'The stock price of google jumps on the earning data today',  
    'Google plunge on China Data!'  
]  
  
#Getting the bag of words  
from sklearn.decomposition import LatentDirichletAllocation  
vect=CountVectorizer(ngram_range=(1, 1),stop_words='english')  
sentences_vec=vect.fit_transform(sentences)  
  
#Running LDA on the bag of words.  
from sklearn.feature_extraction.text import CountVectorizer  
lda=LatentDirichletAllocation(n_components=3)  
lda.fit_transform(sentences_vec)
```

Output

```
array([[0.04283242, 0.91209846, 0.04506912],  
       [0.06793339, 0.07059533, 0.86147128]])
```

We will be using LDA for topic modeling in the third case study of this chapter and will discuss the concepts and interpretation in detail.

To review, in order to approach any NLP-based problem, we need to follow the pre-processing, feature extraction, and inference steps. Now, let's dive into the case studies.

Case Study 1: NLP and Sentiment Analysis-Based Trading Strategies

Natural language processing offers the ability to quantify text. One can begin to ask questions such as: How positive or negative is this news? and How can we quantify words?

Perhaps the most notable application of NLP is its use in algorithmic trading. NLP provides an efficient means of monitoring market sentiments. By applying

NLP-based sentiment analysis techniques to news articles, reports, social media, or other web content, one can effectively determine whether those sources have a positive or negative sentiment score. Sentiment scores can be used as a directional signal to buy stocks with positive scores and sell stocks with negative ones.

Trading strategies based on text data are becoming more popular as the amount of unstructured data increases. In this case study we are going to look at how one can use NLP-based sentiments to build a trading strategy.

In this case study, we will focus on:

- Producing news sentiments using supervised and unsupervised algorithms.
- Enhancing sentiment analysis by using a deep learning model, such as LSTM.
- Comparison of different sentiment generation methodologies for the purpose of building a trading strategy.
- Using sentiments and word vectors effectively as features in a trading strategy.
- Collecting data from different sources and preprocessing it for sentiment analysis.
- Using NLP Python packages for sentiment analysis.
- Building a framework for backtesting results of a trading strategy using available Python packages.

This case study combines concepts presented in previous chapters. The overall model development steps of this case study are similar to the seven-step model development in prior case studies, with slight modifications.



Blueprint for Building a Trading Strategy Based on Sentiment Analysis

1. Problem definition

Our goal is to (1) use NLP to extract information from news headlines, (2) assign a sentiment to that information, and (3) use sentiment analysis to build a trading strategy.

The data used for this case study will be from the following sources:

News headlines data compiled from the RSS feeds of several news websites

For the purpose of this study, we will look only at the headlines, not at the full text of the stories. Our dataset contains around 82,000 headlines from May 2011 through December 2018.²

Yahoo Finance website for stock data

The return data for stocks used in this case study is derived from Yahoo Finance price data.

Kaggle

We will use the labeled data of news sentiments for a classification-based sentiment analysis model. Note that this data may not be fully applicable to the case at hand and is used here for demonstration purposes.

Stock market lexicon

Lexicon refers to the component of an NLP system that contains information (semantic, grammatical) about individual words or word strings. This is created based on stock market conversations in microblogging services.³

The key steps of this case study are outlined in [Figure 10-5](#).

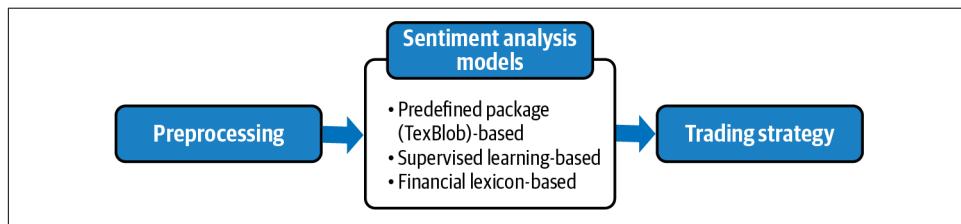


Figure 10-5. Steps in a sentiment analysis-based trading strategy

Once we are done with preprocessing, we will look at the different sentiment analysis models. The results from the sentiment analysis step are used to develop the trading strategy.

² The news can be downloaded by a simple web-scraping program in Python using packages such as Beautiful Soup. Readers should talk to the website or follow its terms of service in order to use the news for commercial purpose.

³ The source of this lexicon is Nuno Oliveira, Paulo Cortez, and Nelson Areal, "Stock Market Sentiment Lexicon Acquisition Using Microblogging Data and Statistical Measures," *Decision Support Systems* 85 (March 2016): 62–73.

2. Getting started—loading the data and Python packages

2.1. Loading the Python packages. The first set of libraries to be loaded are the NLP-specific libraries discussed above. Refer to the Jupyter notebook of this case study for details of the other libraries.

```
from textblob import TextBlob
import spacy
import nltk
import warnings
from nltk.sentiment.vader import SentimentIntensityAnalyzer
nltk.download('vader_lexicon')
nlp = spacy.load("en_core_web_lg")
```

2.2. Loading the data. In this step, we load the stock price data from Yahoo Finance. We select 10 stocks for this case study. These stocks are some of the largest stocks in the S&P 500 by market share:

```
tickers = ['AAPL', 'MSFT', 'AMZN', 'GOOG', 'FB', 'WMT', 'JPM', 'TSLA', 'NFLX', 'ADBE']
start = '2010-01-01'
end = '2018-12-31'
df_ticker_return = pd.DataFrame()
for ticker in tickers:
    ticker_yf = yf.Ticker(ticker)
    if df_ticker_return.empty:
        df_ticker_return = ticker_yf.history(start = start, end = end)
        df_ticker_return['ticker']= ticker
    else:
        data_temp = ticker_yf.history(start = start, end = end)
        data_temp['ticker']= ticker
        df_ticker_return = df_ticker_return.append(data_temp)
df_ticker_return.to_csv(r'Data\Step3.2_ReturnData.csv')
df_ticker_return.head(2)
```

	Open	High	Low	Close	Volume	Dividends	Stock Splits	ticker
Date								
2010-01-04	26.40	26.53	26.27	26.47	123432400	0.0	0.0	AAPL
2010-01-05	26.54	26.66	26.37	26.51	150476200	0.0	0.0	AAPL

The data contains the price and volume data of the stocks along with their ticker name. In the next step, we look at the news data.

3. Data preparation

In this step, we load and preprocess the news data, followed by combining the news data with the stock return data. This combined dataset will be used for the model development.

3.1. Preprocessing news data. The news data is downloaded from the News RSS feed, and the file is available in JSON format. The JSON files for different dates are kept under a zipped folder. The data is downloaded using the standard web-scraping Python package BeautifulSoup, which is an open source framework. Let us look at the content of the downloaded JSON file:

```
z = zipfile.ZipFile("Data/Raw Headline Data.zip", "r")
testFile=z.namelist()[10]
fileData= z.open(testFile).read()
fileDataSample = json.loads(fileData)['content'][1:500]
fileDataSample
```

Output

```
'li class="n-box-item date-title" data-end="1305172799" data-start="1305086400"
data-txt="Tuesday, December 17, 2019">Wednesday, May 11,2011</li><li
class="n-box-item sa-box-item" data-id="76179" data-ts="1305149244"><div
class="media media-overflow-fix"><div class="media-left"><a class="box-ticker"
href="/symbol/CSCO" target="blank">CSCO</a></div><div class="media-body"><h4
class="media-heading"><a href="/news/76179" sasource="on_the_move_news_
fidelity" target="_blank">Cisco (NASDAQ:CSCO): Pr'
```

We can see that the JSON format is not suitable for the algorithm. We need to get the news from the JSONs. Regex becomes the vital part of this step. Regex can find a pattern in the raw, messy text and perform actions accordingly. The following function parses HTML by using information encoded in the JSON file:

```
def jsonParser(json_data):
    xml_data = json_data['content']

    tree = etree.parse(StringIO(xml_data), parser=etree.HTMLParser())

    headlines = tree.xpath("//h4[contains(@class, 'media-heading')]/a/text()")
    assert len(headlines) == json_data['count']

    main_tickers = list(map(lambda x: x.replace('/symbol/', ''), \
                           tree.xpath("//div[contains(@class, 'media-left')]/a/@href")))
    assert len(main_tickers) == json_data['count']
    final_headlines = [''.join(f.xpath('.//text()')) for f in \
                       tree.xpath("//div[contains(@class, 'media-body')]/ul/li[1]")]
    if len(final_headlines) == 0:
        final_headlines = [''.join(f.xpath('.//text()')) for f in \
                           tree.xpath("//div[contains(@class, 'media-body')]")]
        final_headlines = [f.replace(h, '').split('\xa0')[0].strip() \
                           for f,h in zip (final_headlines, headlines)]
    return main_tickers, final_headlines
```

Let us see how the output looks like after running the JSON parser:

```
jsonParser(json.loads(fileData))[1][1]
```

Output

```
'Cisco Systems (NASDAQ:CSCO) falls further into the red on FQ4  
guidance of $0.37-0.39 vs. $0.42 Street consensus. Sales seen flat  
to +2% vs. 8% Street view. CSCO recently -2.1%.'
```

As we can see, the output is converted into a more readable format after JSON parsing.

While evaluating the sentiment analysis models, we also analyze the relationship between the sentiments and subsequent stock performance. In order to understand the relationship, we use *event return*, which is the return that corresponds to the event. We do this because at times the news is reported late (i.e., after market participants are aware of the announcement) or after market close. Having a slightly wider window ensures that we capture the essence of the event. *Event return* is defined as:

$$R_{t-1} + R_t + R_{t+1}$$

where R_{t-1} , R_{t+1} are the returns before and after the news data, and R_t is the return on the day of the news (i.e., time t).

Let us extract the event return from the data:

```
#Computing the return  
df_ticker_return['ret_curr'] = df_ticker_return['Close'].pct_change()  
#Computing the event return  
df_ticker_return['eventRet'] = df_ticker_return['ret_curr']\n    + df_ticker_return['ret_curr'].shift(-1) + df_ticker_return['ret_curr'].shift(1)
```

Now we have all the data in place. We will prepare a combined dataframe, which will have the news headlines mapped to the date, the returns (event return, current return, and next day's return), and stock ticker. This dataframe will be used for building the sentiment analysis model and the trading strategy:

```
combinedDataFrame = pd.merge(data_df_news, df_ticker_return, how='left', \  
left_on=['date','ticker'], right_on=['date','ticker'])  
combinedDataFrame = combinedDataFrame[combinedDataFrame['ticker'].isin(tickers)]  
data_df = combinedDataFrame[['ticker','headline','date','eventRet','Close']]  
data_df = data_df.dropna()  
data_df.head(2)
```

Output

	ticker	headline		date	eventRet	Close
5	AMZN	Whole Foods (WFM) -5.2% following a downgrade...		2011-05-02	0.017650	201.19
11	NFLX	Netflix (NFLX +1.1%) shares post early gains a...		2011-05-02	-0.013003	33.88

Let us look at the overall shape of the data:

```
print(data_df.shape, data_df.ticker.unique().shape)
```

Output

```
(2759, 5) (10,)
```

In this step, we prepared a clean dataframe that has ticker, headline, event return, return for a given day, and future return for 10 stock tickers, totaling 2,759 rows of data. Let us evaluate the models for sentiment analysis in the next step.

4. Evaluate models for sentiment analysis

In this section, we will go through the following three approaches of computing sentiments for the news:

- Predefined model—TextBlob package
- Tuned model—classification algorithms and LSTM
- Model based on financial lexicon

Let us go through the steps.

4.1. Predefined model—TextBlob package. The `TextBlob` `sentiment` function is a pre-trained model based on the Naive Bayes classification algorithm. The function maps adjectives that are frequently found in movie reviews⁴ to sentiment polarity scores ranging from -1 to $+1$ (negative to positive), converting a sentence to a numerical value. We apply this on all headline articles. An example of getting the sentiment for a news text is shown below:

```
text = "Bayer (OTCPK:BAYRY) started the week up 3.5% to €74/share in Frankfurt, \
touching their
highest level in 14 months, after the U.S. government said \
a $25M glyphosate decision against the
company should be reversed."
TextBlob(text).sentiment.polarity
```

Output

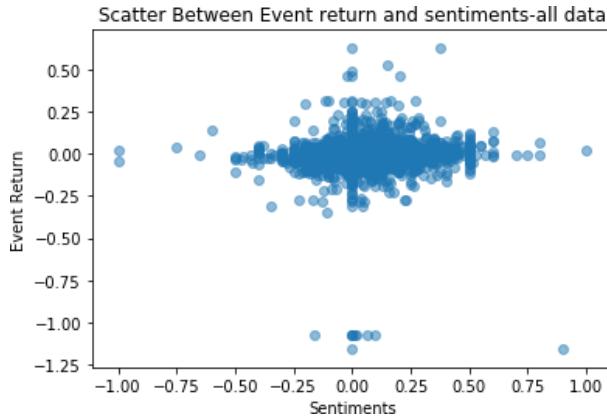
```
0.5
```

⁴ We also train a sentiment analysis model on the financial data in the subsequent section and compare the results against the TextBlob model.

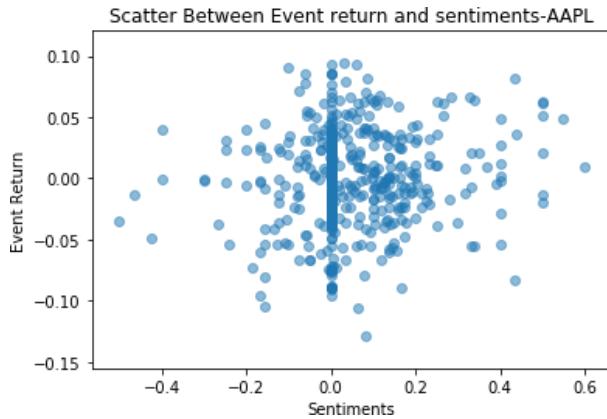
The sentiment for the statement is 0.5. We apply this on all headlines we have in the data:

```
data_df['sentiment_textblob'] = [TextBlob(s).sentiment.polarity for s in \
data_df['headline']]
```

Let us inspect the scatterplot of the sentiments and returns to examine the correlation between the two for all 10 stocks.



A plot for a single stock (APPL) is also shown in the following chart (see the code in the Jupyter notebook in the GitHub repository for this book for more details on the code):



From the scatterplots, we can see that there is not a strong relationship between the news and the sentiments. The correlation between return and sentiments is positive (4.27%), which means that news with positive sentiments leads to positive return and is expected. However, the correlation is not very high. Even looking at the overall

scatterplot, we see the majority of the sentiments concentrated around zero. This raises the question of whether a sentiment score trained on movie reviews is appropriate for stock prices. The `sentiment_assessments` attribute lists the underlying values for each token and can help us understand the reason for the overall sentiment of a sentence:

```
text = "Bayer (OTCPK:BAYRY) started the week up 3.5% to €74/share\\
in Frankfurt, touching their highest level in 14 months, after the\\
U.S. government said a $25M glyphosate decision against the company\\
should be reversed."
TextBlob(text).sentiment_assessments
```

Output

```
Sentiment(polarity=0.5, subjectivity=0.5, assessments=[(['touching'], 0.5, 0.5, \
None)])
```

We see that the statement has a positive sentiment of 0.5, but it appears the word “touching” gave rise to the positive sentiment. More intuitive words, such as “high,” do not. This example shows that the context of the training data is important for the sentiment score to be meaningful. There are many predefined packages and functions available for sentiment analysis, but it is important to be careful and have a thorough understanding of the problem’s context before using a function or an algorithm for sentiment analysis.

For this case study, we may need sentiments trained on the financial news. Let us take a look at that in the next step.

4.2. Supervised learning—classification algorithms and LSTM

In this step, we develop a customized model for sentiment analysis based on available labeled data. The label data for this is obtained from the [Kaggle website](#):

```
sentiments_data = pd.read_csv(r'Data\LabelledNewsData.csv', \
encoding="ISO-8859-1")
sentiments_data.head(1)
```

Output

	datetime	headline	ticker	sentiment
0	1/16/2020 5:25	\$MMM fell on hard times but could be set to re...	MMM	0
1	1/11/2020 6:43	Wolfe Research Upgrades 3M \$MMM to ;\$Peer Perf...	MMM	1

The data has headlines for the news across 30 different stocks, totaling 9,470 rows, and has sentiments labeled zero and one. We perform the classification steps using the classification model development template presented in [Chapter 6](#).

In order to run a supervised learning model, we first need to convert the news headlines into a feature representation. For this exercise, the underlying vector

representations come from a *spaCy word embedding model*, which generally produces a dense, multidimensional semantic representation of words (as shown in the example below). The word embedding model includes 20,000 unique vectors with 300 dimensions. We apply this on all headlines in the data processed in the previous step:

```
all_vectors = pd.np.array([pd.np.array([token.vector for token in nlp(s)]).\\
mean(axis=0)*pd.np.ones((300))\\
for s in sentiments_data['headline']]])
```

Now that we have prepared the independent variable, we train the classification model in a similar manner as discussed in [Chapter 6](#). We have the sentiments label zero or one as the dependent variable. We first divide the data into training and test sets and run the key classification models (i.e., logistic regression, CART, SVM, random forest, and artificial neural network).

We will also include LSTM, which is an RNN-based model,⁵ in the list of models considered. An RNN-based model performs well for NLP, because it stores the information for current features as well neighboring ones for prediction. It maintains a memory based on past information, which enables the model to predict the current output conditioned on long distance features and looks at the words in the context of the entire sentence, rather than simply looking at the individual words.

For us to be able to feed the data into our LSTM model, all input documents must have the same length. We use the Keras `tokenizer` function to tokenize the strings and then use `texts_to_sequences` to make sequences of words. More details can be found on the [Keras website](#). We will limit the maximum review length to `max_words` by truncating longer reviews and pad shorter reviews with a null value (0). We can accomplish this using the `pad_sequences` function, also in Keras. The third parameter is the `input_length` (set to 50), which is the length of each comment sequence:

```
### Create sequence
vocabulary_size = 20000
tokenizer = Tokenizer(num_words=vocabulary_size)
tokenizer.fit_on_texts(sentiments_data['headline'])
sequences = tokenizer.texts_to_sequences(sentiments_data['headline'])
X_LSTM = pad_sequences(sequences, maxlen=50)
```

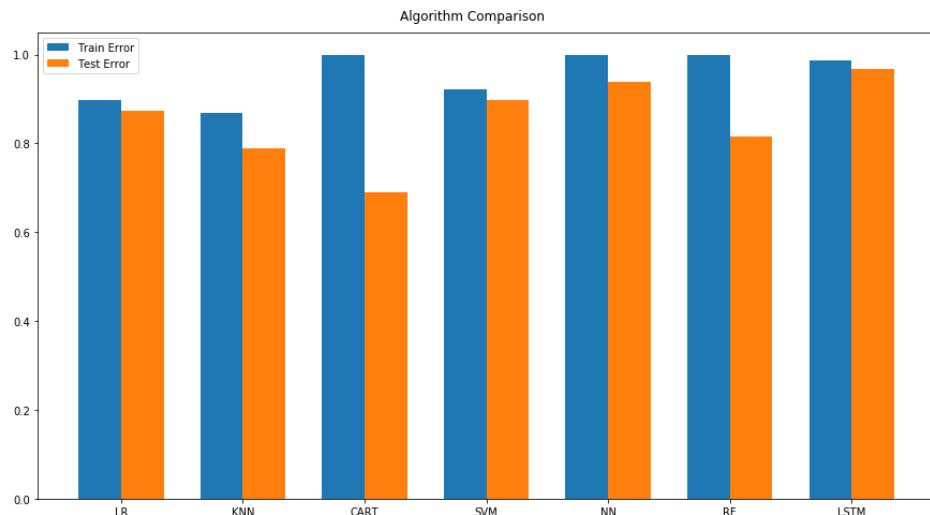
In the following code snippet, we use the Keras library to build an artificial neural network classifier based on an underlying LSTM model. The network starts with an *embedding* layer. This layer lets the system expand each token to a larger vector, allowing the network to represent a word in a meaningful way. The layer takes 20,000 as the first argument (i.e., the size of our vocabulary) and 300 as the second input parameter (i.e., the dimension of the embedding). Finally, given that this is a classification problem and the output needs to be labeled as zero or one, the

⁵ Refer to [Chapter 5](#) for more details on RNN models.

`KerasClassifier` function is used as a wrapper over the LSTM model to produce a binary (zero or one) output:

```
from keras.wrappers.scikit_learn import KerasClassifier
def create_model(input_length=50):
    model = Sequential()
    model.add(Embedding(20000, 300, input_length=50))
    model.add(LSTM(100, dropout=0.2, recurrent_dropout=0.2))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam', \
    metrics=['accuracy'])
    return model
model_LSTM = KerasClassifier(build_fn=create_model, epochs=3, verbose=1, \
    validation_split=0.4)
model_LSTM.fit(X_train_LSTM, Y_train_LSTM)
```

The comparison of all the machine learning models is as follows:



As expected, the LSTM model has the best performance in the test set (accuracy of 96.7%) as compared to all other models. The performance of the ANN, with a training set accuracy of 99% and a test set accuracy of 93.8%, is comparable to the LSTM-based model. The performances of random forest (RF), SVM, and logistic regression (LR) are reasonable as well. CART and KNN do not perform as well as other models. CART shows high overfitting. Let us use the LSTM model for the computation of the sentiments in the data in the following steps.

4.3. Unsupervised—model based on a financial lexicon

In this case study, we update the VADER lexicon with words and sentiments from a lexicon adapted to stock market conversations in microblogging services:

Lexicons

Special dictionaries or vocabularies that have been created for analyzing sentiments. Most lexicons have a list of positive and negative *polar* words with some score associated with them. Using various techniques, such as the position of words, the surrounding words, context, parts of speech, and phrases, scores are assigned to the text documents for which we want to compute the sentiment. After aggregating these scores, we get the final sentiment:

VADER (*Valence Aware Dictionary for Sentiment Reasoning*)

A prebuilt sentiment analysis model included in the NLTK package. It can give both positive and negative polarity scores as well as the strength of the emotion of a text sample. It is rule-based and relies heavily on human-rated texts. These are words or any textual form of communication labeled according to their semantic orientation as either positive or negative.

This lexical resource was automatically created using diverse statistical measures and a large set of labeled messages from StockTwits, which is a social media platform designed for sharing ideas among investors, traders, and entrepreneurs.⁶ The sentiments are between -1 and 1, similar to the sentiments from TextBlob. In the following code snippet, we train the model based on the financial sentiments:

```
# stock market lexicon
sia = SentimentIntensityAnalyzer()
stock_lex = pd.read_csv('Data/lexicon_data/stock_lex.csv')
stock_lex['sentiment'] = (stock_lex['Aff_Score'] + stock_lex['Neg_Score'])/2
stock_lex = dict(zip(stock_lex.Item, stock_lex.sentiment))
stock_lex = {k:v for k,v in stock_lex.items() if len(k.split(' '))==1}
stock_lex_scaled = {}
for k, v in stock_lex.items():
    if v > 0:
        stock_lex_scaled[k] = v / max(stock_lex.values()) * 4
    else:
        stock_lex_scaled[k] = v / min(stock_lex.values()) * -4

final_lex = []
final_lex.update(stock_lex_scaled)
final_lex.update(sia.lexicon)
sia.lexicon = final_lex
```

⁶ The source of this lexicon is Nuno Oliveira, Paulo Cortez, and Nelson Areal, “Stock Market Sentiment Lexicon Acquisition Using Microblogging Data and Statistical Measures,” *Decision Support Systems* 85 (March 2016): 62–73.

Let us check the sentiment of a news item:

```
text = "AAPL is trading higher after reporting its October sales\\
rose 12.6% M/M. It has seen a 20%+ jump in orders"
sia.polarity_scores(text)['compound']
```

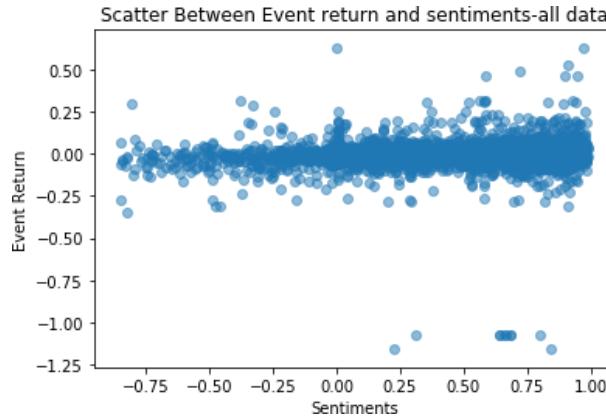
Output

```
0.4535
```

We get the sentiments for all the news headlines based in our dataset:

```
vader_sentiments = pd.np.array([sia.polarity_scores(s)['compound'] \\
for s in data_df['headline']])
```

Let us look at the relationship between the returns and sentiments, which is computed using the lexicon-based methodology for the entire dataset.



There are not many instances of high returns for lower sentiment scores, but the data may not be very clear. We will look deeper into the comparison of different types of sentiment analysis in the next section.

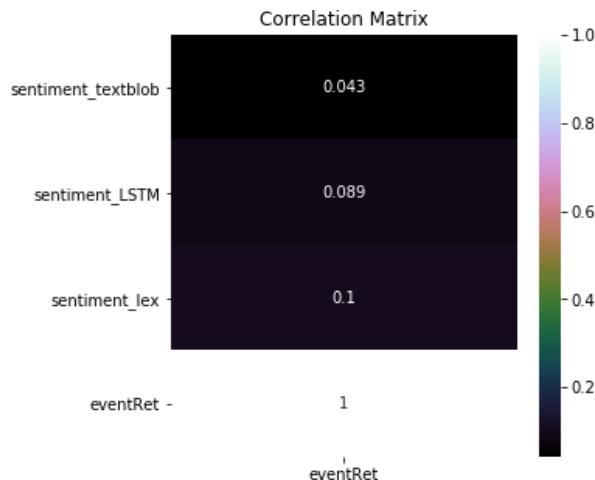
4.4. Exploratory data analysis and comparison

In this section, we compare the sentiments computed using the different techniques presented above. Let us look at the sample headlines and the sentiments from three different methodologies, followed by a visual analysis:

	ticker	headline	sentiment_textblob	sentiment_LSTM	sentiment_lex
4620	TSM	TSMC (TSM +1.8%) is trading higher after reporting its October sales rose 12.6% M/M. DigiTimes adds TSMC has seen a 20%+ jump in orders from QCOM, NVDA, SPRD, and Mediatek. The numbers suggest TSMC could beat its Q4 guidance (though December tends to be weak), and that chip demand could be stabilizing after getting hit hard by inventory corrections. (earlier) (UMC sales)	0.036667	1	0.5478

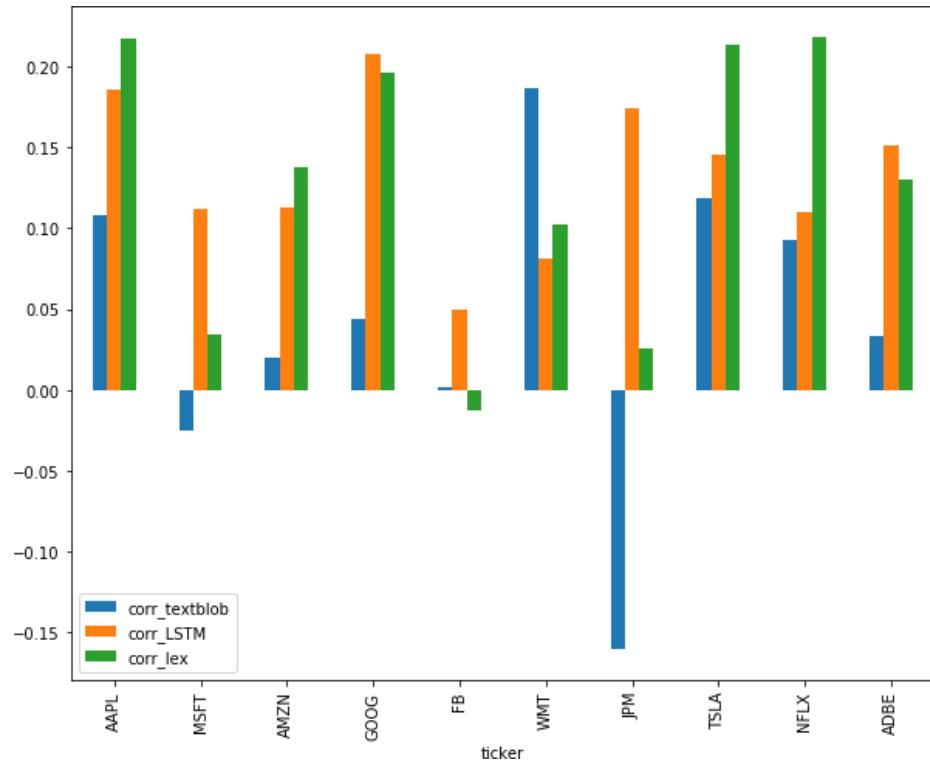
Looking at one of the headlines, the sentiment from this sentence is positive. However, the TextBlob sentiment result is smaller in magnitude, suggesting that the sentiment is more neutral. This points back to the previous assumption that the model trained on movie sentiments likely will not be accurate for stock sentiments. The classification-based model correctly suggests the sentiment is positive, but it is binary. Sentiment_lex has a more intuitive output with a significantly positive sentiment.

Let us review the correlation of all the sentiments from different methodologies versus returns:



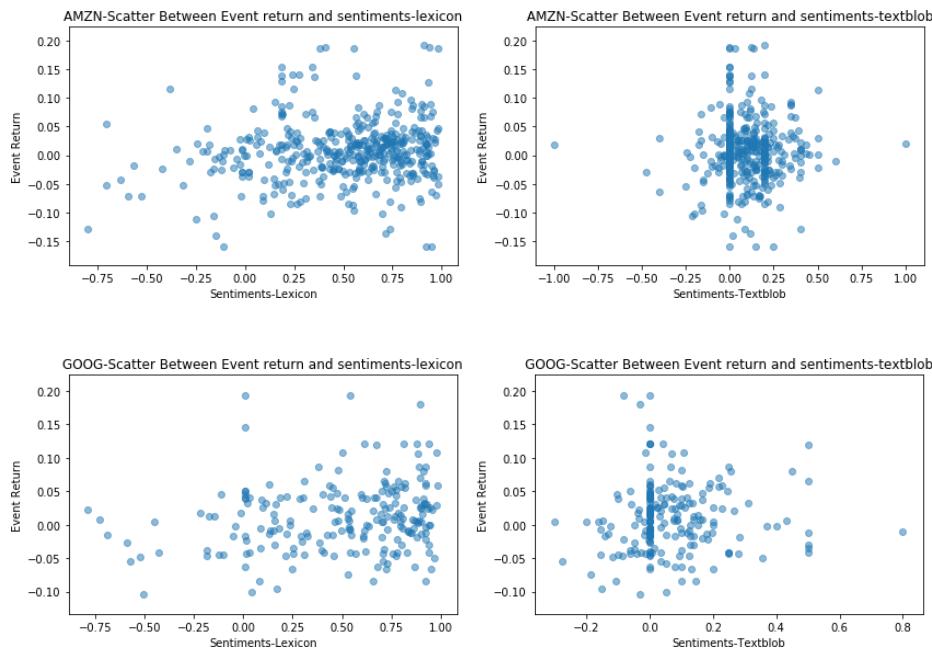
All sentiments have positive relationships with the returns, which is intuitive and expected. The sentiments from the lexicon methodology are highest, which means the stock's event return can be predicted the best using the lexicon methodology. Recall that this methodology leverages financial terms in the model. The LSTM-based method also performs better than the TextBlob approach, but the performance is slightly worse compared to the lexicon-based methodology.

Let us look at the performance of the methodology at the ticker level. We chose a few tickers with the highest market cap for the analysis:



Looking at the chart, the correlation from the lexicon methodology is highest across all stock tickers, which corroborates the conclusion from the previous analysis. It means the returns can be predicted the best using the lexicon methodology. The TextBlob-based sentiments show unintuitive results in some cases, such as with JPM.

Let us look at the scatterplot for lexicon versus TextBlob methodologies for AMZN and GOOG. We will set the LSTM-based method aside since the binary sentiments will not be meaningful in the scatterplot:



The lexicon-based sentiments on the left show a positive relationship between the sentiments and returns. Some of the points with the highest returns are associated with the most positive news. Also, the scatterplot is more uniformly distributed in the case of lexicon as compared to TextBlob. The sentiments for TextBlob are concentrated around zero, probably because the model is not able to categorize financial sentiments well. For the trading strategy, we will be using the lexicon-based sentiments, as these are the most appropriate based on the analysis in this section. The LSTM-based sentiments are good as well, but they are labeled either zero or one. The more granular lexicon-based sentiments are preferred.

5. Models evaluation—building a trading strategy

The sentiment data can be used in several ways for building a trading strategy. The sentiments can be used as a stand-alone signal to decide buy, sell, or hold actions. The sentiment score or the word vectors can also be used to predict the return or price of a stock. That prediction can be used to build a trading strategy.

In this section, we demonstrate a trading strategy in which we buy or sell a stock based on the following approach:

- Buy a stock when the change in sentiment score (current sentiment score/previous sentiment score) is greater than 0.5. Sell a stock when the change in sentiment score is less than -0.5. The sentiment score used here is based on the lexicon-based sentiments computed in the previous step.
- In addition to the sentiments, we use moving average (based on the last 15 days) while making a buy or sell decision.
- Trades (i.e., buy or sell) are in 100 shares. The initial amount available for trading is set to \$100,000.

The strategy threshold, the lot size, and the initial capital can be tweaked depending on the performance of the strategy.

5.1. Setting up a strategy. To set up the trading strategy, we use *backtrader*, which is a convenient Python-based framework for implementing and backtesting trading strategies. Backtrader allows us to write reusable trading strategies, indicators, and analyzers instead of having to spend time building infrastructure. We use the [Quick-start code in the backtrader documentation](#) as a base and adapt it to our sentiment-based trading strategy.

The following code snippet summarizes the buy and sell logic for the strategy. Refer to the Jupyter notebook of this case study for the detailed implementation:

```
# buy if current close more than simple moving average (sma)
# AND sentiment increased by >= 0.5
if self.dataclose[0] > self.sma[0] and self.sentiment - prev_sentiment >= 0.5:
    self.order = self.buy()

# sell if current close less than simple moving average(sma)
# AND sentiment decreased by >= 0.5
if self.dataclose[0] < self.sma[0] and self.sentiment - prev_sentiment <= -0.5:
    self.order = self.sell()
```

5.2. Results for individual stocks. First, we run our strategy on GOOG and look at the results:

```
ticker = 'GOOG'
run_strategy(ticker, start = '2012-01-01', end = '2018-12-12')
```

The output shows the trading log for some of the days and the final return:

Output

```
Starting Portfolio Value: 100000.00
2013-01-10, Previous Sentiment 0.08, New Sentiment 0.80 BUY CREATE, 369.36
2014-07-17, Previous Sentiment 0.73, New Sentiment -0.22 SELL CREATE, 572.16
2014-07-18, OPERATION PROFIT, GROSS 22177.00, NET 22177.00
2014-07-18, Previous Sentiment -0.22, New Sentiment 0.77 BUY CREATE, 593.45
2014-09-12, Previous Sentiment 0.66, New Sentiment -0.05 SELL CREATE, 574.04
```

2014-09-15, OPERATION PROFIT, GROSS -1876.00, NET -1876.00
2015-07-17, Previous Sentiment 0.01, New Sentiment 0.90 BUY CREATE, 672.93

.

2018-12-11, Ending Value 149719.00

We analyze the backtesting result in the following plot produced by the backtrader package. Refer to the Jupyter notebook of this case study for the detailed version of this chart.



The results show an overall profit of \$49,719. The chart is a typical chart⁷ produced by the backtrader package and is divided into four panels:

Top panel

The top panel is the *cash value observer*. It keeps track of the cash and the total portfolio value during the life of the backtesting run. In this run, we started with \$100,000 and ended with \$149,719.

Second panel

This panel is the *trade observer*. It shows the realized profit/loss of each trade. A trade is defined as opening a position and taking the position back to zero

⁷ Refer to the plotting section of the [backtrader website](#) for more details on the backtrader's charts and the panels.

(directly or crossing over from long to short or short to long). Looking at this panel, five out of eight trades are profitable for the strategy.

Third panel

This panel is *buy sell observer*. It indicates where buy and sell operations have taken place. In general, we see that the buy action takes place when the stock price is increasing, and the sell action takes place when the stock price has started declining.

Bottom panel

This panel shows the sentiment score, varying between -1 and 1.

Now we choose one of the days (2015-07-17) when a buy action was triggered and analyze the news for Google on that and the previous day:

```
GOOG_ticker= data_df[data_df['ticker'].isin([ticker])]  
New= list(GOOG_ticker[GOOG_ticker['date'] == '2015-07-17']['headline'])  
Old= list(GOOG_ticker[GOOG_ticker['date'] == '2015-07-16']['headline'])  
print("Current News:", New, "\n\n", "Previous News:", Old)
```

Output

Current News: ["Axiom Securities has upgraded Google (GOOG +13.4%, GOOGL +14.8%) to Buy following the company's Q2 beat and investor-pleasing comments about spending discipline, potential capital returns, and YouTube/mobile growth. MKM has launched coverage at Buy, and plenty of other firms have hiked their targets. Google's market cap is now above \$450B."]

Previous News: ["While Google's (GOOG, GOOGL) Q2 revenue slightly missed estimates when factoring traffic acquisitions costs (TAC), its ex-TAC revenue of \$14.35B was slightly above a \$14.3B consensus. The reason: TAC fell to 21% of ad revenue from Q1's 22% and Q2 2014's 23%. That also, of course, helped EPS beat estimates.", 'Google (NASDAQ:GOOG): QC2 EPS of \$6.99 beats by \$0.28.]

Clearly, the news on the selected day mentions the upgrade of Google, a piece of positive news. The previous day mentions the revenue missing estimates, which is negative news. Hence, there was a significant change of the news sentiment on the selected day, resulting in a buy action triggered by the trading algorithm.

Next, we run the strategy for FB:

```
ticker = 'FB'  
run_strategy(ticker, start = '2012-01-01', end = '2018-12-12')
```

Output

```
Start Portfolio value: 100000.00  
Final Portfolio Value: 108041.00  
Profit: 8041.00
```



The details of the backtesting results of the strategy are as follows:

Top panel

The cash value panel shows an overall profit of \$8,041.

Second panel

The trade observer panel shows that six out of seven actions were profitable.

Third panel

The buy/sell observer shows that in general the buy (sell) action took place when the stock price was increasing (decreasing).

Bottom panel

It shows a high number of positive sentiments for FB around the 2013–2014 period.

5.3. Results for multiple stocks. In the previous step, we executed the trading strategy on individual stocks. Here, we run it on all 10 stocks for which we computed the sentiments:

```
results_tickers = []
for ticker in tickers:
    results_tickers[ticker] = run_strategy(ticker, start = '2012-01-01', \
    end = '2018-12-12')
pd.DataFrame.from_dict(results_tickers).set_index(\n    [pd.Index(["PerUnitStartPrice", "StrategyProfit"])]))
```

Output

	AAPL	MSFT	AMZN	GOOG	FB	WMT	JPM	TSLA	NFLX	ADBE
PerUnitStartPrice	50.86	21.96	179.03	331.46	38.23	48.78	27.31	28.08	10.32	28.57
StrategyProfit	3735.00	4067.00	75377.00	49719.00	8041.00	1152.00	2014.00	15755.00	25181.00	17027.00

The strategy performs quite well and yields an overall profit for all the stocks. As mentioned before, the buy and sell actions are performed in a lot size of 100. Hence, the dollar amount used is proportional to the stock price. We see the highest nominal profit from AMZN and GOOG, which is primarily attributed to the high dollar amounts invested for these stocks given their high stock price. Other than overall profit, several other metrics, such as Sharpe ratio and maximum drawdown, can be used to analyze the performance.

5.4. Varying the strategy time period

In the previous analysis, we used the time period from 2011 to 2018 for our backtesting. In this step, to further analyze the effectiveness of our strategy, we vary the time period of the backtesting and analyze the results. First, we run the strategy for all the stocks for the time period between 2012 and 2014:

```
results_tickers = []
for ticker in tickers:
    results_tickers[ticker] = run_strategy(ticker, start = '2012-01-01', \
    end = '2014-12-31')
```

Output

	AAPL	MSFT	AMZN	GOOG	FB	WMT	JPM	TSLA	NFLX	ADBE
StockPriceBeginning	50.86	21.96	179.03	331.46	38.23	48.78	27.31	28.08	10.32	28.57
StrategyProfit	2794.00	617.00	-2873.00	23191.00	3528.00	-313.00	2472.00	11994.00	2712.00	3367.00

The strategy yields an overall profit for all the stocks except AMZN and WMT. Now we run the strategy between 2016 and 2018:

```
results_tickers = []
for ticker in tickers:
    results_tickers[ticker] = run_strategy(ticker, start = '2016-01-01', \
    end = '2018-12-31')
```

Output

	AAPL	MSFT	AMZN	GOOG	FB	WMT	JPM	TSLA	NFLX	ADBE
PerUnitStartPrice	97.95	50.26	636.99	741.84	102.22	54.97	55.84	223.41	109.96	91.97
StrategyProfit	-262.00	3324.00	67454.00	31430.00	648.00	657.00	0.00	10886.00	25020.00	12551.00

We see a good performance of the sentiment-based strategy across all the stocks except AAPL, and we can conclude that it performs quite well on different time periods. The strategy can be adjusted by modifying the trading rules or lot sizes. Additional metrics can also be used to understand the performance of the strategy. The sentiments can also be used along with the other features, such as correlated variables and technical indicators for prediction.

Conclusion

In this case study, we looked at various ways in which unstructured data can be converted to structured data and then used for analysis and prediction using tools for NLP. We have demonstrated three different approaches, including deep learning models to develop a model for computing the sentiments. We performed a comparison of the models and concluded that one of the most important steps in training the model for sentiment analysis is using a domain-specific vocabulary.

We also used a pretrained English model by spaCy to convert a sentence into sentiments and used the sentiments as signals to develop a trading strategy. The initial results suggested that the model trained on a financial lexicon-based sentiment could prove to be a viable model for a trading strategy. Additional improvements to this can be made by using more complex pretrained sentiment analysis models, such as BERT by Google, or different pretrained NLP models available in open source platforms. Existing NLP libraries fill in some of the preprocessing and encoding steps to allow us to focus on the inference step.

We could build on the trading strategy by including more correlated variables, technical indicators, or even improved sentiment analysis by using more sophisticated preprocessing steps and models based on more relevant financial text data.

Case Study 2: Chatbot Digital Assistant

Chatbots are computer programs that maintain a conversation with a user in natural language. They can understand the user's intent and send responses based on an organization's business rules and data. These chatbots use deep learning and NLP to process language, enabling them to understand human speech.

Chatbots are increasingly being implemented across many domains for financial services. Banking bots enable consumers to check their balance, transfer money, pay bills, and more. Brokering bots enable consumers to find investment options, make investments, and track balances. Customer support bots provide instant responses, dramatically increasing customer satisfaction. News bots deliver personalized current events information, while enterprise bots enable employees to check leave balance, file expenses, check their inventory balance, and approve transactions. In addition to

automating the process of assisting customers and employees, chatbots can help financial institutions gain information about their customers. The bot phenomenon has the potential to cause broad disruption in many areas within the finance sector.

Depending on the way bots are programmed, we can categorize chatbots into two variants:

Rule-based

This variety of chatbots is trained according to rules. These chatbots do not learn through interactions and may sometimes fail to answer complex queries outside of the defined rules.

Self-learning

This variety of bots relies on ML and AI technologies to converse with users. Self-learning chatbots are further divided into *retrieval-based* and *generative*:

Retrieval-based

These chatbot are trained to rank the best response from a finite set of predefined responses.

Generative

These chatbots are not built with predefined responses. Instead, they are trained using a large number of previous conversations. They require a very large amount of conversational data to train.

In this case study, we will prototype a self-learning chatbot that can answer financial questions.

This case study focuses on:

- Building a customized logic and rules parser using NLP for a chatbot.
- Understanding the data preparation required for building a chatbot.
- Understanding the basic building blocks of a chatbot.
- Leveraging available Python packages and corpuses to train a chatbot in a few lines of code.



Blueprint for Creating a Custom Chatbot Using NLP

1. Problem definition

The goal of this case study is to build a basic prototype of a conversational chatbot powered by NLP. The primary purpose of this chatbot is to help a user retrieve a financial ratio about a particular company. Such chatbots are designed to quickly retrieve the details about a stock or an instrument that may help the user make a trading decision.

In addition to retrieving a financial ratio, the chatbot could also engage in casual conversations with a user, perform basic mathematical calculations, and provide answers to questions from a list used to train it. We intend to use Python packages and functions for chatbot creation and to customize several components of the chatbot architecture to adapt to our requirements.

The chatbot prototype created in this case study is designed to understand user inputs and intention and retrieve the information they are seeking. It is a small prototype that could be enhanced for use as an information retrieval bot in banking, brokering, or customer support.

2. Getting started—loading the libraries

For this case study, we will use two text-based libraries: spaCy and ChatterBot. spaCy has been previously introduced; ChatterBot is a Python library used to create simple chatbots with minimal programming required.

An untrained instance of ChatterBot starts off with no knowledge of how to communicate. Each time a user enters a statement, the library saves the input and response text. As ChatterBot receives more inputs, the number of responses it can offer and the accuracy of those responses increase. The program selects the response by searching for the closest matching known statement to the input. It then returns the most likely response to that statement based on how frequently each response is issued by the people the bot communicates with.

2.1. Load libraries. We import spaCy using the following Python code:

```
import spacy #Custom NER model.  
from spacy.util import minibatch, compounding
```

The ChatterBot library has the modules `LogicAdapter`, `ChatterBotCorpusTrainer`, and `ListTrainer`. These modules are used by our bot in order to construct responses to user queries. We begin by importing the following:

```
from chatterbot import ChatBot
from chatterbot.logic import LogicAdapter
from chatterbot.trainers import ChatterBotCorpusTrainer
from chatterbot.trainers import ListTrainer
```

Other libraries used in this exercise are as follows:

```
import random
from itertools import product
```

Before we move to the customized chatbot, let us develop a chatbot using the default features of the ChatterBot package.

3. Training a default chatbot

ChatterBot and many other chatbot packages come with a data utility module that can be used to train chatbots. Here are the ChatterBot components we will be using:

Logic adapters

Logic adapters determine the logic for how ChatterBot selects a response to a given input statement. It is possible to enter any number of logic adapters for your bot to use. In the example below, we are using two inbuilt adapters: *Best-Match*, which returns the best known responses, and *MathematicalEvaluation*, which performs mathematical operations.

Preprocessors

ChatterBot's preprocessors are simple functions that modify the input statement a chatbot receives before the statement gets processed by the logic adapter. The preprocessors can be customized to perform different preprocessing steps, such as tokenization and lemmatization, in order to have clean and processed data. In the example below, the default processor for cleaning white spaces, `clean_whitespace`, is used.

Corpus training

ChatterBot comes with a corpus data and utility module that makes it easy to quickly train the bot to communicate. We use the already existing corpuses `english`, `english.greetings`, and `english.conversations` for training the chatbot.

List training

Just like the corpus training, we train the chatbot with the conversations that can be used for training using `ListTrainer`. In the example below, we have trained the chatbot using some sample commands. The chatbot can be trained using a significant amount of conversation data.

```

chatB = ChatBot("Trader",
    preprocessors=['chatterbot preprocessors.clean_whitespace'],
    logic_adapters=['chatterbot logic.BestMatch',
                    'chatterbot logic.MathematicalEvaluation'])

# Corpus Training
trainerCorpus = ChatterBotCorpusTrainer(chatB)

# Train based on English Corpus
trainerCorpus.train(
    "chatterbot.corpus.english"
)
# Train based on english greetings corpus
trainerCorpus.train("chatterbot.corpus.english.greetings")

# Train based on the english conversations corpus
trainerCorpus.train("chatterbot.corpus.english.conversations")

trainerConversation = ListTrainer(chatB)
# Train based on conversations

# List training
trainerConversation.train([
    'Help!',
    'Please go to google.com',
    'What is Bitcoin?',
    'It is a decentralized digital currency'
])

# You can train with a second list of data to add response variations
trainerConversation.train([
    'What is Bitcoin?',
    'Bitcoin is a cryptocurrency.'
])

```

Once the chatbot is trained, we can test the trained chatbot by having the following conversation:

```

>Hi
How are you doing?

>I am doing well.
That is good to hear

>What is 78964 plus 5970
78964 plus 5970 = 84934

>what is a dollar
dollar: unit of currency in the united states.

>What is Bitcoin?
It is a decentralized digital currency

```

```

>Help!
Please go to google.com

>Tell me a joke
Did you hear the one about the mountain goats in the andes? It was "ba a a a d".

>What is Bitcoin?
Bitcoin is a cryptocurrency.

```

In this example, we see a chatbot that gives an intuitive reply in response to the input. The first two responses are due to the training on the English greetings and English conversation corpuses. Additionally, the responses to *Tell me a joke* and *what is a dollar* are due to the training on the English corpus. The computation in the fourth line is the result of the chatbot being trained on the `MathematicalEvaluation` logical adapter. The responses to *Help!* and *What is Bitcoin?* are the result of the customized list trainers. Additionally, we see two different replies to *What is Bitcoin?*, given that we trained it using the list trainers.

Next, we move on to creating a chatbot designed to use a customized logical adapter to give financial ratios.

4. Data preparation: Customized chatbot

We want our chatbot to be able to recognize and group subtly different inquiries. For example, one might want to ask about the company *Apple Inc.* by simply referring to it as *Apple*, and we would want to map it to a ticker—AAPL, in this case. Constructing commonly used phrases in order to refer to firms can be built by using a dictionary as follows:

```

companies = {
    'AAPL': ['Apple', 'Apple Inc'],
    'BAC': ['BAML', 'BofA', 'Bank of America'],
    'C': ['Citi', 'Citibank'],
    'DAL': ['Delta', 'Delta Airlines']
}

```

Similarly, we want to build a map for financial ratios:

```

ratios = {
    'return-on-equity-ttm': ['ROE', 'Return on Equity'],
    'cash-from-operations-quarterly': ['CFO', 'Cash Flow from Operations'],
    'pe-ratio-ttm': ['PE', 'Price to equity', 'pe ratio'],
    'revenue-ttm': ['Sales', 'Revenue'],
}

```

The keys of this dictionary can be used to map to an internal system or API. Finally, we want the user to be able to request the phrase in multiple formats. Saying *Get me the [RATIO] for [COMPANY]* should be treated similarly to *What is the [RATIO] for [COMPANY]?* We build these sentence templates for our model to train on by building a list as follows:

```

string_templates = ['Get me the {ratio} for {company}',
                    'What is the {ratio} for {company}?',
                    'Tell me the {ratio} for {company}',
                    ]

```

4.1. Data construction. We begin constructing our model by creating *reverse dictionaries*:

```

companies_rev = {}
for k, v in companies.items():
    for ve in v:
        companies_rev[ve] = k
ratios_rev = {}
for k, v in ratios.items():
    for ve in v:
        ratios_rev[ve] = k
companies_list = list(companies_rev.keys())
ratios_list = list(ratios_rev.keys())

```

Next, we create sample statements for our model. We build a function that gives us a random sentence structure, inquiring about a random financial ratio for a random company. We will be creating a custom named entity recognition_ model in the spaCy framework. This requires training the model to pick up the word or phrase in a sample sentence. To train the spaCy model, we need to provide it with an example, such as (*Get me the ROE for Citi, {"entities": [(11, 14, RATIO), (19, 23, COMPANY)]}*).

4.2. Training data. The first part of the training example is the sentence. The second is a dictionary that consists of entities and the starting and ending index of the label:

```

N_training_samples = 100
def get_training_sample(string_templates, ratios_list, companies_list):
    string_template=string_templates[random.randint(0, len(string_templates)-1)]
    ratio = ratios_list[random.randint(0, len(ratios_list)-1)]
    company = companies_list[random.randint(0, len(companies_list)-1)]
    sent = string_template.format(ratio=ratio,company=company)
    ents = {"entities": [(sent.index(ratio), sent.index(ratio)+\
    len(ratio), 'RATIO'),
                          (sent.index(company), sent.index(company)+len(company), \
                          'COMPANY')]}}
    return (sent, ents)

```

Let us define the training data:

```

TRAIN_DATA = [
get_training_sample(string_templates, ratios_list, companies_list) \
for i in range(N_training_samples)
]

```

5. Model creation and training. Once we have the training data, we construct a *blank* model in spaCy. spaCy's models are statistical, and every decision they make—for example, which part-of-speech tag to assign, or whether a word is a named entity—is a prediction. This prediction is based on the examples the model has seen during training. To train a model, you first need training data—examples of text and the labels you want the model to predict. This could be a part-of-speech tag, a named entity, or any other information. The model is then shown the unlabeled text and makes a prediction. Because we know the correct answer, we can give the model feedback on its prediction in the form of an *error gradient* of the loss function. This calculates the difference between the training example and the expected output, as shown in [Figure 10-6](#). The greater the difference, the more significant the gradient, and the more updates we need to make to our model.

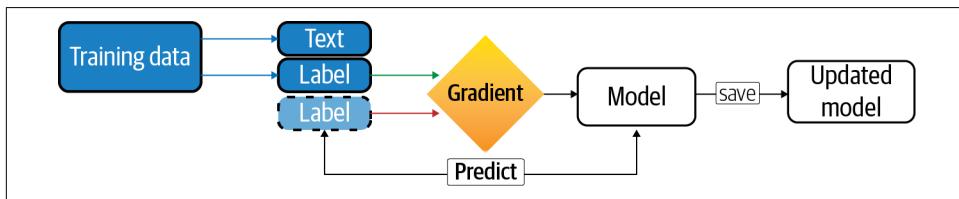


Figure 10-6. Machine learning-based training in spaCy

```
nlp = spacy.blank("en")
```

Next, we create an NER pipeline to our model:

```
ner = nlp.create_pipe("ner")
nlp.add_pipe(ner)
```

Then we add the training labels that we use:

```
ner.add_label('RATIO')
ner.add_label('COMPANY')
```

5.1. Model optimization function

Now we start optimization of our models:

```
optimizer = nlp.begin_training()
move_names = list(ner.move_names)
pipe_exceptions = ["ner", "trf_wordpiece", "trf_tok2vec"]
other_pipes = [pipe for pipe in nlp.pipe_names if pipe not in pipe_exceptions]
with nlp.disable_pipes(*other_pipes): # only train NER
    sizes = compounding(1.0, 4.0, 1.001)
    # batch up the examples using spaCy's minibatch
    for itn in range(30):
        random.shuffle(TRAIN_DATA)
        batches = minibatch(TRAIN_DATA, size=sizes)
        losses = []
```

```

for batch in batches:
    texts, annotations = zip(*batch)
    nlp.update(texts, annotations, sgd=optimizer,
               drop=0.35, losses=losses)
    print("Losses", losses)

```

Training the NER model is akin to updating the weights for each token. The most important step is to use a good optimizer. The more examples of our training data that we provide spaCy, the better it will be at recognizing generalized results.

5.2. Custom logic adapter

Next, we build our custom logic adapter:

```

from chatterbot.conversation import Statement
class FinancialRatioAdapter(LogicAdapter):
    def __init__(self, chatbot, **kwargs):
        super(FinancialRatioAdapter, self).__init__(chatbot, **kwargs)
    def process(self, statement, additional_response_selection_parameters):
        user_input = statement.text
        doc = nlp(user_input)
        company = None
        ratio = None
        confidence = 0
        # We need exactly 1 company and one ratio
        if len(doc.ents) == 2:
            for ent in doc.ents:
                if ent.label_ == "RATIO":
                    ratio = ent.text
                    if ratio in ratios_rev:
                        confidence += 0.5
                if ent.label_ == "COMPANY":
                    company = ent.text
                    if company in companies_rev:
                        confidence += 0.5
        if confidence > 0.99: (its found a ratio and company)
            outtext = '''https://www.zacks.com/stock/chart/
            /{comapny}/fundamental/{ratio} '''.format(ratio=ratios_rev[ratio]\
            , company=companies_rev[company])
            confidence = 1
        else:
            outtext = 'Sorry! Could not figure out what the user wants'
            confidence = 0
        output_statement = Statement(text=outtext)
        output_statement.confidence = confidence
        return output_statement

```

With this custom logic adapter, our chatbot will take each input statement and try to recognize a *RATIO* and/or *COMPANY* using our NER model. If the model finds exactly one *COMPANY* and exactly one *RATIO*, it constructs a URL to guide the user.

5.3. Model usage—training and testing

Now we begin using our chatbot by using the following import:

```
from chatterbot import ChatBot
```

We construct our chatbot by adding the `FinancialRatioAdapter` logical adapter that we created above to the chatbot. Although the following code snippet only shows us adding the `FinancialRatioAdapter`, note that other logical adapters, lists, and corpuses used in the prior training of the chatbot are also included. Please refer to the Jupyter notebook of the case study for more details.

```
chatbot = ChatBot(  
    "My ChatterBot",  
    logic_adapters=[  
        'financial_ratio_adapter.FinancialRatioAdapter'  
    ]  
)
```

Now we test our chatbot using the following statements:

```
converse()  
  
>What is ROE for Citibank?  
https://www.zacks.com/stock/chart/C/fundamental/return-on-equity-ttm  
  
>Tell me PE for Delta?  
https://www.zacks.com/stock/chart/DAL/fundamental/pe-ratio-ttm  
  
>What is Bitcoin?  
It is a decentralized digital currency  
  
>Help!  
Please go to google.com  
  
>What is 786940 plus 75869  
786940 plus 75869 = 862809  
  
>Do you like dogs?  
Sorry! Could not figure out what the user wants
```

As shown above, the custom logic adapter for our chatbot finds a RATIO and/or COMPANY in the sentence using our NLP model. If an exact pair is detected, the model constructs a URL to guide the user to the answer. Additionally, other logical adapters, such as mathematical evaluation, work as expected.

Conclusion

Overall, this case study provides an introduction to a number of aspects of chatbot development.

Using the ChatterBot library in Python allows us to build a simple interface to resolve user inputs. To train a blank model, one must have a substantial training dataset. In this case study, we looked at patterns available to us and used them to generate training samples. Getting the right amount of training data is usually the hardest part of constructing a custom chatbot.

This case study is a demo project, and significant enhancements can be made to each component to extend it to a wide variety of tasks. Additional preprocessing steps can be added to have cleaner data to work with. To generate a response from our bot for input questions, the logic can be refined further to incorporate better similarity measures and embeddings. The chatbot can be trained on a bigger dataset using more advanced ML techniques. A series of custom logic adapters can be used to construct a more sophisticated ChatterBot. This can be generalized to more interesting tasks, such as retrieving information from a database or asking for more input from the user.

Case Study 3: Document Summarization

Document summarization refers to the selection of the most important points and topics in a document and arranging them in a comprehensive manner. As discussed earlier, analysts at banks and other financial service organizations pore over, analyze, and attempt to quantify qualitative data from news, reports, and documents. Document summarization using NLP can provide in-depth support in this analyzing and interpretation. When tailored to financial documents, such as earning reports and financial news, it can help analysts quickly derive key topics and market signals from content. Document summarization can also be used to improve reporting efforts and can provide timely updates on key matters.

In NLP, *topic models* (such as LDA, introduced earlier in the chapter) are the most frequently used tools for the extraction of sophisticated, interpretable text features. These models can surface key topics, themes, or signals from large collections of documents and can be effectively used for document summarization.

In this case study, we will focus on:

- Implementing the LDA model for topic modeling.
- Understanding the necessary data preparation (i.e., converting a PDF for an NLP-related problem).
- Topic visualization.



Blueprint for Using NLP for Document Summarization

1. Problem definition

The goal of this case study is to effectively discover common topics from earnings call transcripts of publicly traded companies using LDA. A core advantage of this technique compared to other approaches, is that no prior knowledge of the topics is needed.

2. Getting started—loading the data and Python packages

2.1. Loading the Python packages. For this case study, we will extract the text from a PDF. Hence, the Python library *pdf-miner* is used for processing the PDF files into a text format. Libraries for feature extraction and topic modeling are also loaded. The libraries for the visualization will be loaded later in the case study:

Libraries for pdf conversion

```
from pdfminer.pdfinterp import PDFResourceManager, PDFPageInterpreter
from pdfminer.converter import TextConverter
from pdfminer.layout import LAParams
from pdfminer.pdfpage import PDFPage
import re
from io import StringIO
```

Libraries for feature extraction and topic modeling

```
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.decomposition import LatentDirichletAllocation
from sklearn.feature_extraction.stop_words import ENGLISH_STOP_WORDS
```

Other libraries

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

3. Data preparation

The `convert_pdf_to_txt` function defined below pulls out all characters from a PDF document except the images. The function simply takes in the PDF document, extracts all characters from the document, and outputs the extracted text as a Python list of strings:

```
def convert_pdf_to_txt(path):
    rsrcmgr = PDFResourceManager()
```

```

retstr = StringIO()
laparams = LAParams()
device = TextConverter(rsrcmgr, retstr, laparams=laparams)
fp = open(path, 'rb')
interpreter = PDFPageInterpreter(rsrcmgr, device)
password = ""
maxpages = 0
caching = True
pagenos=set()

for page in PDFPage.get_pages(fp, pagenos,\n    maxpages=maxpages, password=password,caching=caching,\n    check_extractable=True):\n    interpreter.process_page(page)

text = retstr.getvalue()

fp.close()
device.close()
retstr.close()
return text

```

In the next step, the PDF is converted to text using the above function and saved in a text file:

```

Document=convert_pdf_to_txt('10K.pdf')
f=open('Finance10k.txt','w')
f.write(Document)
f.close()
with open('Finance10k.txt') as f:
    clean_cont = f.read().splitlines()

```

Let us look at the raw document:

```
clean_cont[1:15]
```

Output

```

[' ',
 '',
 'SECURITIES AND EXCHANGE COMMISSION',
 '',
 '',
 'Washington, D.C. 20549',
 '',
 '',
 '\xa0',
 'FORM ',
 '\xa0',
 '',
 'QUARTERLY REPORT PURSUANT TO SECTION 13 OR 15(d) OF',
 ''
]
```

The text extracted from the PDF document contains uninformative characters that need to be removed. These characters reduce the effectiveness of our models as they provide unnecessary count ratios. The following function uses a series of regular expression (*regex*) searches as well as list comprehension to replace uninformative characters with a blank space:

```
doc=[i.replace('\xe2\x80\x9c', '') for i in clean_cont ]  
doc=[i.replace('\xe2\x80\x9d', '') for i in doc ]  
doc=[i.replace('\xe2\x80\x99s', '') for i in doc ]  
  
docs = [x for x in doc if x != ' ']  
docss = [x for x in docs if x != ' ']  
financedoc=[re.sub("[^a-zA-Z]+", " ", s) for s in docss]
```

4. Model construction and training

The `CountVectorizer` function from the `sklearn` module is used with minimal parameter tuning to represent the clean document as a *document term matrix*. This is performed because our modeling requires that strings be represented as integers. The `CountVectorizer` shows the number of times a word occurs in the list after the removal of stop words. The document term matrix was formatted into a Pandas data-frame in order to inspect the dataset. This dataframe shows the word-occurrence count of each term in the document:

```
vect=CountVectorizer(ngram_range=(1, 1),stop_words='english')  
fin=vect.fit_transform(financedoc)
```

In the next step, the document term matrix will be used as the input data to the LDA algorithm for topic modeling. The algorithm was fitted to isolate five distinct topic contexts, as shown by the following code. This value can be adjusted depending on the level of granularity one intends to obtain from the modeling:

```
lda=LatentDirichletAllocation(n_components=5)  
lda.fit_transform(fin)  
lda_dtf=lda.fit_transform(fin)  
  
sorting=np.argsort(lda.components_)[ :, ::-1]  
features=np.array(vect.get_feature_names())
```

The following code uses the `mglearn` library to display the top 10 words within each specific topic model:

```
import mglearn  
mglearn.tools.print_topics(topics=range(5), feature_names=features,  
                           sorting=sorting, topics_per_chunk=5, n_words=10)
```

Output

topic 1	topic 2	topic 3	topic 4	topic 5
-----	-----	-----	-----	-----
assets	quarter	loans	securities	value

balance	million	mortgage	rate	total
losses	risk	loan	investment	income
credit	capital	commercial	contracts	net
period	months	total	credit	fair
derivatives	financial	real	market	billion
liabilities	management	estate	federal	equity
derivative	billion	securities	stock	september
allowance	ended	consumer	debt	december
average	september	backed	sales	table

Each topic in the table is expected to represent a broader theme. However, given that we trained the model on only a single document, the themes across the topics may not be very distinct from each other.

Looking at the broader theme, topic 2 discusses quarters, months, and currency units related to asset valuation. Topic 3 reveals information on income from real estate, mortgages, and related instrument. Topic 5 also has terms related to asset valuation. The first topic references balance sheet items and derivatives. Topic 4 is slightly similar to topic 1 and has words related to an investment process.

In terms of overall theme, topics 2 and 5 are quite distinct from the others. There may also be some similarity between topics 1 and 4, based on the top words. In the next section, we will try to understand the separation between these topics using the Python library *pyLDAvis*.

5. Visualization of topics

In this section, we visualize the topics using different techniques.

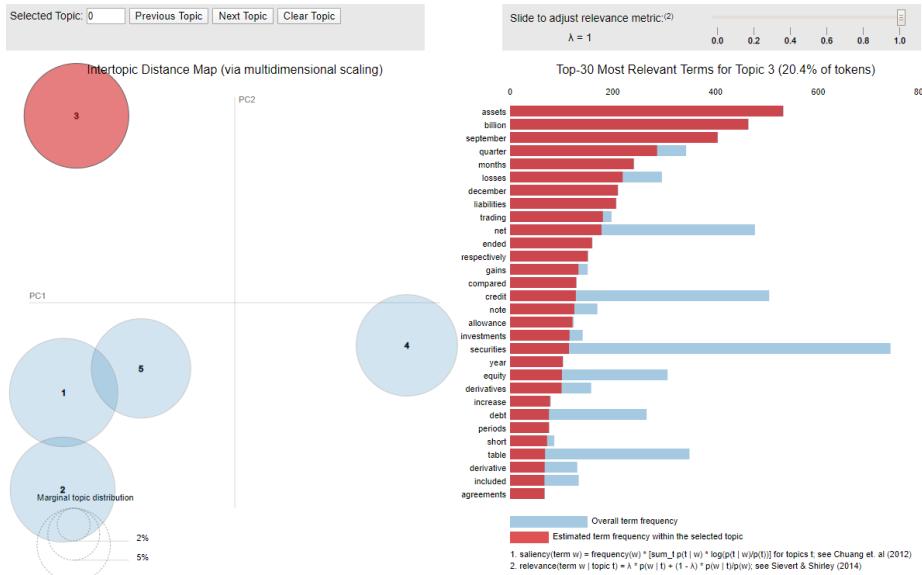
5.1. Topic visualization. *Topic visualization* facilitates the evaluation of topic quality using human judgment. *pyLDAvis* is a library that displays the global relationships between topics while also facilitating their semantic evaluation by inspecting the terms most closely associated with each topic and, inversely, the topics associated with each term. It also addresses the challenge in which frequently used terms in a document tend to dominate the distribution over words that define a topic.

Below, the *pyLDAvis*_ library is used to visualize the topic models:

```
from __future__ import print_function
import pyLDAvis
import pyLDAvis.sklearn

zit=pyLDAvis.sklearn.prepare(lda,fin,vect)
pyLDAvis.show(zit)
```

Output



We notice that topics 2 and 5 are quite distant from each other. This is what we observed in the section above from the overall theme and list of words under these topics. Topics 1 and 4 are quite close, which validates our observation above. Such close topics should be analyzed more intricately and might be combined if needed. The relevance of the terms under each topic, as shown in the right panel of the chart, can also be used to understand the differences. Topics 3 and 4 are relatively close as well, although topic 3 is quite distant from the others.

5.2. Word cloud. In this step, a *word cloud* is generated for the entire document to note the most recurrent terms in the document:

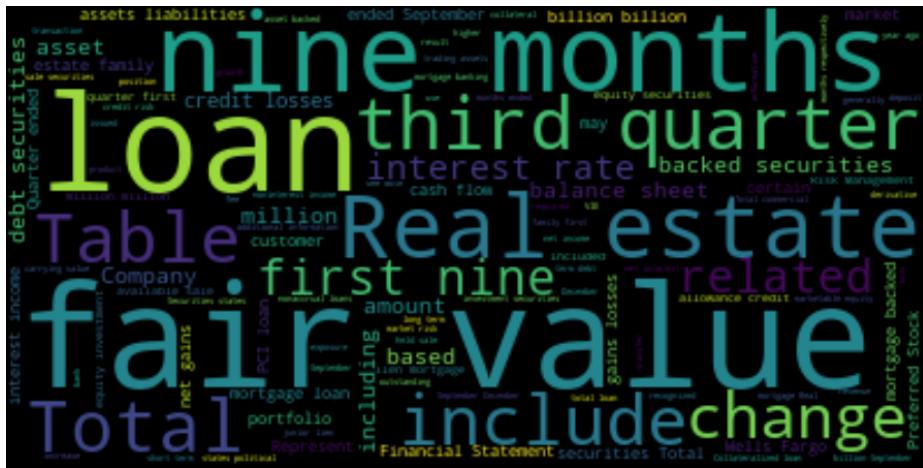
```
#Loading the additional packages for word cloud
from os import path
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
from wordcloud import WordCloud,STOPWORDS

#Loading the document and generating the word cloud
d = path.dirname(__name__)
text = open(path.join(d, 'Finance10k.txt')).read()

stopwords = set(STOPWORDS)
wc = WordCloud(background_color="black", max_words=2000, stopwords=stopwords)
wc.generate(text)
```

```
plt.figure(figsize=(16,13))
plt.imshow(wc, interpolation='bilinear')
plt.axis("off")
plt.show()
```

Output



The word cloud generally agrees with the results from the topic modeling, as recurrent words, such as *loan*, *real estate*, *third quarter*, and *fair value*, are larger and bolder.

By integrating the information from the steps above, we may come up with the list of topics represented by the document. For the document in our case study, we see that words like *third quarter*, *first nine*, and *nine months* occur quite frequently. In the word list, there are several topics related to balance sheet items. So the document might be a third-quarter financial balance sheet with all credit and assets values in that quarter.

Conclusion

In this case study, we explored the use of topic modeling to gain insights into the content of a document. We demonstrated the use of the LDA model, which extracts plausible topics and allows us to gain a high-level understanding of large amounts of text in an automated way.

We performed extraction of the text from a document in PDF format and performed further data preprocessing. The results, alongside the visualizations, demonstrated that the topics are intuitive and meaningful.

Overall, the case study shows how machine learning and NLP can be applied across many domains—such as investment analysis, asset modeling, risk management, and regulatory compliance—to summarize documents, news, and reports in order to significantly reduce manual processing. Given this ability to quickly access and verify relevant, filtered information, analysts may be able to provide more comprehensive and informative reports on which management can base their decisions.

Chapter Summary

The field of NLP has made significant progress, resulting in technologies that have and will continue to revolutionize how financial institutions operate. In the near term, we are likely to see an increase in NLP-based technologies across different domains of finance, including asset management, risk management, and process automation. The adoption and understanding of NLP methodologies and related infrastructure are very important for financial institutions.

Overall, the concepts in Python, machine learning, and finance presented in this chapter through the case studies can be used as a blueprint for any other NLP-based problem in finance.

Exercises

- Using the concepts from case study 1, use NLP-based techniques to develop a trading strategy using Twitter data.
- In case study 1, use the word2vec word embedding method to generate the word vectors and incorporate it into the trading strategy.
- Using the concepts from case study 2, test a few more logical adapters to the chatbot.
- Using the concepts from case study 3, perform topic modeling on a set of financial news articles for a given day and retrieve the key themes of the day.

Index

A

accounting fraud, early use of NLP to detect, [347](#)
accuracy (evaluation metric), [77](#)
action
 in reinforcement learning, [284](#)
 in trading, [287](#)
action-value function (Q-value), [286](#)
AdaGrad (adaptive gradient algorithm), [39](#)
Adam (adaptive moment estimation), [39](#)
adaptive boosting (AdaBoost), [68](#)
 advantages and disadvantages, [69](#)
 hyperparameters, [69](#)
 implementation in Python, [69](#)
adaptive gradient algorithm (AdaGrad), [39](#)
adjusted R² metric, [76](#)
affinity propagation clustering, [242](#)
 grouping investors, [264](#)
 pairs trading, [254](#)
agent (RL system)
 definition, [9](#), [284](#)
 trading and, [287](#)
agent class, [305-307](#)
agglomerative hierarchical clustering, [241](#)
 (see also hierarchical clustering)
AI (artificial intelligence), defined, [6](#)
algorithmic trading, [2](#)
algorithms, comparing, [24](#)
Amazon Web Services (AWS), [44](#)
Anaconda, [15](#)
ANNs (see artificial neural networks)
area under ROC curve (AUC), [78](#)
ARIMA (autoregressive integrated moving average) model, [91](#), [107](#), [110](#)

artificial intelligence (AI), defined, [6](#)
artificial neural networks (ANNs), [31-45](#)
 alternatives to running on CPU, [43](#)
 ANN-based supervised learning models, [71](#)
 architecture, [32-34](#)
 creating in Python, [40-44](#)
 derivative pricing with, [122](#)
 derivatives hedging with, [321-325](#)
 hyperparameters, [36-40](#)
 layers, [33](#)
 neurons, [32](#)
 reinforcement learning and, [292](#)
 running on cloud services, [44](#)
 running on GPU, [44](#)
 training, [34-36](#)
asset price prediction, [4](#), [49](#)
AUC (area under ROC curve), [78](#)
autocorrelation, [88](#)
automated trading systems (see algorithmic trading)
automation, finance and, [3](#)
autoregressive integrated moving average (ARIMA) model (see ARIMA)

B

backpropagation, [35-36](#)
backtesting
 defined, [298](#)
 eigen portfolio, [214-216](#)
 hierarchical risk parity, [275-276](#)
backtrader, [378](#)
bag of words model, [356](#)
bagging (bootstrap aggregation), [65](#)
batch size, [40](#)

- Bellman equations, 288
Bellman optimality equation, 288
bias, defined, 73
bias-variance trade-off, 73
binary (decision) tree, 63
bitcoin trading: enhancing speed and accuracy, 227-236
 data preparation, 228-230
 evaluation of algorithms and models, 230-235
loading data and Python packages, 228
bitcoin trading: strategy, 179-190
 data preparation, 181-185
 data visualization, 184
 evaluation of algorithms and models, 185-187
 exploratory data analysis, 181
 feature engineering, 182-184
 finalizing the model, 187-190
 loading data and Python packages, 180
 model tuning and grid search, 187
Black-Scholes formula, 114
Black-Scholes model
 for call option price, 115
 reinforcement learning-based derivatives
 hedging compared to, 325-332
bond market, yield curve and, 141
boosting, 65
- C**
- CART (see classification and regression trees)
charge-off, 167
chatbots, 4, 383-393
 data preparation for customized chatbot, 388
 loading libraries, 385
 model creation and training, 390-392
 NLP case study, 383-393
 rule-based versus self-learning, 384
 training a default chatbot, 386-388
ChatterBot, 386
classification
 defined, 8
 evaluation metrics, 77
 regression versus, 8, 49
classification and regression trees (CART), 63-65
 advantages and disadvantages, 65
 derivative pricing with, 122
- hyperparameters, 65
implementation in Python, 65
learning a CART model, 63
pruning the tree, 64
representation, 63
stopping criterion, 64
- clustering, 9, 237-277
 affinity propagation, 242
 defined, 9
 dimensionality reduction versus, 237
 hierarchical, 240-242
 hierarchical risk parity (see hierarchical risk parity)
 investors (see clustering investors)
 k-means, 239
 pairs trading and (see pairs trading)
 techniques, 239-243
clustering investors, 259-267
 affinity propagation, 264
 cluster intuition, 265
 data preparation, 262
 evaluation of algorithms and models, 263-265
 exploratory data analysis, 261
 k-means clustering, 263-264
 loading data and Python packages, 261
cointegration, 257-259
comparison of algorithms, 24
comparison of models, 24
Conda, 15
confusion matrix, 78
cost functions, 38, 52
credit (see loan default probability)
credit card underwriting, 3
cross validation, 74
cross-entropy (log loss), 39
cumulative discounted reward, 285
- D**
- dashboard, for robo-advisor, 138-140
data cleaning, steps in, 21
data preparation, steps in, 21
data science, defined, 7
data transformation, in model development, 23
data visualization, 19
data, sample code for loading, 17
datasets, splitting training/testing, 24
decision (binary) tree, 63

decision tree classifiers (see classification and regression trees (CART))
deep learning
 ANNs and, 31
 defined, 6
 reinforcement learning and, 292
 RNNs and, 92
 time series modeling with, 92-94
deep neural networks
 ANN-based supervised learning models, 72
 defined, 34
deep Q-network (DQN), 296
dendograms, 241
deploying a model, 28
derivative pricing, 4, 114-125
 data generation, 117
 data preparation/analysis, 120
 defining functions and parameters, 116
 evaluation of models, 120-125
 exploratory data analysis, 118
 loading data and Python packages, 116-118
 removing volatility data, 123
 tuning/finalizing model, 122
derivatives hedging, 316-333
 evaluation of algorithms and models, 321-325
 exploratory data analysis, 320
 generating data, 319
 loading Python packages, 319
 policy gradient script, 321-324
 testing the data, 325-332
 training the data, 325
descriptive statistics, in model development, 18
deterministic policy, 285
dimensionality reduction, 195-236
 bitcoin trading (see bitcoin trading: enhancing speed and accuracy)
 clustering versus, 237
 defined, 8, 195
 kernel PCA, 201
 principal component analysis (PCA), 198-201
 t-distributed stochastic neighbor embedding, 202
 techniques, 197-202
 yield curve construction (see yield curve construction/interest rate modeling)
direct policy search, 293
discounting factor, 285

divisive hierarchical clustering, 241
document summarization, 393-400
 data preparation, 394-396
 loading data and Python packages, 394
 model construction and training, 396
 visualization of topics, 397-399
document term matrix, 396
DQN (deep Q-network), 296
drift, 330

E

Eigen decomposition, 199
eigen portfolio, 202-217
 backtesting, 214-216
 data preparation, 205
 evaluation of algorithms and models, 207-216
 exploratory data analysis, 204
 loading data and Python packages, 203
 Sharpe ratio to determine best portfolio, 210-213
elastic net (EN), 56, 106
elbow method (k-means clustering), 247
ensemble models (supervised learning), 65-71
 adaptive boosting, 68
 extra trees, 68
 gradient boosting method, 70
 random forest, 66-67

environment

 reinforcement learning, 284
 trading and, 287
epoch, 40
epsilon greedy algorithm, 307
epsilon variable, 313
error gradient, 390
Euclidean distance, 60
evaluation metrics, identifying, 24
evaluation of models, steps in, 23
event return, 367
experience replay, 296, 307
exploratory data analysis, in model development, 18-20
extra trees (extremely randomized trees), 68

F

feature engineering, 179
feature representation in NLP, 356-360
 bag of words model, 356
 TF-IDF, 358

word embedding, 358
feature selection, in model development, 22
finalizing a model, steps in, 27-28
financial news, sentiment analysis of, 5
fixed income market, yield curve and, 141
forward propagation, 34
fraud detection, 3, 153-165
 data preparation, 156
 evaluation of models, 156-159
 exploratory data analysis, 155
 loading data and Python packages, 154
 model tuning, 159-165
future reward, 285

G

gamma, 313
gates, 93
generative chatbots, 384
gensim, 359
geometric Brownian motion (GBM), 114
Gini cost function, 64
Google Colaboratory, 44
gradient ascent, 298
gradient boosting method (GBM), 70
 advantages and disadvantages, 71
 hyperparameters, 70
 implementation in Python, 70
gradient descent, 35
grid search, 25, 53
gym (simulation environment), 337

H

helper functions, 308
hidden layers, 34, 36
hierarchical clustering, 240-242, 251-254
hierarchical risk parity (HRP), 267-277
 backtesting, 275-276
 building hierarchy graph/dendrogram, 270
 comparison against other asset allocation
 methods, 273
 data preparation, 269
 evaluation of algorithms and models,
 270-274
 getting portfolio weights for all types of
 asset allocation, 274
 loading data and Python packages, 269
 quasi-diagonalization, 271
 recursive bisection, 272
 stages of HRP algorithm, 271-274

HRP (see hierarchical risk parity)

hyperparameters
 activation functions, 37
 ANNs and, 36-40
 batch size, 40
 cost functions, 38
 epoch, 40
 learning rate, 36
 model tuning and, 25
 number of hidden layers and nodes, 36
 optimizers, 39

I

inertia, 239
inference (natural language processing),
 360-362
 LDA implementation, 362
 Naive Bayes approach, 360
input layer, ANN, 33
insurance underwriting, 3
intelligent investors, 128
intuition, model/variable, 28
investors, clustering (see clustering investors)

K

k-folds cross validation, 134
k-means clustering, 239
 for pairs trading, 247-251
 grouping investors, 263-264
 hyperparameters, 240
 implementation in Python, 240
k-nearest neighbors (KNN), 60
 advantages and disadvantages, 61
 hyperparameters, 61
Kaggle, running ANNs on, 44
Keras, 15, 40
kernel, 59
kernel PCA (KPCA), 201
KNN (see k-nearest neighbors)

L

L1 regularization, 55
L2 regularization, 56
lasso regression (LASSO), 55, 106
latent Dirichlet allocation (LDA), 360-362
latent semantic analysis (LSA), 360
layers, ANN, 33
LDA (latent Dirichlet allocation), 360-362

LDA (linear discriminant analysis), 62
learning rate, 36
lemmatization, 352
lexicons, defined, 373
linear (identity) function, 37
linear discriminant analysis (LDA), 62
linear regression, 52-54
 advantages and disadvantages, 54
 grid search, 53
 implementation in Python, 52
 regularized regression versus, 55
 training a model, 52
linkage method, 251
loan default probability, 166-179
 data preparation, 167-169
 evaluation of algorithms and models, 175
 feature selection, 169-175
 finalizing the model, 177
 loading data and Python packages, 167
 model tuning and grid search, 177
loan underwriting, 3
log loss (cross-entropy), 39
logistic regression, 57
 advantages and disadvantages, 58
 hyperparameters, 58
long short-term memory (LSTM), 93
 sentiment analysis, 370-372
 stock price prediction, 108
loss (cost) functions, 38, 52
LSA (latent semantic analysis), 360

M

machine learning
 defined, 6
 supervised, 7
 types of, 7-10
MAE (mean absolute error), 76
Manhattan distance, 61
Markov decision processes (MDPs), 289-292
Markowitz mean-variance portfolio optimization (see mean-variance portfolio (MVP) optimization)
Matplotlib, 14
matrix seriation, 271
MDPs (Markov decision processes), 289-292
mean absolute error (MAE), 76
mean reversion, 243
mean squared error (MSE), 38, 76
mean-variance portfolio (MVP) optimization

hierarchical risk parity versus, 267-277
weaknesses of, 334
model development
 data preparation, 21
 data transformation, 23
 exploratory data analysis, 18-20
 feature selection, 22
 finalizing the model, 27-28
 identifying evaluation metrics, 24
 loading data and packages, 17
 model tuning, 25
 performance on test set, 27
 saving and deploying, 28
 splitting training/testing datasets, 24
model evaluation, steps in, 23
model finalization, steps in, 27-28
model intuition, 28
model tuning, 25
model-based algorithms, 293
model-free algorithms, 293
momentum (MOM), 183
momentum optimizers, 39
momentum technical indicators, 182
money laundering, 5
moneyness, 116, 329
moving average, 182
MSE (mean squared error), 38, 76
multivariate plot types, Python code for, 20
MVP optimization (see mean-variance portfolio optimization)

N

naive Bayes
 NLP and, 360
 TextBlob and, 368
named entity recognition (NER), 353
natural language processing (NLP), 347-400
 chatbot digital assistant (see chatbots)
 defined, 10
 feature representation, 356-360
 inference, 360-362
 lemmatization, 352
 named entity recognition (NER), 353
 NLTK library, 349
 PoS tagging, 353
 preprocessing, 351-355
 Python packages for, 349
 spaCy library, 350, 354-355
 stemming, 352

stop words removal, 351
TextBlob library, 349
theory and concepts, 350-362
 tokenization, 351
NER (named entity recognition), 353
neuron weights, 34
neurons, 32
NLP (see natural language processing)
NLTK (Natural Language Took Kit), 349
 stemming code, 352
 stop words removal, 351
 tokenizer, 351
nonparametric models, 84
normalization, 23
NumPy, 14

0

OpenAI gym, 337
optimizers, 39
ordinary least squares (OLS) regression, 52-54
 (see also linear regression)
output layer, ANN, 34
overfitting, 54, 73

P

P (transition probability function), 286
pairs trading, 243-259
 affinity propagation, 254
 cluster evaluation, 255-257
 data preparation, 245
 defined, 237
 evaluation of algorithms and models, 247-257
 exploratory data analysis, 245
 hierarchical clustering, 251-254
 k-means clustering, 247-251
 loading data and Python packages, 244
 pairs selection, 257-259
Pandas, 14
parameter optimization, 298
parametric models, 84
part-of-speech (PoS) tagging, 353
partially observable Markov decision process (POMDP), 288
PCA (see principal component analysis)
pickle module, 28
pip, 15
policy development, 298
policy gradient, 297, 321-324

policy, defined, 285
policy-based algorithms, 293
POMDP (partially observable Markov decision process), 288
portfolio allocation, 334-344
 agent and cryptocurrency environment script, 336
 evaluation of algorithms and models, 336-341
 exploratory data analysis, 336
 loading data and Python packages, 335
 testing the data, 342-344
 training the data, 338-341
portfolio management
 clustering investors (see clustering investors)
 hierarchical risk parity (see hierarchical risk parity)
 robo-advisors and, 2
portfolio weights, 208-210
PoS (part-of-speech) tagging, 353
precision (evaluation metric), 77
predicted value, 34
principal component analysis (PCA), 198-201
 asset allocation, 207-216
 bitcoin trading speed/accuracy enhancement, 231-233
 Eigen decomposition, 199
 singular value decomposition, 200
 yield curve construction, 222-226
processing pipeline, 354
Python (generally), 13-29
 advantages of using, 13
 creating an artificial neural network model in, 40-44
 machine learning packages, 14
 model development blueprint, 16-28
 package installation, 15
 steps for model development in Python ecosystem, 15-28
Python, creating an ANN model in, 40-44
 compiling the model, 42
 evaluating the model, 43
 fitting the model, 43
 installing Keras and machine learning packages, 40
 model construction, 41

Q

Q-learning, 294
Q-value, 286
qualitative data, 348
quasi-diagonalization, 271

R

radial basis function (rbf), 201
random forest, 66-67
 advantages and disadvantages, 67
 hyperparameters, 67
 implementation in Python, 67
random under-sampling, 161-165
rate of change (ROC), 182
rbf (radial basis function), 201
recall (evaluation metric), 78
rectified linear unit (ReLU) function, 37
recurrent neural networks (RNNs), 92
 derivatives hedging with, 321
 LSTM, 93
recursive binary splitting, 63
recursive bisection, 272
regex, 366
regression, 83-150
 classification versus, 8, 49
 defined, 8
 supervised (see supervised regression)
 time series models versus, 84
regularization, defined, 55
regularized regression, 55-57
reinforcement learning (RL), 281-345
 artificial neural networks/deep learning, 292
 Bellman equations, 288
 components, 284-288
 components in trading context, 287
 deep Q-network, 296
 defined, 9
 derivatives hedging (see derivatives hedging)
 Markov decision processes, 289-292
 model definition, 286
 model-based algorithms, 293
 model-free algorithms, 293
 modeling framework, 288-293
 models, 293-298
 policy, 285
 policy gradient method, 297
 portfolio allocation (see portfolio allocation)
 Q-learning, 294
 Q-value, 286
 SARSA, 295
 shortcomings, 298
 temporal difference learning, 292
 theory and concepts, 283-283
 trading strategy (see trading strategy,
 reinforcement-learning based)
 value function, 285
relative strength index (RSI), 182
ReLU (rectified linear unit) function, 37
rescaling, 23
residual sum of squares (RSS), 52
retrieval-based chatbots, 384
reward function, 287, 301
reward, defined, 284
ridge regression, 56
risk management, 4
risk tolerance and robo-advisor case study,
 125-141
 data preparation, 128-131
 evaluation of models, 134
 feature selection, 131-134
 finalizing the model, 136
 loading data and Python packages, 127
 model tuning and grid search, 135
 robo-advisor dashboard, 138-140
risk-free assets, 128
risky assets, 128
RL (see reinforcement learning)
RMSPProp (root mean square propagation), 39
RNNs (see recurrent neural networks)
robo-advisors, 2
 (see also risk tolerance and robo-advisor
 case study)
ROC (rate of change), 182
root mean square propagation (RMSPProp), 39
root mean squared error (RMSE), 24, 76
RSI (relative strength index), 182
RSS (residual sum of squares), 52
rule-based chatbots, 384
 R^2 metric, 76

S

SARSA, 295
saving a model, 28
Scikit-learn, 14
SciPy, 14
Seaborn, 15
seasonal variation, 87

Securities and Exchange Commission (SEC), 347
self-learning chatbots, 384
sensitivity (recall), 78
sentiment analysis, 5
sentiment analysis-based trading strategies, 362-383
 data preparation, 365-368
 evaluating models for sentiment analysis, 368-377
 exploratory data analysis and comparison, 374-377
 loading data and Python packages, 365
 models evaluation, 377-383
 results for individual stocks, 378-381
 results for multiple stocks, 381
 setting up a strategy, 378
supervised learning: classification algorithms and LSTM, 370-372
unsupervised learning: model based on financial lexicon, 373-374
 varying the strategy time period, 382
Sequential model (Keras), 41
Sharpe ratio, 210-213
sigmoid function, 37, 300
silhouette method (k-means clustering), 248
simulation environment, 337
singular value decomposition (SVD), 200, 231-233
sklearn, 14, 60, 67
spaCy, 350, 354-355
 importing, 385
 word embedding via, 359
standardization, 23
state (definition), 284, 300
state (value) function, 285
stationary time series, 89
StatsModels, 15
stemming, 352
stochastic oscillators, 182
stochastic policy, 285
stock price prediction, 95-113
 data preparation, 103
 data visualization, 100-102
 descriptive statistics, 100
 evaluation of models, 103-110
 exploratory data analysis, 100-103
 features useful for, 96
 finalizing the model, 112
 loading data, 99
 loading Python packages, 98
 model tuning and grid search, 110-112
 time series analysis, 102
sum of squared residuals (RSS), 52
sum squared error, 64
supervised classification, 151-191
 bitcoin trading strategy (see bitcoin trading: strategy)
 fraud detection (see fraud detection)
 loan default probability (see loan default probability)
 selecting an evaluation metric for, 79
supervised learning, 49-82
 ANN-based models, 71
 classification, 151-191
 (see also classification)
 classification and regression trees, 63-65
 classification metrics, 77
 cross validation, 74
 defined, 7, 49
 ensemble models, 65-71
 evaluation metrics, 75-79
 factors for model selection, 79
 hyperparameters, 62
 k-nearest neighbors, 60
 linear discriminant analysis, 62
 linear regression, 52-54
 logistic regression, 57
 model performance, 73-79
 model selection, 79-82
 model trade-off, 81
 overfitting/underfitting, 73
 overview of models, 51-73
 regression, 8
 regularized regression, 55-57
 selecting evaluation metric for supervised regression, 76
 support vector machine, 58-60
supervised regression
 time series models versus, 84
 yield curve prediction (see yield curve prediction)
support vector machine (SVM), 58-60
 advantages and disadvantages, 60
 hyperparameters, 60
SVD (singular value decomposition), 200, 231-233

T

t-distributed stochastic neighbor embedding (t-SNE), 202, 233
tanh function, 37
temporal difference (TD) learning, 292
 Q-learning and, 294
 SARSA, 295
TensorFlow, 15
testing datasets, 24, 27
TextBlob, 349, 368-370
TF-IDF (term frequency-inverse document frequency), 358
Theano, 15
time series (defined), 86
time series analysis (defined), 84
time series models
 ARIMA, 91
 autocorrelation, 88
 basics, 86-95
 deep learning approach to modeling, 92-94
 differencing, 90
 modifying data for supervised learning model, 95
 stationarity, 89
 supervised regression versus, 84
 time series components, 87
 traditional models, 90-92
time series prediction, 72
tokenization, 351
topic modeling, LDA for, 362
topic visualization, 397
trade settlement, 5
trading strategy, reinforcement-learning based, 298-316
 agent class, 305-307
 data preparation, 303
 evaluation of algorithms and models, 303-314
 exploratory data analysis, 302
 helper functions, 308
 implementation steps and modules, 304
 loading data and Python packages, 301
 model tuning, 313
 testing the data, 314-315
 training the model, 308-313
training datasets, 24
training, for ANNs, 34-36
 backpropagation, 35-36
 forward propagation, 34

transition probability function (P), 286
trend, defined, 87
true positive rate (recall), 78
truncated SVD, 200, 231-233
tuning a model, 25

U

under-sampling, 161-165
underfitting, 73
underwriting, 3
univariate plot types, Python code for, 19
unsupervised learning
 defined, 8, 195
 LDA for topic modeling, 362

V

VADER (Valence Aware Dictionary for Sentiment Reasoning), 373
value function, 285
 Q-value and, 286
variable intuition, 28
variance, defined, 73
visualization of data, 19
volatility, 116

W

weights, neuron, 34
word cloud, 398
word embedding, 358
 via spaCy, 359
 via word2vec model, 359

Y

yield curve construction/interest rate modeling
 case study, 217-227
 data preparation, 220
 data visualization, 219-220
 evaluation of algorithms and models, 222-226
exploratory data analysis, 219-220
 loading data and Python packages, 218
yield curve prediction, 141-149
 data visualization, 144-146
 evaluation of models, 146
 exploratory data analysis, 144-146
 loading data and Python packages, 143
 model tuning and grid search, 147-149

About the Authors

Hariom Tatsat currently works as a vice president in the Quantitative Analytics Division of an investment bank in New York. Hariom has extensive experience as a quant in the areas of predictive modeling, financial instrument pricing, and risk management in several global investment banks and financial organizations. He completed his MS at UC Berkeley and his BE at IIT Kharagpur (India). Hariom has also completed FRM (Financial Risk Manager) certification and CQF (Certificate in Quantitative Finance) and is a candidate for CFA Level 3.

Sahil Puri works as a quantitative researcher. His work involves testing model assumptions and finding strategies for multiple asset classes. Sahil has applied multiple statistical and machine learning-based techniques to a wide variety of problems. Examples include generating text features, labeling curve anomalies, nonlinear risk factor detection, and time series prediction. He completed his MS at UC Berkeley and his BE at Delhi College of Engineering (India).

Brad Lookabaugh works as a VP of Portfolio Management at Union Home Ownership Investors, a real estate investment startup based in San Francisco. His work focuses on the implementation of machine learning and investment decision models in business processes, internal systems, and consumer-facing products. Similar to his coauthors, Brad completed his MS in financial engineering at UC Berkeley.

Colophon

The animal on the cover of *Machine Learning and Data Science Blueprints for Finance* is the common quail (*Coturnix coturnix*), a migratory bird that breeds in Europe, Turkey, and central Asia to China, wintering in parts of southeast Asia and across the continent of Africa.

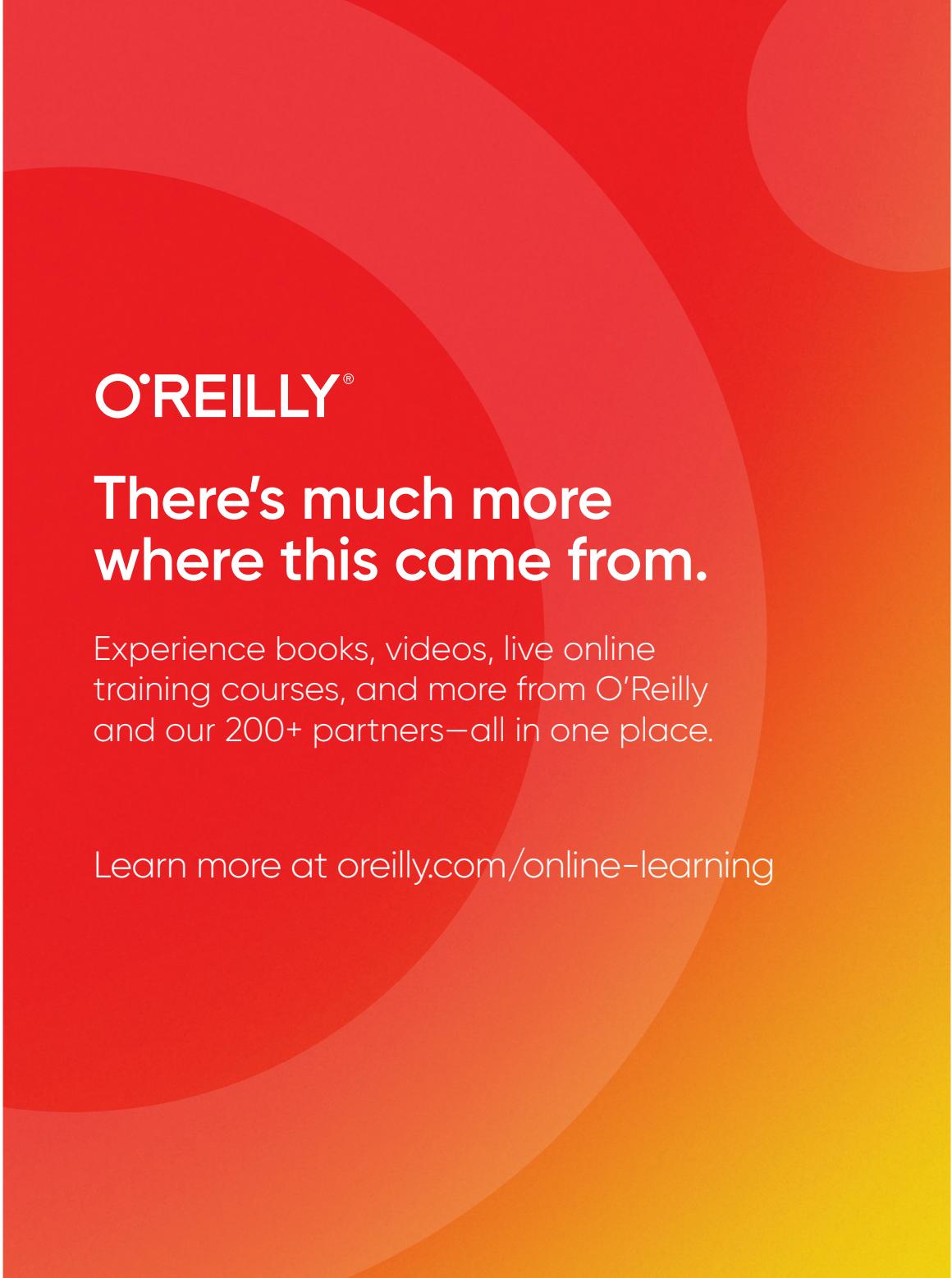
The common quail is small and round, streaked brown with a white eyestripe and, in males, a white chin. It has long wings to benefit its migratory nature, and is several inches in length. Its size and plumage allow it to blend well into its environment. Its appearance in combination with its secretive nature means the common quail is rarely seen and more often detected by the male's emphatic whistle.

Common quail consume mostly seeds, grains, and nuts, but females require a high protein diet for breeding, dining on beetles, ants, and earwigs among other small insects. Whether pecking at wind-scattered seeds or insects, the quail feeds mainly on the ground. Though it travels great migratory distances, the common quail is reluctant to fly even when disturbed.

The common quail has been depicted in Egyptian hieroglyphs dating back to circa 5000 B.C.E. and have been raised for human consumption since the construction of the Great Pyramids. Common quail in Europe lay up to 13 eggs per clutch, and chicks can fly when they are just 11 days old.

While the common quail's conservation status is currently listed as of least concern, industrial scale trapping is driving the species into decline. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from *Shaw's Zoology*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.



O'REILLY®

There's much more where this came from.

Experience books, videos, live online training courses, and more from O'Reilly and our 200+ partners—all in one place.

Learn more at oreilly.com/online-learning