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

While the author has used their best efforts in preparing this book, they make no representations or warranties with the respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. It is sold on the understanding that the author is not engaged in rendering professional services and the author shall not be liable for damages arising herefrom. If professional advice or other expert assistance is required, the services of a competent professional should be sought.

Part I

Introduction

Chapter 1

Introduction To Advanced Algorithmic Trading

In this introductory chapter we will consider why we want to adopt a fully statistical approach to quantitative trading and discuss Bayesian Statistics, Time Series Analysis and Machine Learning.

In addition we will look at what the book contains, the technical background you will need to get the most out of the book, why it differs from the previous book *Successful Algorithmic Trading* and choices of backtesting software for the strategies we will discuss at the end of the book.

1.1 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. This coupled to the availability of financial data, as provided by services such as Yahoo Finance, Google Finance, Quandl and IQ Feed, has lead to a sharp increase in individuals learning how to become a quantitative trader.

However, many of these individuals never get past learning basic "technical analysis" and so avoid important topics such as risk management, portfolio construction and algorithmic execution. In addition they often neglect more effective means of generating alpha, such as can be provided via detailed statistical analysis.

In this book I want to provide a "next step" for those who have already begun their algorithmic trading career, or are looking to try more advanced methods. In particular, we will be making use of 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 I can assure you that not only is it extremely interesting when applied to real world data, but it will provide you with a solid "mental framework" for how to think about all of your future trading methods and approaches.

Obviously, statistical analysis is a huge field of academic interest. Trying to distill the topics important for quantitative trading is difficult. However, there are three main areas that we will concentrate on in this book:

- Bayesian Statistics
- Time Series Analysis
- Machine Learning

Each of these three areas has its place in quantitative finance.

1.1.1 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, our own 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 we can update our beliefs 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. In our case, we will be studying financial asset prices in order to predict future values or understand why they change. The Bayesian framework provides us with a modern, sophisticated 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 we understand the basics of how Bayesian Statistics is carried out, particularly in relation to the Markov Chain Monte Carlo method, which will we discuss at length in the book section on Bayesian Statistics.

To carry out Bayesian Inference in this book we will use a "probabilistic programming" tool, written in Python, called **PyMC**.

1.1.2 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 studying the tools in time series analysis we can make elementary assessments of financial asset behaviour and use this to consider more advanced methods, in a structured way.

The main idea in Time Series Analysis is that of *serial correlation*. Briefly, in terms of daily trading prices, serial correlation describes to us 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.

Time Series Analysis can be thought of as a much more rigourous approach to understanding the behaviour of financial asset prices than "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, including hypothesis testing, goodness-of-fit tests and model selection, all of which serve to help us rigourously determine asset behaviour and thus eventually increase our profitability of our strategies. We can understand trends, seasonality, long-memory effects and volatility clustering in much more detail.

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

1.1.3 Machine Learning

Machine Learning is another subset of statistical learning that applies modern statistical models, across huge data sets, whether they have a temporal component or not. Machine Learning is part of the broader "data science" and quant ecosystem.

Machine Learning is generally subdivided into two separate categories, namely *supervised learning* and *unsupervised learning*. The former uses "training data" to train an algorithm to detect patterns in data. The latter has no concept of training (hence the "unsupervised") and algorithms solely act on the data without being penalised or rewarded for correct answers.

We will be using machine learning techniques such as Support Vector Machines and Random Forests to find more complicated relationships between differing sets of financial data. If these patterns can be successfully validated then we can use them to infer structure in the data and 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 we will use the Python **scikit-learn** library, as well as **pandas**, for data analysis.

1.2 How Is The Book Laid Out?

The book is broadly laid out in four sections. The first three are theoretical and teach you the basics through to intermediate usage of Bayesian Statistics, Time Series Analysis and Machine Learning. The fourth section applies all of the previous theory to real trading strategies.

The book begins with a discussion on the Bayesian philosophy of statistics and uses the binomial model 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 key examples such as switch-point analysis (for regime detection) and stochastic volatility. Finally, we conclude by discussing the burgeoning area of Bayesian Econometrics.

In Time Series Analysis we begin by discussing the concept of Serial Correlation, before applying it to simple models such as White Noise and the Random Walk. From these two models we can build up more sophisticated approaches to explaining Serial Correlation, culminating in the Autoregressive Integrated Moving Average (ARIMA) family of models.

We then move on to consider volatility clustering, or *conditional heteroskedasticity*, and define and utilise the Generalised Autoregressive Conditional Heteroskedastic (GARCH) family of models.

Subsequent to ARIMA and GARCH we will consider long-memory effects in financial time series, take a deeper look at cointegration (for statistical arbitrage) and consider approaches to state space models including Hidden Markov Models and Kalman Filters.

All the while we will be applying these time series models to current financial data and assessing how they perform in terms of inference and prediction.

In the Machine Learning section we will begin with a more rigourous definition of supervised and unsupervised learning, and then discuss the notation and methodology of statistical machine learning. We will use the humble linear regression as our first model, swiftly moving on to linear classification with logistic regression, linear discriminant analysis and the Naive Bayes Classifier.

We will then be ready to consider the more advanced non-linear methods such as Support Vector Machines and Random Forests. We will consider unsupervised techniques such as Principal Components Analysis, k-Means Clustering and Non-Negative Matrix Factorisation.

We will apply these techniques to asset price prediction, natural language processing and subsequently sentiment analysis.

Finally we will discuss where to go from here. There are plenty of academic topics of interest to review, including Non-Linear Time Series Methods, Bayesian Nonparametrics and Deep Learning using Neural Networks. However, these topics will have to wait for later books!

1.3 Required Technical Background

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

1.3.1 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, you do not have to had a university education in order to use this book. There are plenty of fantastic resources for learning these topics on the internet. I prefer:

- 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, Bayesian Statistics, Time Series Analysis and Machine Learning *are* quantitative subjects. There is no avoiding the fact that we will be using some intermediate mathematics to quantify our ideas.

I recommend the following courses for helping you get up to scratch with your 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

1.3.2 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 of you will likely have programmed in VB Script or VB.NET, through Excel. I would strongly recommend taking some introductory Python and R programming courses if this is the case as it will teach you about deeper programming topics that will be utilised in this book.

Here are some useful courses:

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

1.4 How Does This Differ From "Successful Algorithmic Trading"?

Successful Algorithmic Trading was written primarily to help readers think in rigourous quantitative terms about their trading. In 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, machine learning and Bayesian stats, as applied to rigourous quantitative trading strategies across multiple asset classes.

Hence this book is largely theoretical for the first three sections and then highly practical for the fourth, where we discuss the implementation of actual trading strategies.

I have added far more strategies to this book than in the previous version and this should give you a solid idea in how to continue researching and improving your own strategies and trading ideas.

This book is *not* a book that covers extensions of the event-driven backtester, nor does it dwell on software-specific testing methodology or how to build an institutional-grade infrastructure system. It is primarily about quantitative trading *strategies* and how to carry out research into their profitability.

1.5 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. In this section I'll describe how to easily install Python and R.

1.5.1 Installing Python

In order to follow the code for the Bayesian Statistics and Machine Learning chapters you will need to install a Python environment.

Possibly the easiest 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 all of the necessary libraries you need to get going with the code in this book.

Once installed you will have access to the Spyder Integrated Development Environment (IDE), which provides a Python syntax-highlighting text editor, an IPython console for interactive workflow and visualisation, and an object/variable explorer for helpful debugging.

All of the code in the Python sections of this book has been designed to be run using Anaconda/Spyder for both Python 2.7.x and 3.4.x+, but will also happily work in "vanilla" Python virtual environments, once the necessary libraries have been installed.

If you have any questions about Python installation, please email me at mike@quantstart.com.

1.5.2 Installing R

R is a little bit tricker to install than Anaconda, but not massively so. I make use of an IDE for R, known as R Studio. This provides a similar interface to Anaconda, in that you get an R syntax-highlighting console and visualisation tools all in the same document interface.

R Studio requires R itself, so you must first download R before using R Studio. This can be done for Windows, Mac OS X or Linux from the following link: https://cran.rstudio.com/

You'll want to select the pre-compiled binary from the top of the page that fits your particular operating system.

Once you have successfully installed R, the next step (if desired!) is to download R Studio: https://www.rstudio.com/products/rstudio/download/

Once again, you'll need to pick the version for your particular platform and operating system type (32/64-bit). You need to select one of the links under "Installers for Supported Platforms".

All of the code in the R sections of this book has been designed to be run using "vanilla" R and/or R Studio.

If you have any questions about R installation, please email me at mike@quantstart.com.

1.6 Backtesting Software Options

These days, there are a myriad of options for carrying out backtests and new software (both open source and proprietary) appears every month.

I have decided to explain the strategies in simple terms and to concentrate predominantly on the mathematical techniques. We will make use of *vectorised* (that is, non event-driven) systems purely for reasons of speed and ease of implementation.

Python, via the pandas library, and R, both allow straightforward vectorised backtesting, which can give us a good first-order approximation to how well a strategy is likely to do in production.

Hence all performance figures will be derived on the basis of these vectorised backtests and we will discuss how much of an impact transaction costs are likely to have on performance.

1.6.1 Alternatives

There are many alternative backtesting environments available and I strongly encourage you to code up these strategies in more realistic environments if you wish to trade them in a live environment. In particular, you could consider:

- QSForex - My own open-source high-frequency event-driven backtester for the Forex market using the OANDA brokerage: https://www.quantstart.com/qsforex
- 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

1.7 What Do You Get In The Rough Cut Version?

Firstly, I'd like to thank you for pre-ordering the book in its 'rough cut' state. It is immensely valuable to me - and subsequently current and future readers - to have a continual process of feedback while the book is being finished. For *C++ For Quantitative Finance* and *Successful Algorithmic Trading* I was able to incorporate many suggestions that came directly from readers of the site and the books.

Since this is a pre-order 'rough cut' release of *Advanced Algorithmic Trading*, not all of the topics mentioned in the website ebook page will be available at this point in time. However, the book is continually being written and so as new content is produced it will be added to the 'rough cut', prior to its full release early in 2016.

I have endeavoured to make sure that the Time Series Analysis section is nearly complete. It currently covers White Noise, Random Walks, ARMA, ARIMA, GARCH and State-Space Models. It is currently not covering Multivariate Models, Cointegration, Long-Memory Effects or Market Microstructure. However, the material covered up to the GARCH model already provides a very useful introduction to Time Series Analysis for those who have not considered it before. The remaining sections will be added in later releases.

The Bayesian Statistics section currently contains discussion on the basics of Bayesian Inference and the analytical approach to inference on binomial proportions. This is sufficient material necessary to understand the later material on time series. It is currently not covering Markov Chain Monte Carlo techniques, Switch-Point Analysis, Stochastic Volatility or further Bayesian Econometric tools. These will be added in later releases.

The Machine Learning section currently discusses two of the major issues in Supervised Learning, namely the Bias-Variance Tradeoff and k-Fold Cross-Validation. It also discusses our first advanced machine learning technique, namely the Support Vector Machine. It currently does not cover a broad introduction to Supervised and Unsupervised Learning, Linear Regression, Linear Classification, Kernel Density Estimation, Tree-Based Methods, Unsupervised Learning and Natural Language Processing. These will be added in later releases.

The Quantitative Trading Strategies section currently only has a strategy based primarily on the material from the Time Series section, namely the combined ARIMA+GARCH predictive model. It currently does not cover High Frequency Bid-Ask Spread Prediction, Asset Returns Forecasting using Machine Learning techniques, Kalman Filters for Pairs Trading, Volatility Forecasting or Sentiment Analysis. These strategies, and more, will be added in later releases.

If there are any topics that you think would be particularly suitable for the book, then please email me at mike@quantstart.com and I'll do my best to try and incorporate them prior to the final release.

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 the two 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?

The following table describes the alternative philosophies of the frequentist and Bayesian approaches:

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.

Thus in the Bayesian interpretation probability is a *summary of an individual's opinion.* A key point is that different (rational, intelligent) individuals can have different opinions (and thus different prior beliefs), since they have differing access to data and ways of interpreting it. However, as both of these individuals come across new data that they both have access to, their (potentially differing) prior beliefs will lead to posterior beliefs that will begin converging 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 have no confidence in 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 following box, we derive Bayes' rule 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.

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 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 via the concept of *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 emphasises 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 clustered 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 $\theta = 0.5$.

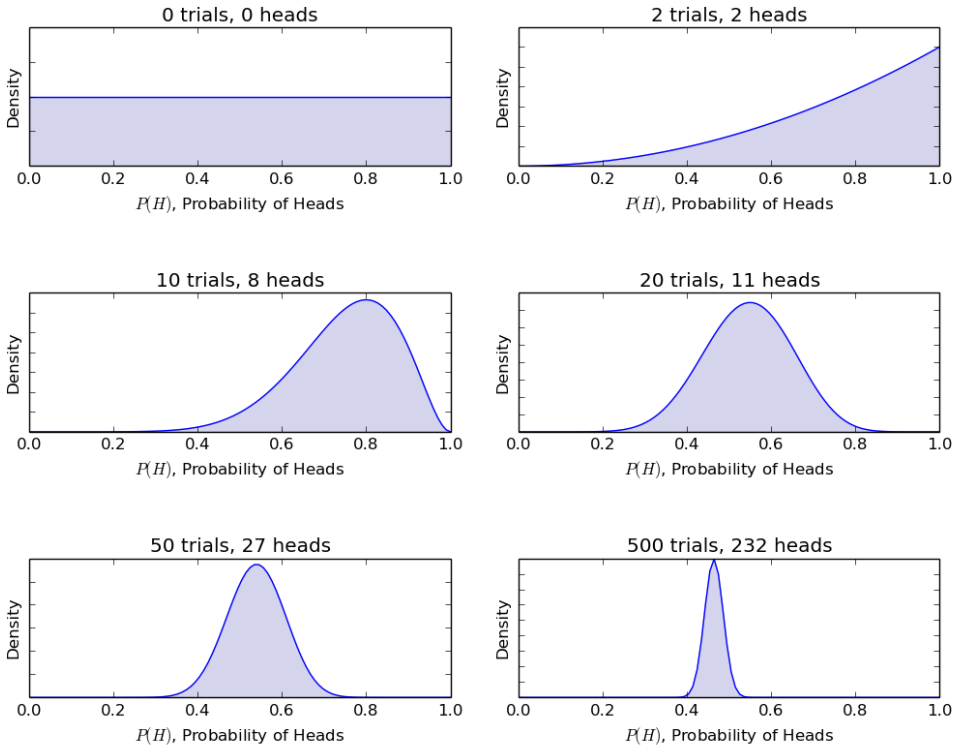

Figure 2.1: Bayesian update procedure using the Beta-Binomial Model

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 the specific techniques as we go while 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 given by:

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

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 question then becomes - which probability distribution do we use 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.

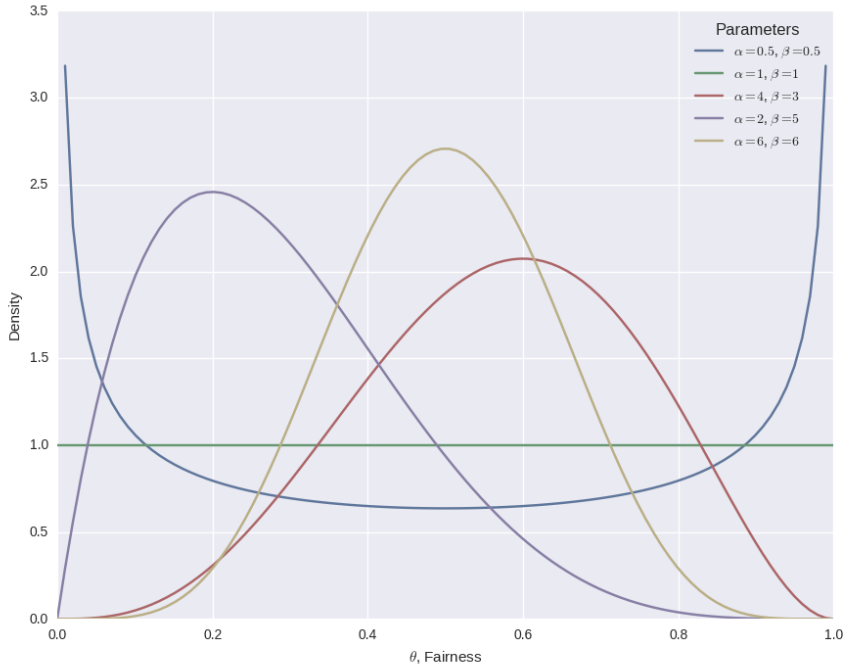

Figure 3.1: Different realisations of the beta distribution for various parameters α and β.

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 the right (a coin biased to come up heads more often), whereas an increase in β moves the distribution towards the left (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 $\theta = 0.5$, i.e. when the coin is far.

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

- **Support** - It's 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, we immediately know 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.

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.

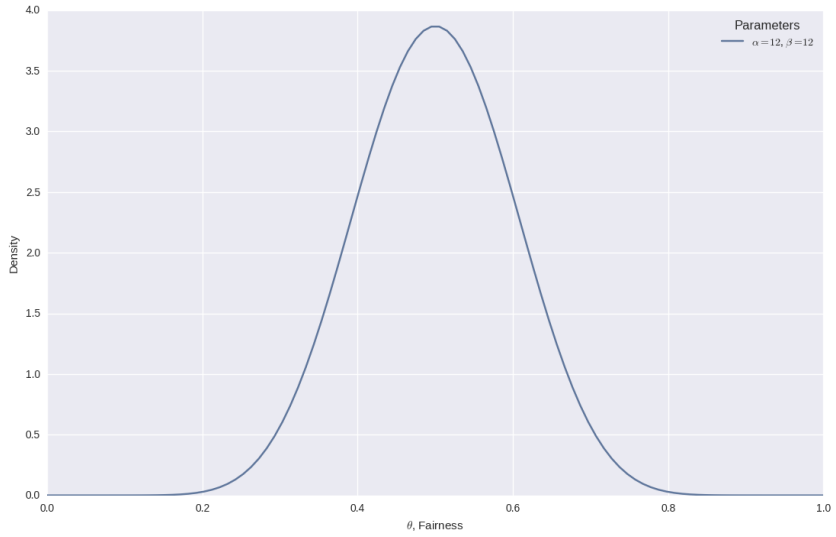

Figure 3.2: A beta distribution with $\alpha = 12$ and $\beta = 12$.

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 and 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.

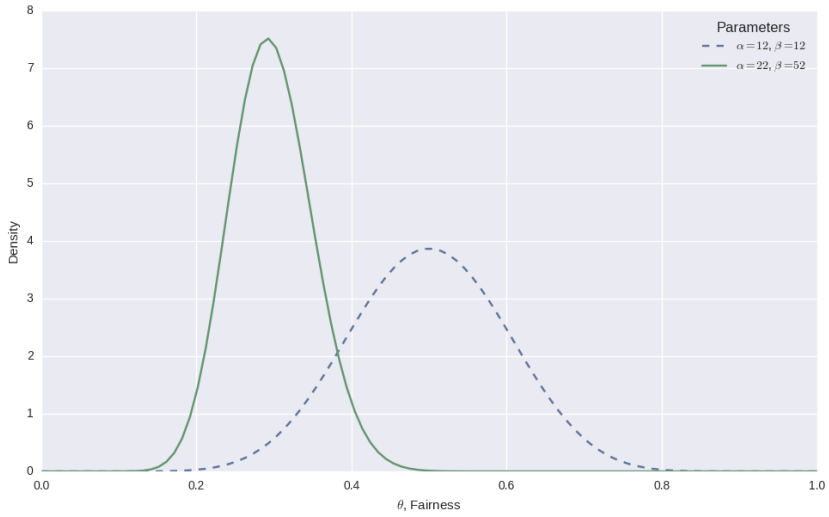

Figure 3.3: The prior and posterior belief distributions about the fairness θ.

Notice how the peak shifts dramatically to the left 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:

$$\begin{aligned} \mu_{\text{post}} &= \frac{\alpha}{\alpha+\beta} && (3.24)\\ &= \frac{22}{22+52} && (3.25)\\ &= 0.297 && (3.26)\\ & && (3.27) \end{aligned}$$

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

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

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 discussed the fact 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 and Hamiltonian Monte Carlo.

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

4.1 Bayesian Inference Goals

Our goal in carrying out Bayesian Statistics is to *produce quantitative trading strategies based on Bayesian models*. However, 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 computation of the posterior distribution in the case of inference on a binomial proportion

In this chapter specifically 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. In particular, many more complicated modelling situations, particularly those related to

hierarchical models with hundreds of parameters, which we will consider in later chapters, 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 θ, the parameters:

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

The fundamental problem is that we are often unable to evaluate this integral analytically and so we must turn to a numerical approximation method 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. This in turn means that our posterior distributions will also be high dimensional. Hence, 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 becomes extremely sparse within that space and hence leads to problems of statistical significance. Practically, in order to gain any statistical significance, the volume of data needed 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 consider Metropolis-Hastings, the Gibbs Sampler, Hamiltonian MCMC and the No-U-Turn Sampler (NUTS). The latter is actually incorporated into PyMC3, the software we'll be using to numerically infer our binomial propoertion in this chapter.

4.3 The Metropolis Algorithm

The first MCMC algorithm considered in this chapter is due to Metropolis[36], which was developed in 1953. Hence it is not a recent method! While there have been substantial improvements on MCMC sampling algorithms since, it will suffice for this section. The intuition gained on this simpler method will help us understand more complex samplers in later sections and chapters.

The basic recipes for most MCMC algorithms tend to follow this pattern (see Davidson-Pilon[19] 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 position and takes a "proposal width" for its standard deviation σ.

This proposal width is a 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 miss a region of higher probability initially. However, a smaller proposal width won't cover as much of the space as quickly and thus 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[50] points out in his article on MCMC sampling, we're 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[] is a Python library (currently in beta) that carries out "Probabilistic Programming". That is, we can define a probabilistic model 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.

It also makes use of the Python Theano[] library, often used for highly CPU/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, which is sufficient to express the main ideas, without getting bogged down in MCMC implementation specifics. In later chapters we will explore more features of PyMC3 once we come to carry 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 θ a mean $\mu = 0.5$ and a standard deviation $\sigma = 0.1$.

A beta distribution has two parameters, α and β, that characterise the "shape" of our beliefs. A mean $\mu = 0.5$ and s.d. $\sigma = 0.1$ 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:

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 Binonial Proportion with PyMC3

We're 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 --process-dependency-links git+https://github.com/pymc-devs/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:

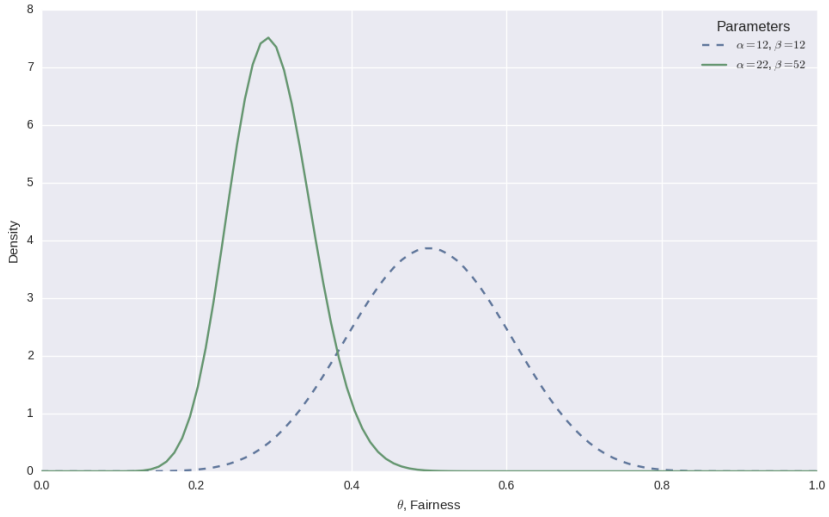

Figure 4.1: The prior and posterior belief distributions about the fairness θ

```
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 our 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 (we will go into detail about this in later chapters). 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 almost 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:

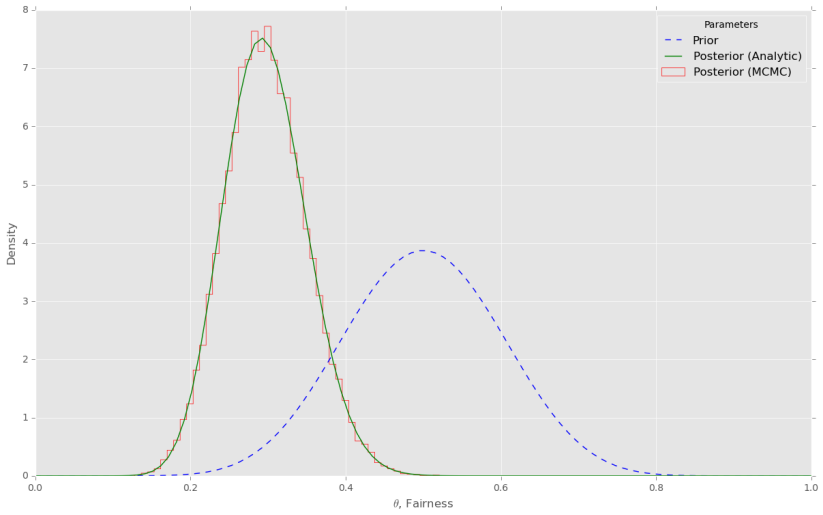

Figure 4.2: Comparison of the analytic and MCMC-sampled posterior belief distributions about the fairness θ, overlaid with the prior belief

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'd expect. In a relatively simple model such as this we do not need to compute 100,000 samples and far fewer would do. However, it does emphasise the convergence 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.

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**). We will discuss the trace, burn in and other convergence issues in later sections when we study more sophisticated samplers. 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:

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

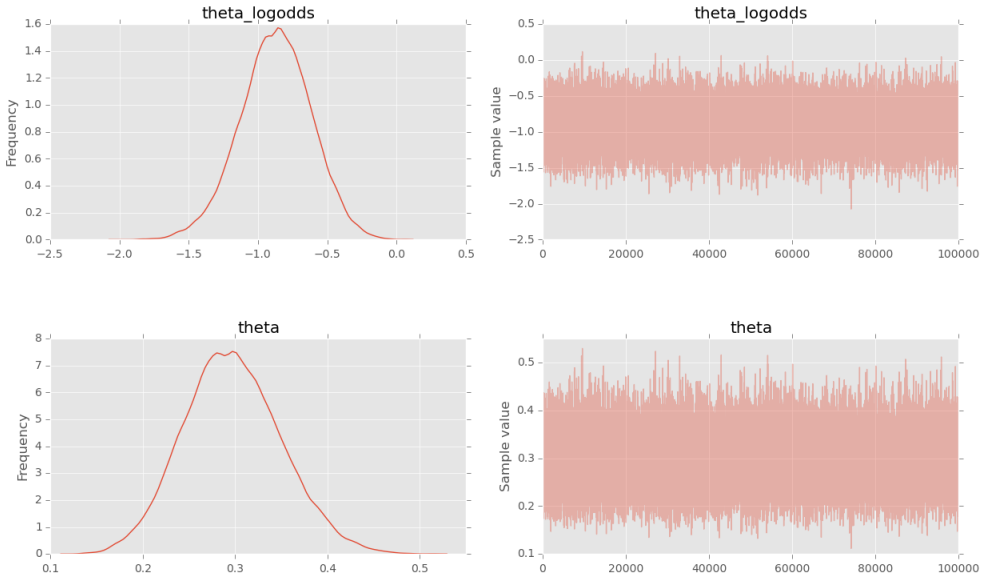

Figure 4.3: Trace plot of the MCMC sampling procedure for the fairness parameter θ

MCMC sampling procedure has "converged to the distribution" since the sampling series looks stationary.

In more complicated cases, which we will examine in later sections, we will see that we need to consider a "burn in" period as well as "thin" the results to remove autocorrelation, both of which will improve convergence.

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 Next Steps

At this stage we have a good understanding of the basics behind MCMC, as well as a specific method known as the Metropolis algorithm, as applied to inferring a binomial proportion.

However, as we discussed above, PyMC3 uses a much more sophisticated MCMC sampler known as the No-U-Turn Sampler (NUTS). In order to gain an understanding of this sampler we eventually need to consider further sampling techniques such as Metropolis-Hastings, Gibbs Sampling and Hamiltonian Monte Carlo (on which NUTS is based).

We also want to start applying Probabilistic Programming techniques to more complex models, such as hierarchical models. This in turn will help us produce sophisticated quantitative

trading strategies.

4.7 Bibliographic Note

The algorithm described in this chapter is due to Metropolis[36]. An improvement by Hastings[28] led to the Metropolis-Hastings algorithm. The Gibbs sampler is due to Geman and Geman[24]. Gelfand and Smith[22] 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[20] and the No-U-Turn Sampler (NUTS) is due to Hoffman and Gelman[29]. Gelman et al[23] 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[34].

A very popular on-line introduction to Bayesian Statistics is by Cam Davidson-Pilon and others[19], which has a fantastic chapter on MCMC (and PyMC3). Thomas Wiecki has also written a great blog post[50] explaining the rationale for MCMC.

The PyMC3 project[2] 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 the basics of 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 (which will be discussed at length in the book part on Statistical Machine Learning 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's now turn our attention to the frequentist approach to linear regression.

5.1 Frequentist Linear Regression

The frequentist, or classical, approach to multiple linear regression assumes a model of the form[27]:

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

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:

$$\text{RSS}(\beta) = \sum_{i=1}^{N}(y_i - f(x_i))^2 \tag{5.2}$$

$$= \sum_{i=1}^{N}(y_i - \beta^T x_i)^2 \tag{5.3}$$

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[27] 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*. This means that it is a single value in $\mathbb{R}^{p+1}$. 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. That is, we reformulate the above linear regression model to use probability distributions. The syntax for a linear regression in a Bayesian framework looks like this:

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

In words, our response datapoints $\mathbf{y}$ 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 a very different formulation to the frequentist approach. In the frequentist setting 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[**?**]:

- **Prior Distributions:** If we have any prior knowledge about the parameters β, then we can choose prior distributions that reflect this. If we don't, 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 doesn't really give us much clue as to how to *specify* a model and sample from it using Markov Chain Monte Carlo. In the next few sections we will use PyMC3 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 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. Thus it helps us gain intuition into how the model works. The second reason is that it allows us to see how the model performs (i.e. the values and uncertainty it returns) 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, use seaborn to plot it, and ultimately use the Generalised Linear Models (GLM) module of PyMC3 to formulate a Bayesian linear regression and sample from it, on our 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 we'll be using these 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.

<P>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/outcome, $\mathbf{y}$, via a "link function", and is assumed to be generated from a statistical distribution from the exponential distribution family. This family of 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:

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

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[50] 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 (this is modified and extended from Jonathan Sedar's post[?]):

```
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

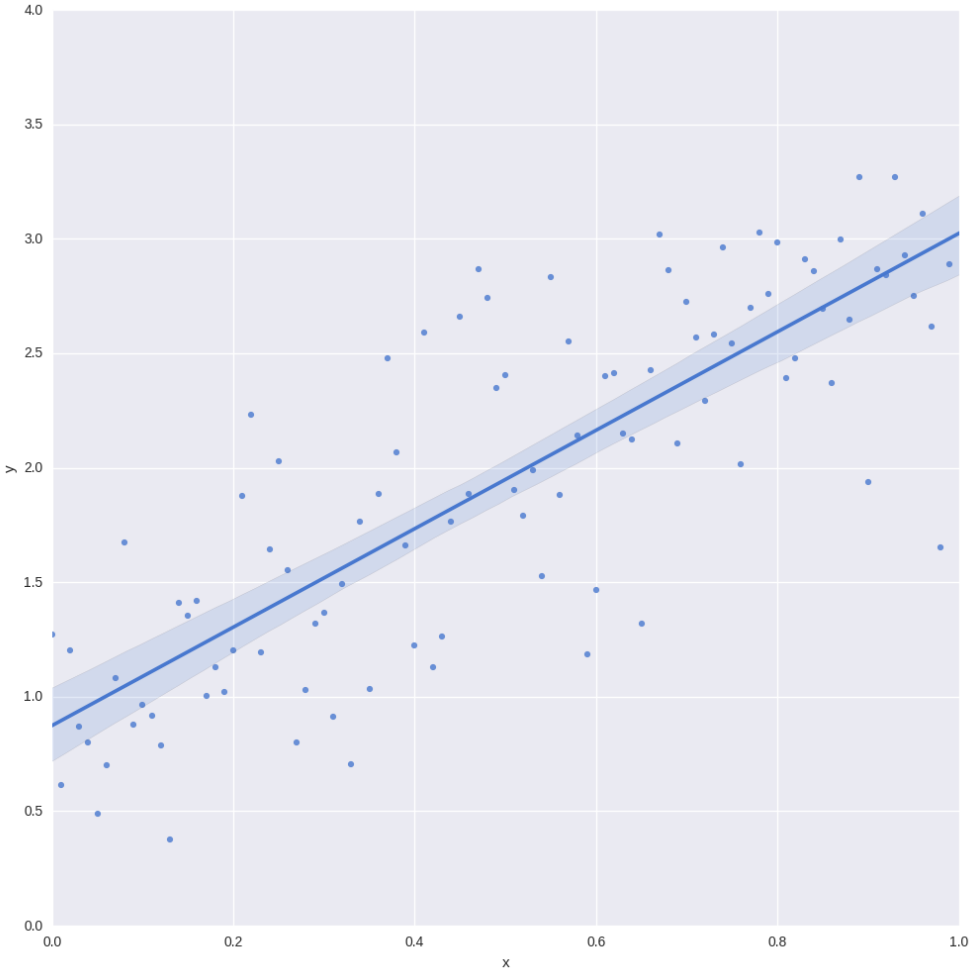

Figure 5.1: Simulation of noisy linear data via Numpy, pandas and seaborn

similar to how R 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 from. Finally, we are going to use the No-U-Turn Sampler[29] 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):
    """
    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 a little off compared 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.

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`).

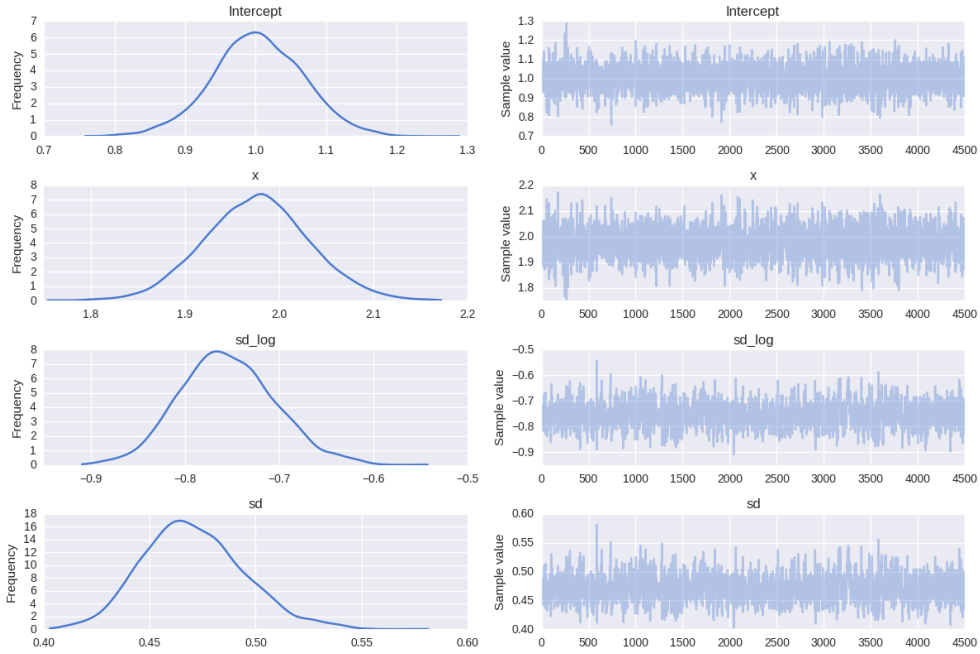

Figure 5.2: Using PyMC3 to fit a Bayesian GLM linear regression model to simulated data

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:

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 Next Steps

The next step is to begin discussion of *robust regression* and *hierarchical linear models*, a powerful modelling technique made tractable by rapid MCMC implementations. From a quantitative

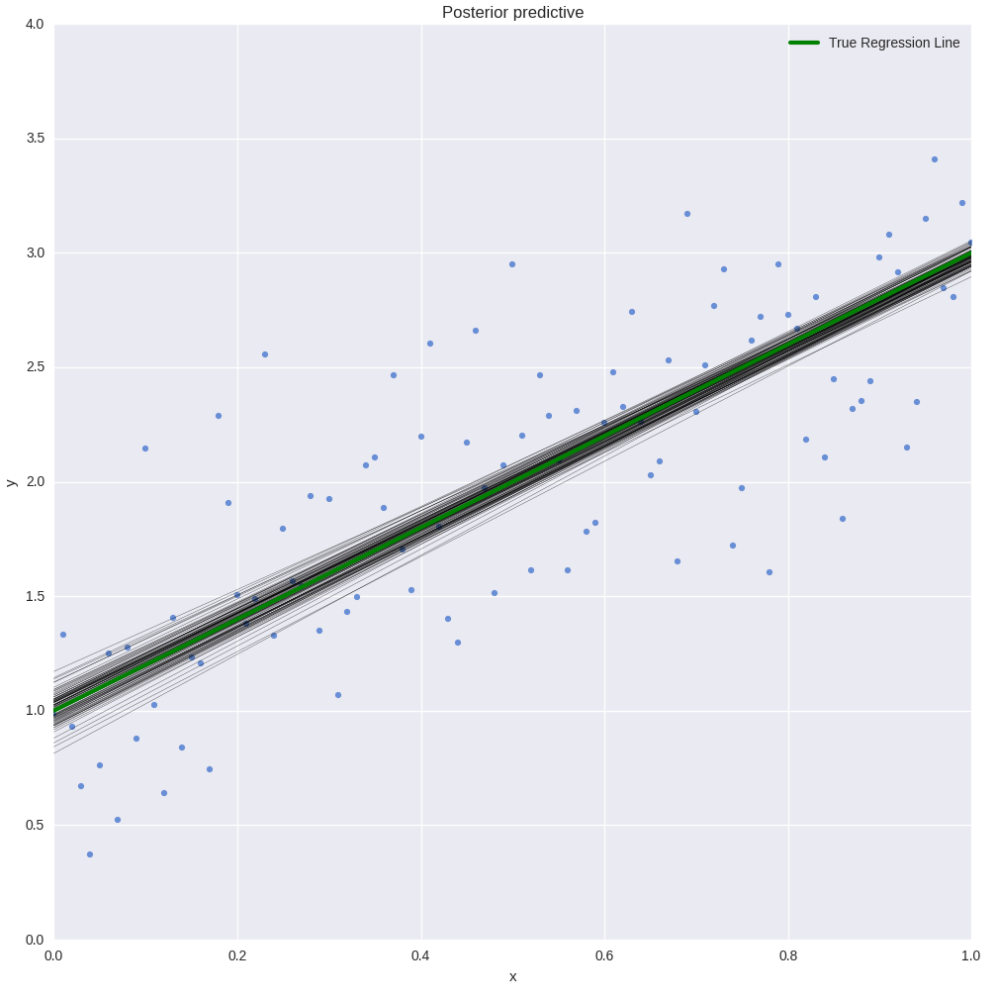

Figure 5.3: Using PyMC3 GLM module to show a set of sampled posterior regression lines

finance point of view we will also take a look at a stochastic volatility model using PyMC3 and see how we can use this model to form trading algorithms.

5.5 Bibliographic Note

An introduction to frequentist linear regression can be found in James et al[32]. A more technical overview, including subset selection methods, can be found in Hastie et al[27]. Gelman et al[23] discuss Bayesian linear models in depth at a reasonably technical level.

This chapter is influenced from previous blog posts by Thomas Wiecki[50] including his discussion of Bayesian GLMs[48, 49] as well as Jonathan Sedar with his posts on Bayesian Inference with PyMC3[44].

5.6 Full Code

```
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()
```

Part III

Time Series Analysis

Chapter 6

Introduction to Time Series Analysis

In this chapter we are going to introduce methods from the field of **time series analysis**. These techniques are extremely important in quantitative finance as the vast majority of asset modelling in the financial industry still makes 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.

6.1 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.

6.2 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 standard (classical/frequentist or Bayesian) statistical tests to our time series models in order to justify certain behaviours, such as regime change in equity markets.

6.3 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 stack. 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 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.

6.4 Time Series Analysis Roadmap

We have previously discussed Bayesian statistics and that 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 high-frequency data across different markets.

- **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.
- **Multivariate Modeling** - We have considered multivariate models in *Successful Algorithmic Trading*, namely when we considered mean-reverting pairs of equities. In this chapter we will more rigourously define *cointegration* and look at further tests for it. We will also consider vector autoregressive (VAR) models [not to be confused with Value-at-Risk!].
- **State-Space Models** - State Space Modelling borrows a long history of modern control theory used in engineering in order to allow 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. This will be one of the major uses of Bayesian analysis in time series.
- **Market Microstructure** - We will consider high frequency trading and examine market microstructure effects in detail, later applying our knowledge to high-frequency forex data.

6.5 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, and now as a consultant to funds, 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 that can trade at high-frequency.

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

Chapter 7

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're 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**.

7.1 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 7.1.1. 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 7.1.2. 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 7.1.3. 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 7.1.4. 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.

However 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:

$$\text{Cov}(x, y) = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y}) \tag{7.1}$$

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.

7.1.1 Example: Sample Covariance in R

This is actually our first usage of R 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 7.1.

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**.

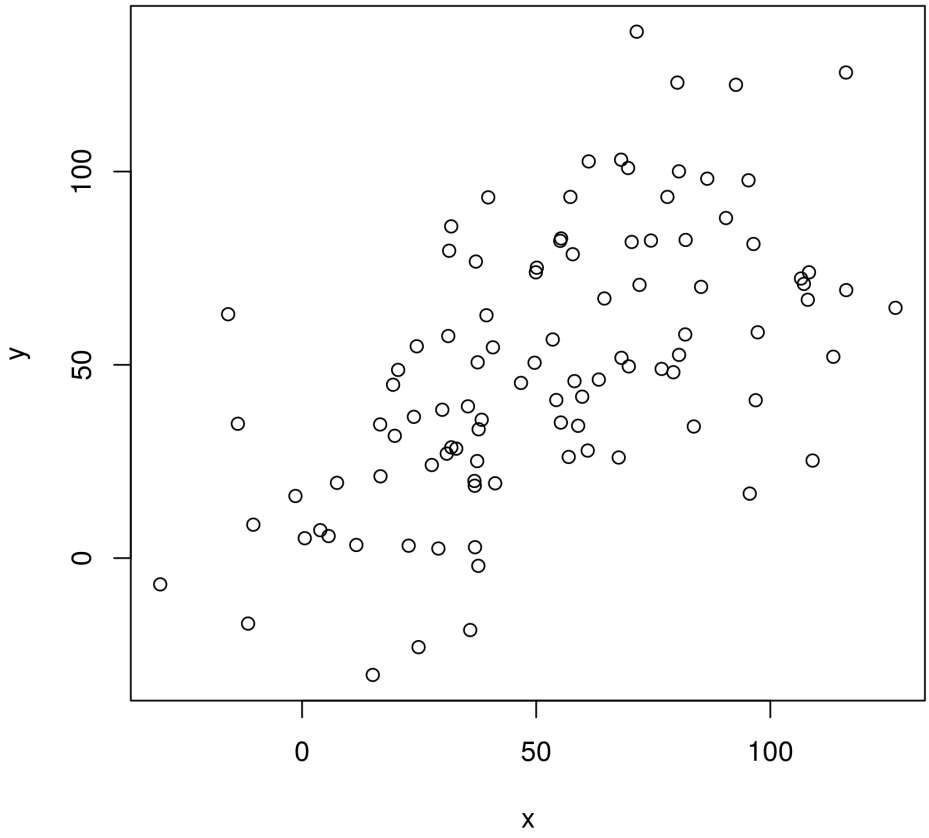

Figure 7.1: Scatter plot of two linearly increasing variables with normally distributed noise.

7.2 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 $\rho(x, y)$:

$$\rho(x, y) = \frac{E[(x - \mu_x)(y - \mu_y)]}{\sigma_x \sigma_y} = \frac{\sigma(x, y)}{\sigma_x \sigma_y} \tag{7.2}$$

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**, $\mathrm{Cor}(x, y)$:

$$\mathrm{Cor}(x, y) = \frac{\mathrm{Cov(x,y)}}{\mathrm{sd}(x)\mathrm{sd}(y)} \tag{7.3}$$

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

7.2.1 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.

7.3 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 7.3.1. 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 don't 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 7.3.2. Stationary in the Mean. A time series is *stationary in the mean* if $\mu(t) = \mu$, a constant.

Now that we've seen how we can discuss expectation values 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 7.3.3. 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:

$$\text{Var(x)} = \frac{\sum(x_t - \bar{x})^2}{n-1} \tag{7.4}$$

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 7.3.4. 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**.

7.4 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 7.4.1. 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 7.4.2. 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.

As before this motivates the definition of serial correlation or **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 7.4.3. 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:

$$c_k = \frac{1}{n}\sum_{t=1}^{n-k}(x_t - \bar{x})(x_{t+k} - \bar{x}) \tag{7.5}$$

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

$$r_k = \frac{c_k}{c_0} \tag{7.6}$$

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.

7.5 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 7.2.

```
> 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.

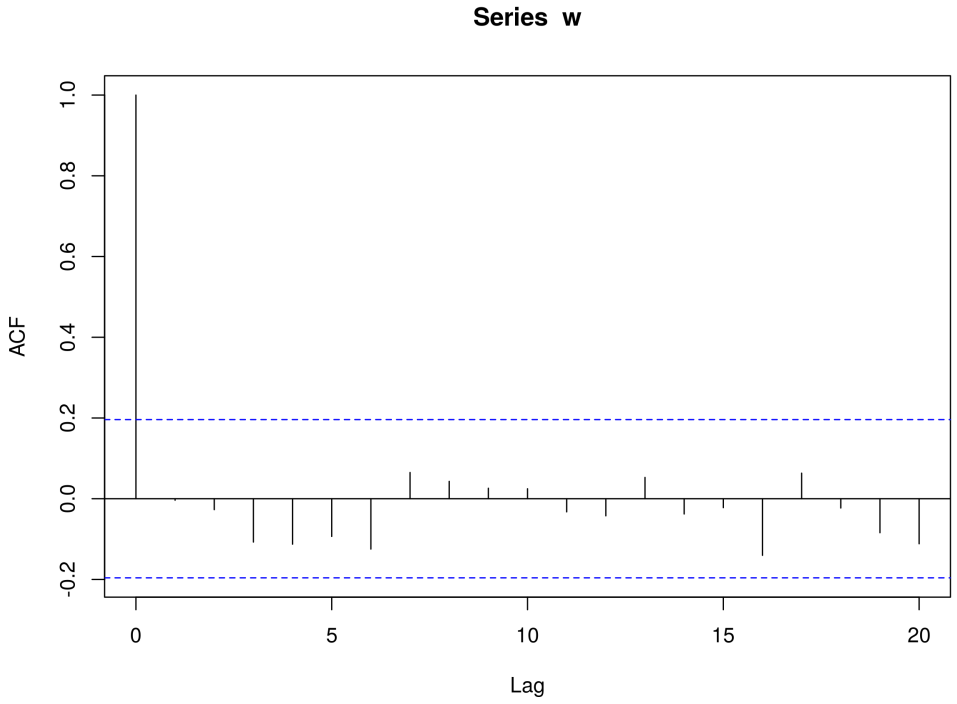

Figure 7.2: Correlogram plotted in R of a sequence of normally distributed random variables.

7.5.1 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 7.3.

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.

7.5.2 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 7.4.

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.

7.6 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**.

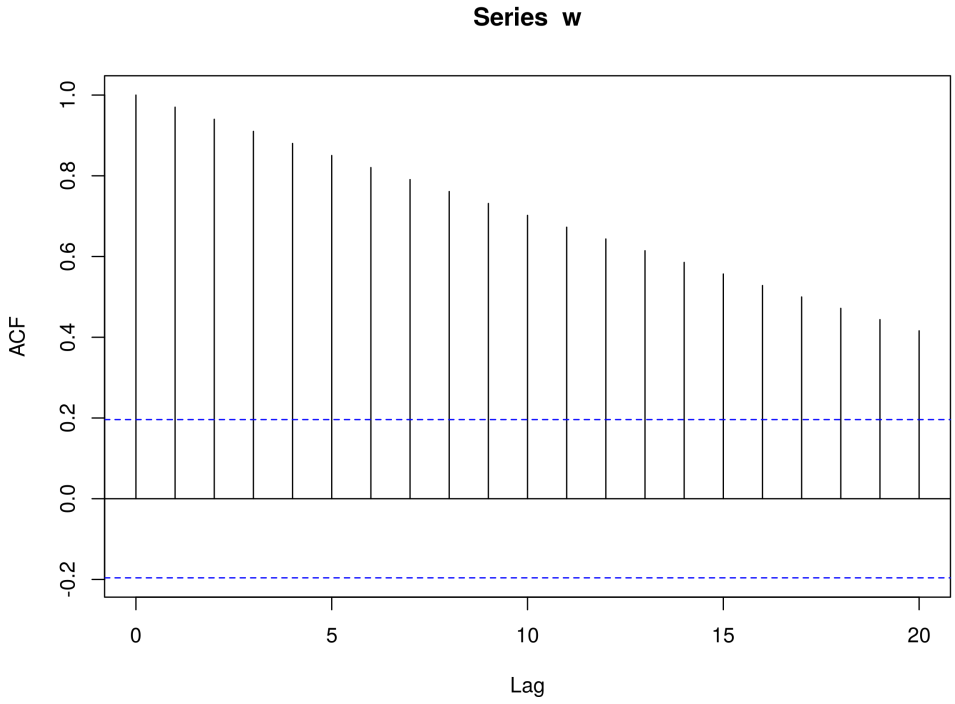

Figure 7.3: Correlogram plotted in R of a sequence of integers from 1 to 100

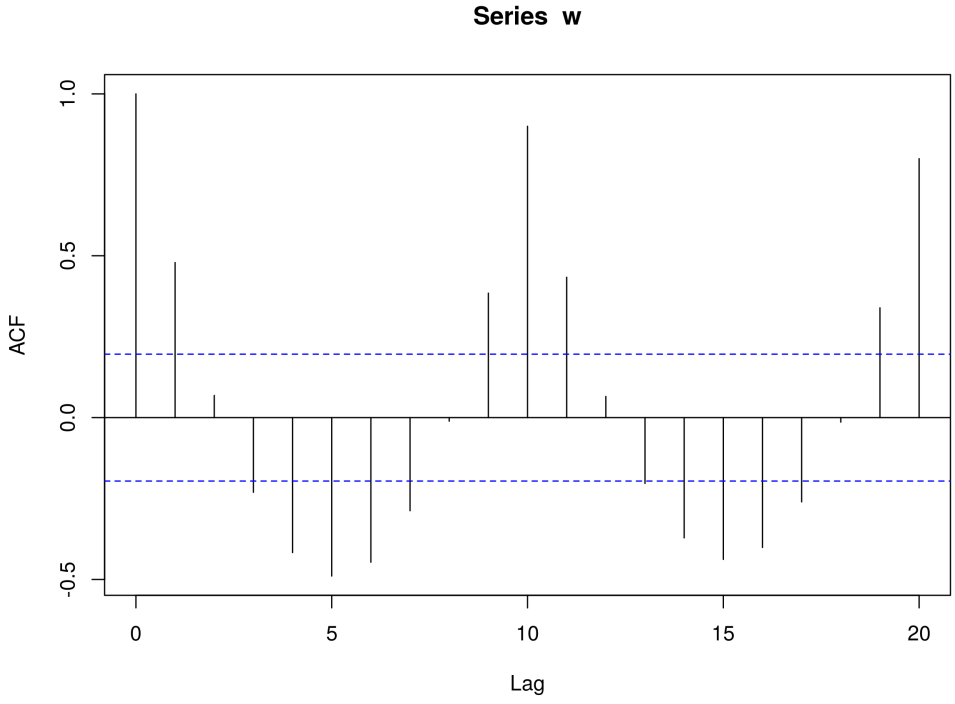

Figure 7.4: Correlogram plotted in R of a sequence of integers from 1 to 10, repeated 10 times

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 8

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.

8.1 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 (allowing for transaction costs, of course!). Also, 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 **second order properties**, since they give us the means to help us make 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's 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

That 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**.

8.2 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 8.2.1. 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 8.2.2. 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.

8.3 White Noise

Let's begin by trying to motivate 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 8.3.1. 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 8.3.2. Discrete White Noise. Consider a time series $\{w_t : t = 1, ...n\}$. If the elements of the series, w_i, 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_i 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.

As we've mentioned before, 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've defined Discrete White Noise, we are going to examine some of the attributes of it, including its **second order properties** and its correlogram.

8.3.1 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:

$$\mu_w = E(w_t) = 0 \tag{8.1}$$

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

8.3.2 Correlogram

We can also plot the correlogram of a DWN using R, see Figure 8.1. 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))
```

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've 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, namely the **Random Walk**.

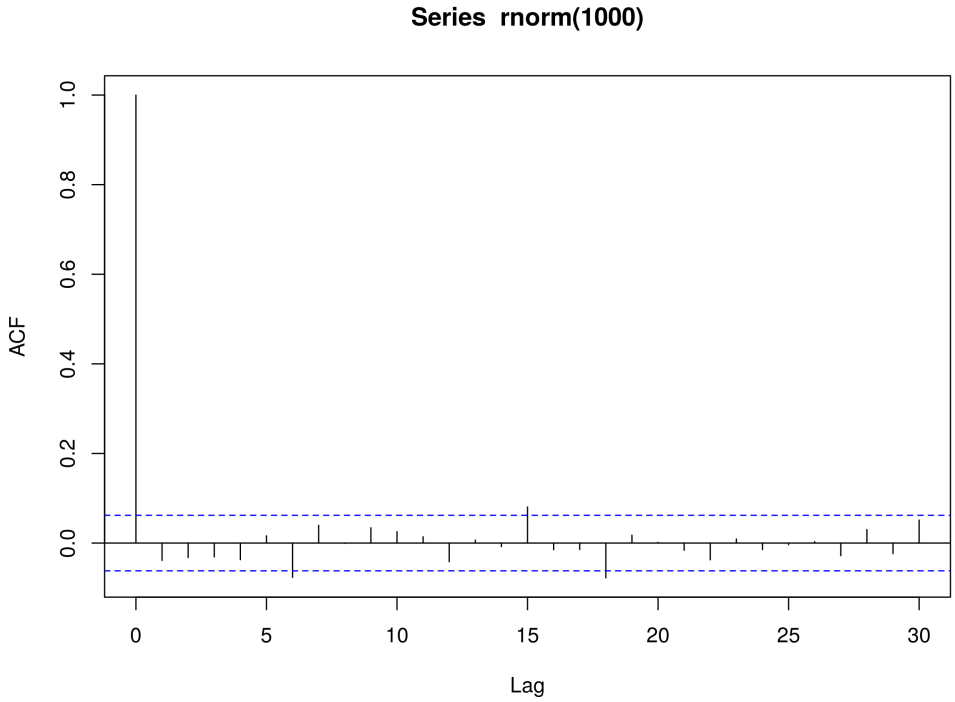

Figure 8.1: Correlogram of Discrete White Noise.

8.4 Random Walk.

A **random walk** is another 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 8.4.1. 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:

$$x_t = \mathbf{B}x_t + w_t = x_{t-1} + w_t \tag{8.2}$$

And stepping back further:

$$x_{t-1} = \mathbf{B}x_{t-1} + w_{t-1} = x_{t-2} + w_{t-1} \tag{8.3}$$

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

$$x_t = (1 + \mathbf{B} + \mathbf{B}^2 + \ldots)w_t \implies x_t = w_t + w_{t-1} + w_{t-2} + \ldots \tag{8.4}$$

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

8.4.1 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**:

$$\mu_x = 0 \tag{8.5}$$
$$\gamma_k(t) = \text{Cov}(x_t, x_{t+k}) = t\sigma^2 \tag{8.6}$$

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*.

8.4.2 Correlogram

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

$$\rho_k(t) = \frac{\text{Cov}(x_t, x_{t+k})}{\sqrt{\text{Var}(x_t)\text{Var}(x_{t+k})}} = \frac{t\sigma^2}{\sqrt{t\sigma^2(t+k)\sigma^2}} = \frac{1}{\sqrt{1+k/t}} \tag{8.7}$$

Notice that this implies 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 8.2.

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

It is simple enough to draw the correlogram too, see Figure 8.3.

```
> acf(x)
```

8.4.3 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've 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, such that we're 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!

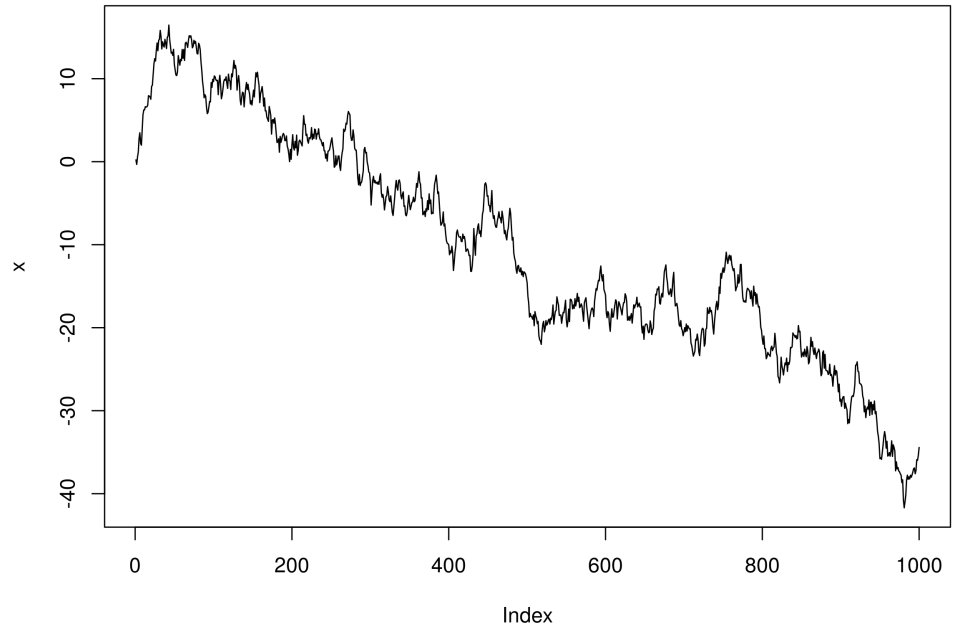

Figure 8.2: Realisation of a Random Walk with 1000 timesteps.

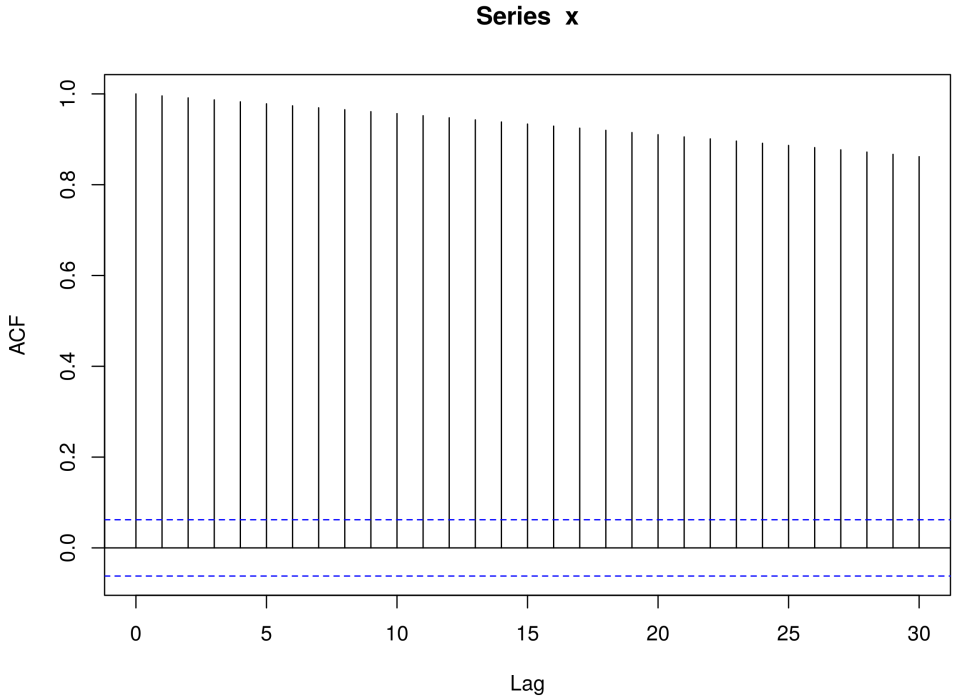

Figure 8.3: Correlogram of a Random Walk.

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 8.4.

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

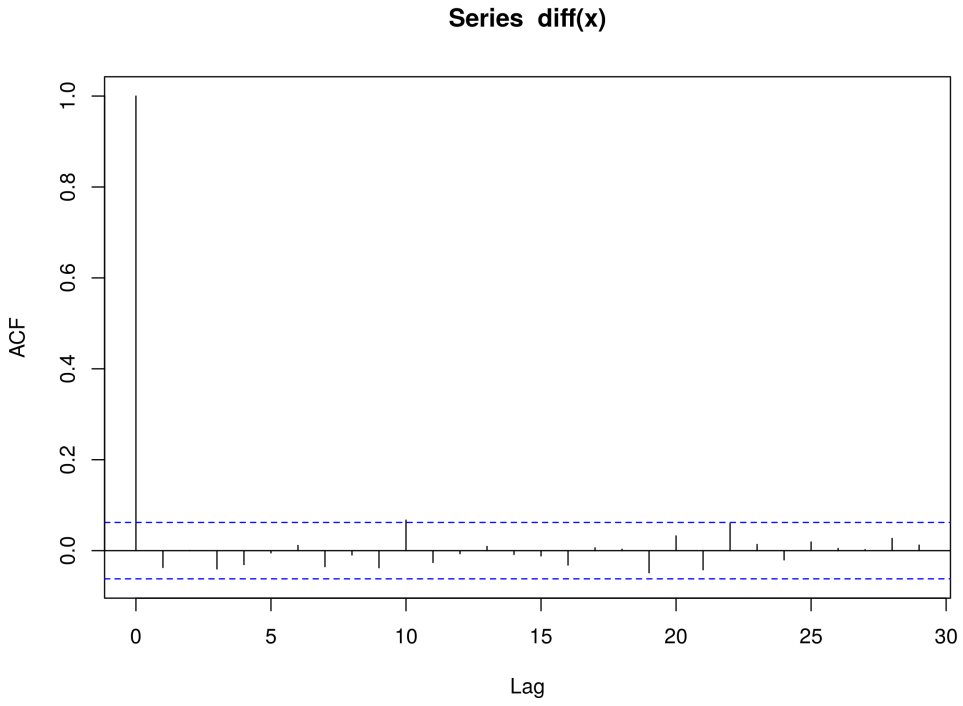

Figure 8.4: Correlogram of the Difference Series from a Simulated Random Walk.

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's 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're able to download any of the data, we must install quantmod as it isn't 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're 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 8.5.

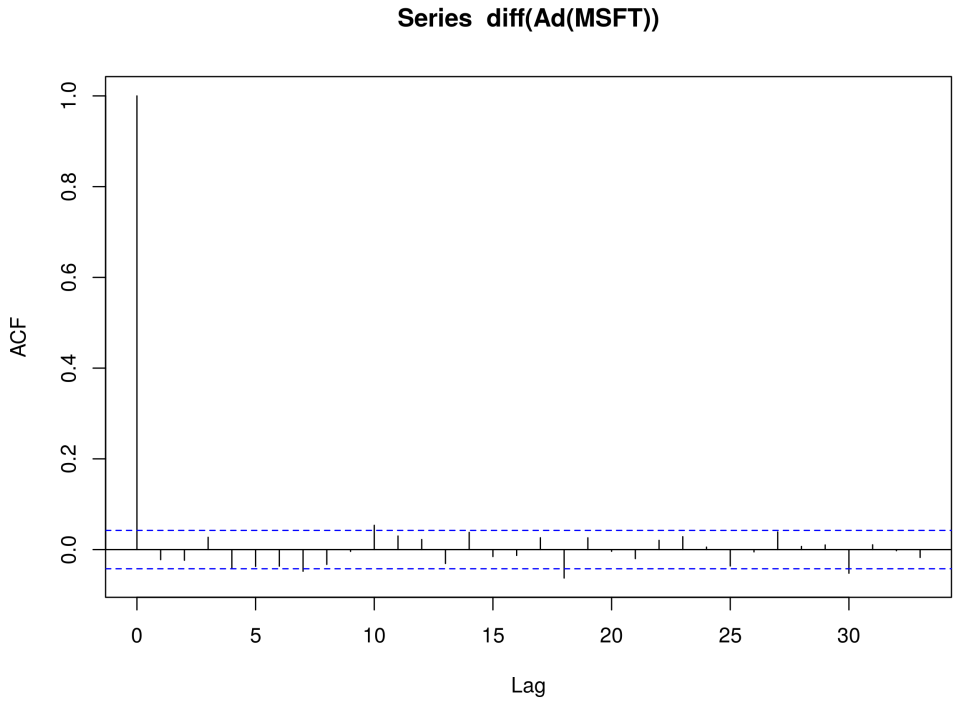

Figure 8.5: Correlogram of the Difference Series from MSFT Adjusted Close.

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 can conclude, with a reasonable degree of certainty, that the adjusted closing prices of MSFT are well approximated by a random walk.

Let's 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)
```

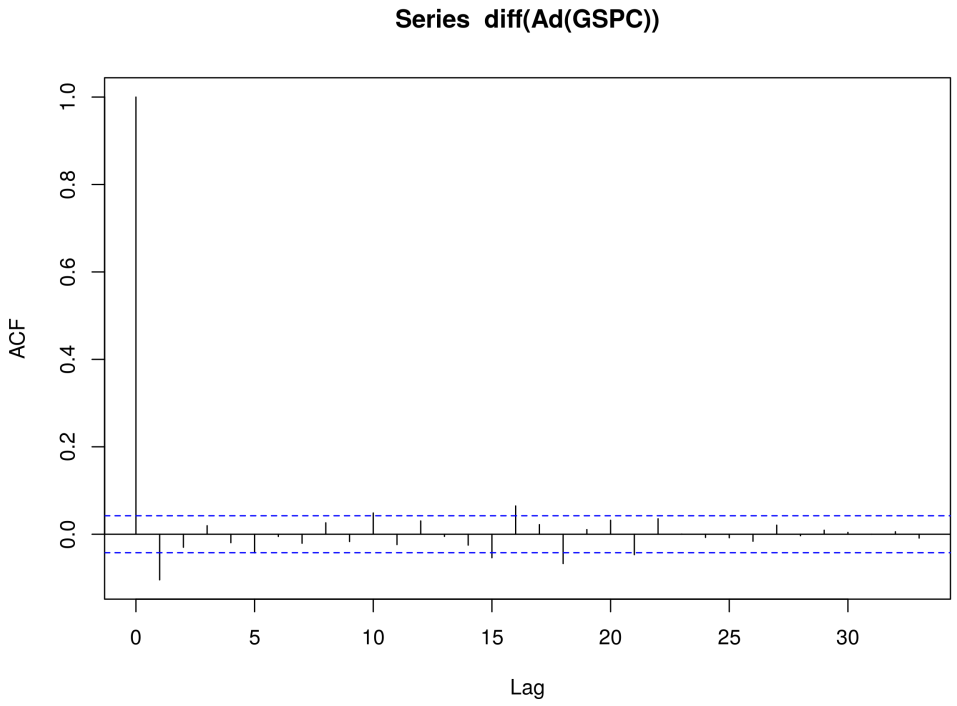

Figure 8.6: Correlogram of the Difference Series from the S&P500 Adjusted Close.

The correlogram is given in Figure 8.6.

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 9

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 and **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've discussed both ARIMA and GARCH separately before we apply them to this strategy.

9.1 How Will We Proceed?

In this chapter we are going to outline some new time series concepts that we'll need 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're 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're going to start by looking at strict stationarity and the AIC.

9.2 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 9.2.1. 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.

9.3 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'll 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 we can't go into too much detail about in 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. Let's provide a definition:

Definition 9.3.1. 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:

$$AIC = -2\log(L) + 2k \tag{9.1}$$

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.

9.4 Autoregressive (AR) Models of order p

The first model we're going to consider, which forms the basis of Part 1, is the Autoregressive model of order p, often shortened to AR(p).

9.4.1 Rationale

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

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

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

$$x_t = \alpha_1 x_{t-1} + \ldots + \alpha_p x_{t-p} + w_t \tag{9.3}$$

$$= \sum_{i=1}^{p} \alpha_i x_{t-i} + w_t \tag{9.4}$$

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*, $\mathbf{B}$, then we can rewrite the above as a function θ of $\mathbf{B}$:

$$\theta_p(\mathbf{B})x_t = (1 - \alpha_1\mathbf{B} - \alpha_2\mathbf{B}^2 - \ldots - \alpha_p\mathbf{B})x_t = w_t \tag{9.5}$$

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, as once we have the α_i coefficients determined, our estimate simply becomes:

$$\hat{x}_t = \alpha_1 x_{t-1} + \ldots + \alpha_p x_{t-p} \tag{9.6}$$

Hence we can make n-step ahead forecasts by producing $\hat{x}_t$, $\hat{x}_{t+1}$, $\hat{x}_{t+2}$, ... 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.

9.4.2 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've 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:

$$\theta_p(\mathbf{B}) = 0 \tag{9.7}$$

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.

Let's consider a few examples to 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.

9.4.3 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:

$$\mu_x = E(x_t) = 0 \tag{9.8}$$

$$\gamma_k = \sum_{i=1}^{p} \alpha_i \gamma_{k-i}, \;\; k > 0 \tag{9.9}$$

$$\rho_k = \sum_{i=1}^{p} \alpha_i \rho_{k-i}, \;\; k > 0 \tag{9.10}$$

Note that it is necessary to know the α_i parameter values prior to calculating the autocorrelations.

Now that we've stated the second order properties we can simulate various orders of AR(p) and plot the corresponding correlograms.

9.4.4 Simulations and Correlograms

AR(1)

Let's 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 9.1.

```
> layout(1:2)
> plot(x, type="l")
> acf(x)
```

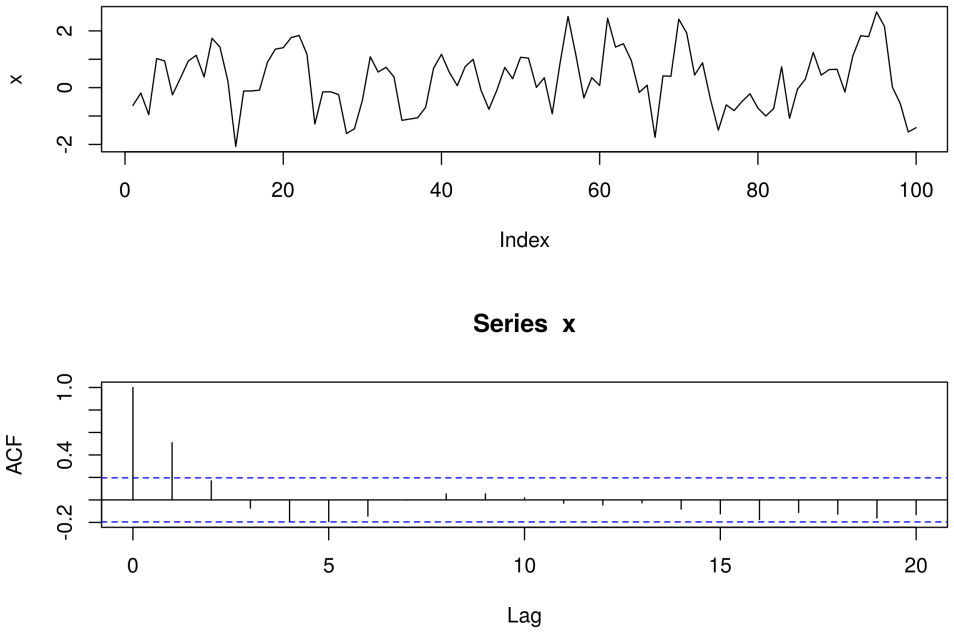

Figure 9.1: Realisation of AR(1) Model, with $\alpha_1 = 0.6$ and Associated Correlogram.

Let's now try fitting an AR(p) process to the simulated data we've 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.

As it turns out 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 let's 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, as we'd expect from the fact we've generated the realisation from the model specifically.

How about if we change the $\alpha_1 = -0.6$? The plot is given in Figure 9.2.

```
> 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
> 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's 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 9.3.

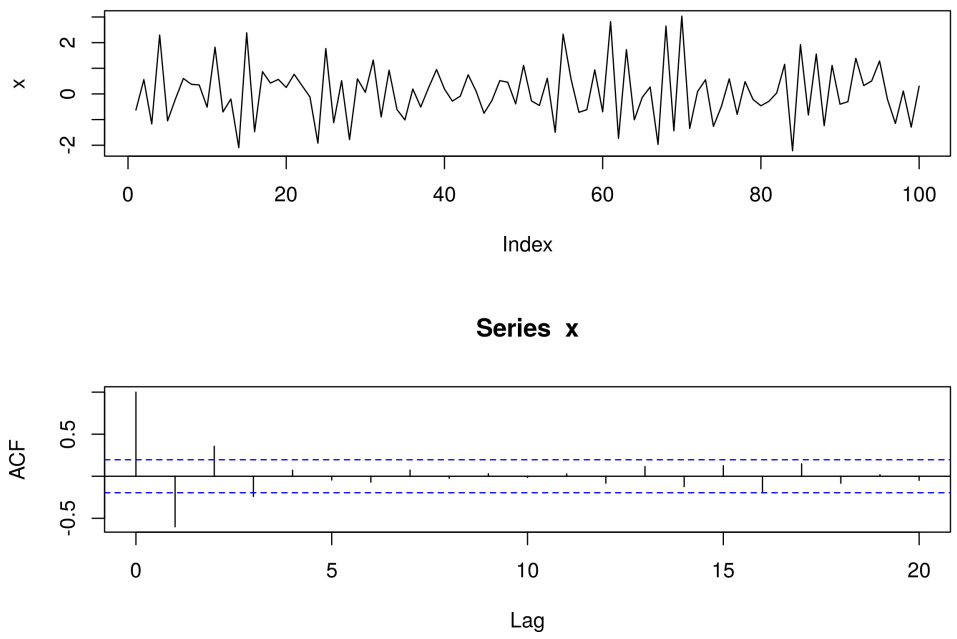

Figure 9.2: Realisation of AR(1) Model, with $\alpha_1 = -0.6$ and Associated Correlogram.

```
> 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'd expect. There are statistically significant peaks at $k = 1$, $k = 3$ and $k = 4$.

Once again, we're 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)
> 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'll 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.

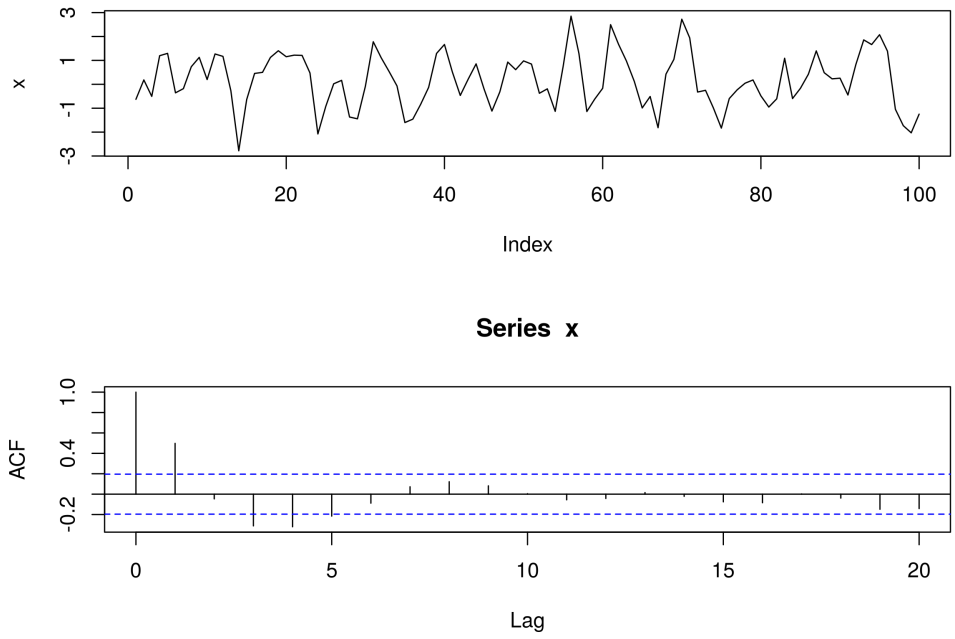

Figure 9.3: Realisation of AR(2) Model, with $\alpha_1 = 0.666$, $\alpha_2 = -0.333$ and Associated Correlogram.

We'll also be using the `arima` command to create confidence intervals around multiple parameters, which is why we've neglected to do it here.

Now that we've created some simulated data it is time to apply the AR(p) models to financial asset time series.

9.4.5 Financial Data

Amazon Inc.

Let's 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'll be using the daily closing prices. The plot is given in Figure 9.4.

```
> 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.

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.

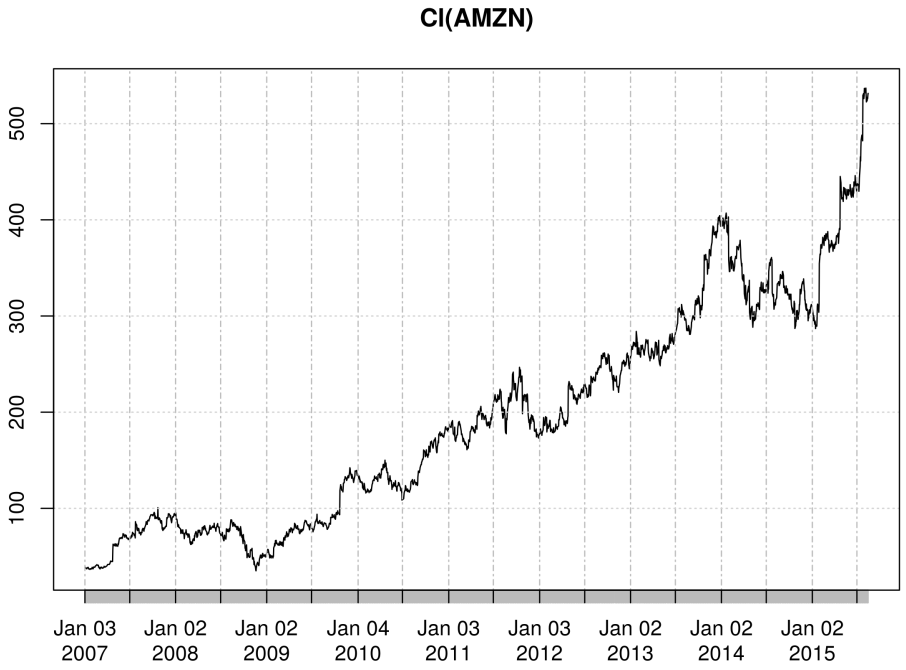

Figure 9.4: Daily Closing Price of AMZN.

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's 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 9.5.

```
> plot(amznrt)
```

At this stage we want to plot the correlogram. We're 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 9.6.

```
> 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
             [,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've also output the aysmptotic variance

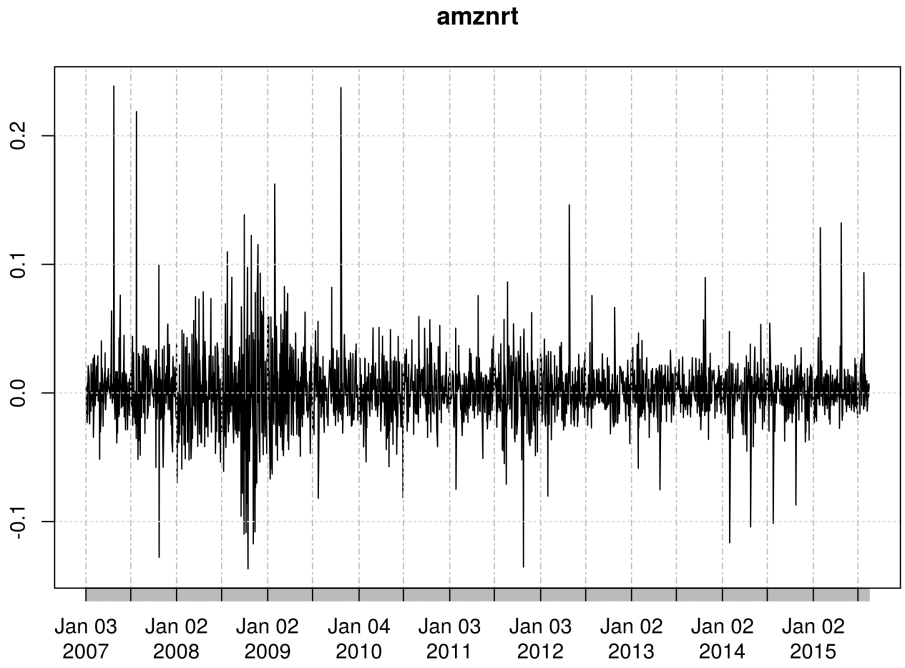

Figure 9.5: First Order Differenced Daily Logarithmic Returns of AMZN Closing Prices.

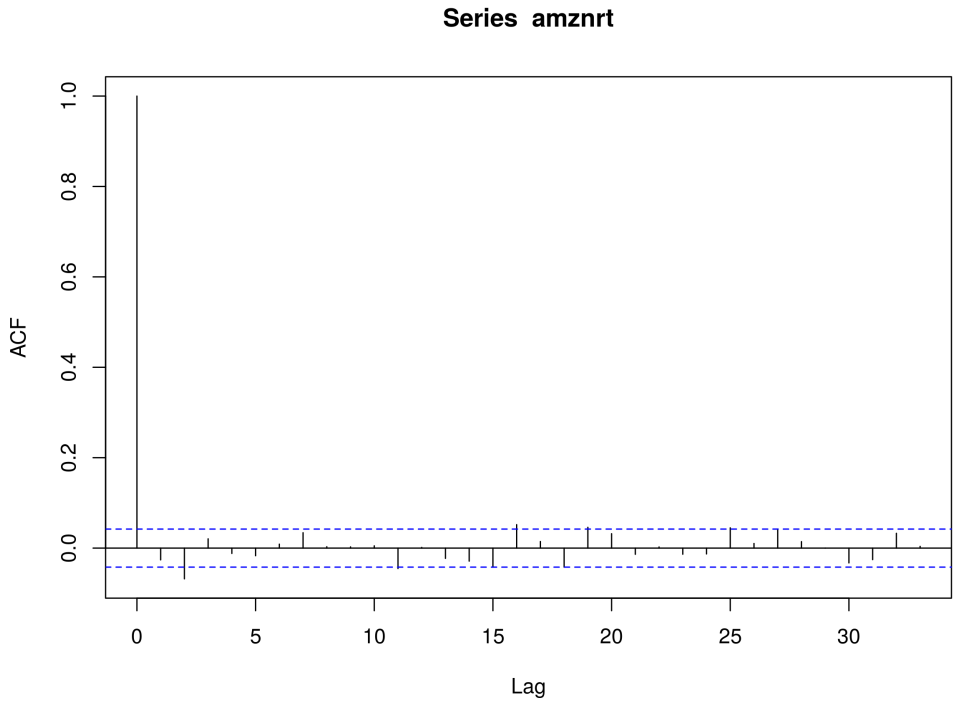

Figure 9.6: Correlogram of First Order Differenced Daily Logarithmic Returns of AMZN Closing Prices.

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'll 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's 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 9.7.

```
> plot(Cl(GSPC))
```

As before, we'll create the first order difference of the log closing prices:

```
> gspcrt = diff(log(Cl(GSPC)))
```

Once again, we can plot the series, as given in Figure 9.8.

```
> plot(gspcrt)
```

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 9.9. 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's see what we get if we do so:

```
> gspcrt.ar <- ar(gspcrt, na.action=na.omit)
> gspcrt.ar$order
[1] 22
> gspcrt.ar$ar
```

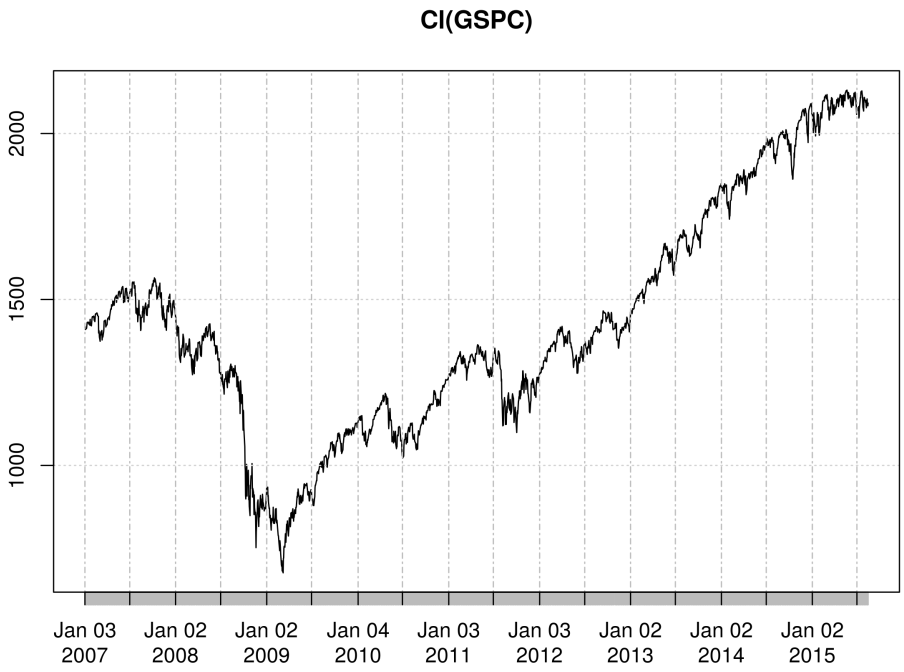

Figure 9.7: Daily Closing Price of S&500.

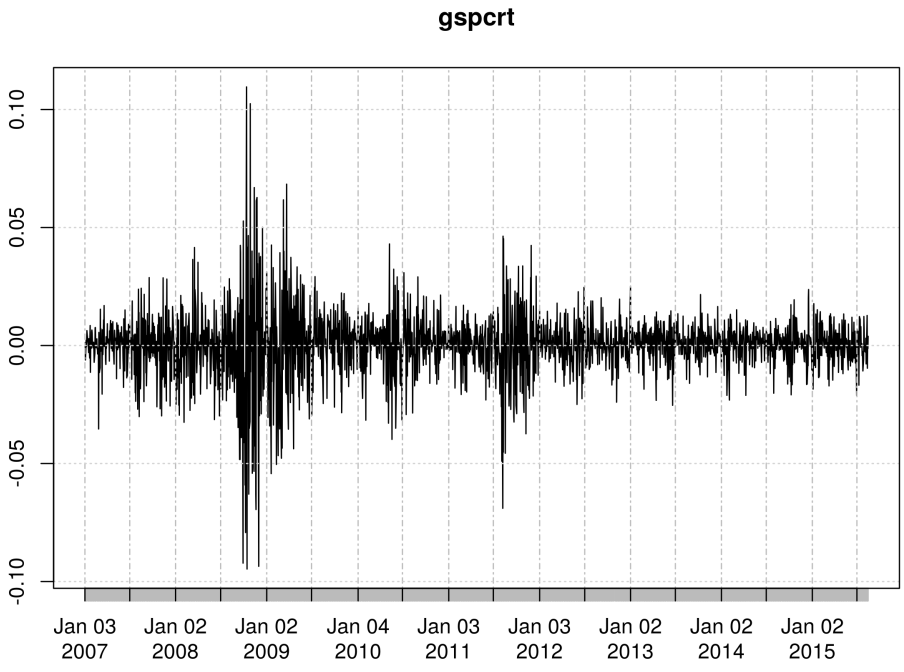

Figure 9.8: First Order Differenced Daily Logarithmic Returns of S&500 Closing Prices.

```
[1] -0.111821507 -0.060150504  0.018791594 -0.025619932 -0.046391435
[6]  0.002266741 -0.030089046  0.030430265 -0.007623949  0.044260402
```

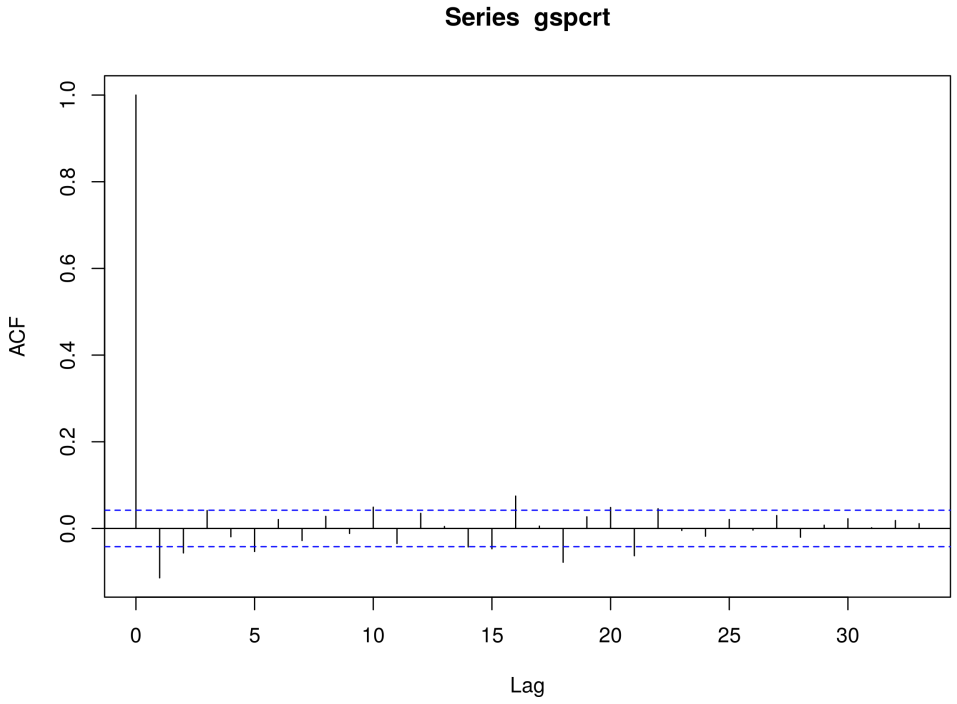

Figure 9.9: Correlogram of First Order Differenced Daily Logarithmic Returns of S&500 Closing Prices.

```
[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 indicative that there is likely a lot more complexity in the serial correlation than a simple linear model of past prices can really account for.

However, we already knew this because we can see that there is significant serial correlation in the volatility. For instance, consider 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'll learn about both of these in the next couple of sections of this chapter. As we repeatedly mention, these 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.

9.5 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.

9.5.1 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.

9.5.2 Definition

Mathematically, the MA(q) is a linear regression model and is similarly structured to AR(p):

Definition 9.5.1. Moving Average Model of order q. A time series model, $\{x_t\}$, is a *moving average model of order q*, MA(q), if:

$$x_t = w_t + \beta_1 w_{t-1} + \ldots + \beta_q w_{t-q} \tag{9.11}$$

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 $\mathbf{B}$:

$$x_t = (1 + \beta_1 \mathbf{B} + \beta_2 \mathbf{B}^2 + \ldots + \beta_q \mathbf{B}^q) w_t = \phi_q(\mathbf{B}) w_t \tag{9.12}$$

We will make use of the ϕ function in subsequent chapters.

9.5.3 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.

$$\text{Mean: } \mu_x = E(x_t) = \sum_{i=0}^{q} E(w_i) = 0 \tag{9.13}$$

$$\text{Var: } \sigma_w^2 (1 + \beta_1^2 + \ldots + \beta_q^2) \tag{9.14}$$

$$\text{ACF: } \rho_k = \begin{cases} 1 & \text{if } k = 0 \\ \sum_{i=0}^{q-k} \beta_i \beta_{i+k} / \sum_{i=0}^{q} \beta_i^2 & \text{if } k = 1, \ldots, q \\ 0 & \text{if } k > q \end{cases}$$

Where $\beta_0 = 1$.

We're 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.

9.5.4 Simulations and Correlograms

MA(1)

Let's start with a MA(1) process. If we set $\beta_1 = 0.6$ we obtain the following model:

$$x_t = w_t + 0.6w_{t-1} \tag{9.15}$$

As with the AR(p) model we can use R to simulate such a series and then plot the correlogram. Since we've 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 9.10.

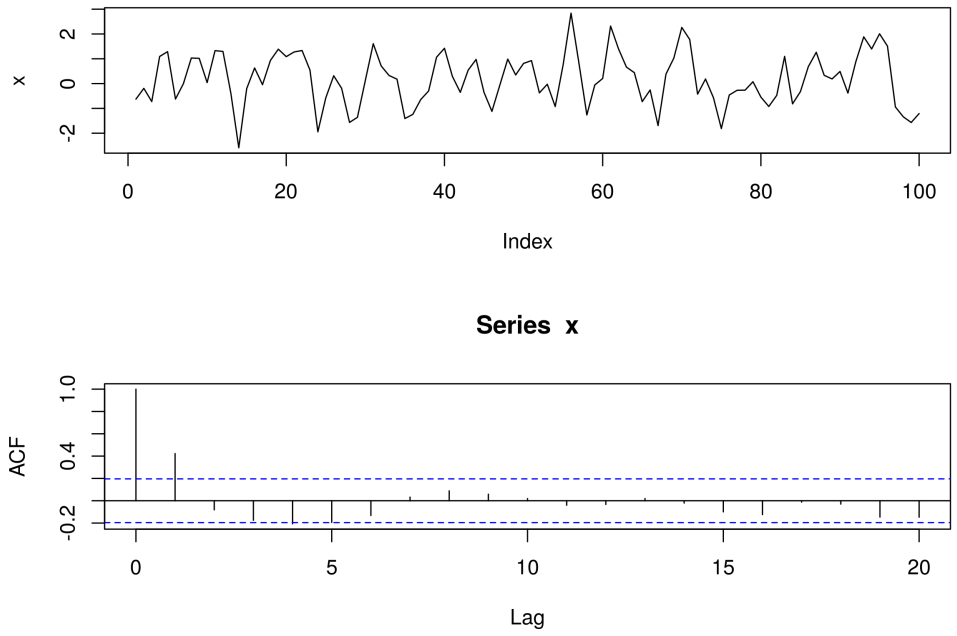

Figure 9.10: Realisation of MA(1) Model, with $\beta_1 = 0.6$ and Associated Correlogram

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.

In fact, this is a useful way of seeing 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're now going to try and fit a MA(1) model to our simulated data. Unfortunately, there isn't an 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:

```
> 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'll return to this point later.

As a quick check we're 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? Let's 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 9.11.

We can see that at $k = 1$ we have a significant peak in the correlogram, except that it shows negative correlation, as we'd expect from a MA(1) model with negative first coefficient. Once again all peaks beyond $k = 1$ are insignificant. Let's 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
```

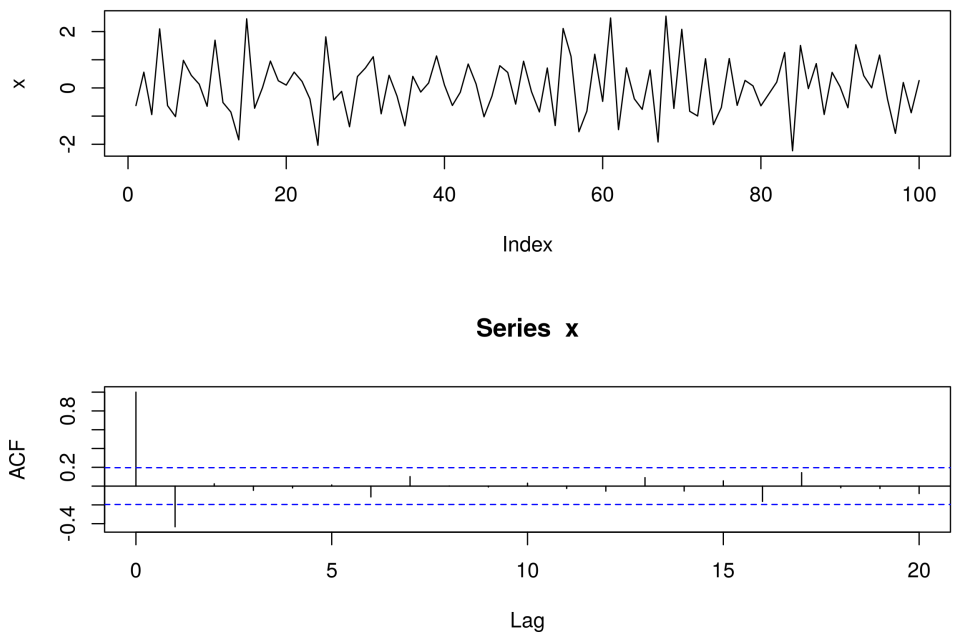

Figure 9.11: Realisation of MA(1) Model, with $\beta_1 = -0.6$ and Associated Correlogram

```
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's 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)

Let's 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's simulate a MA(3) process from this model. I've 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 9.12.

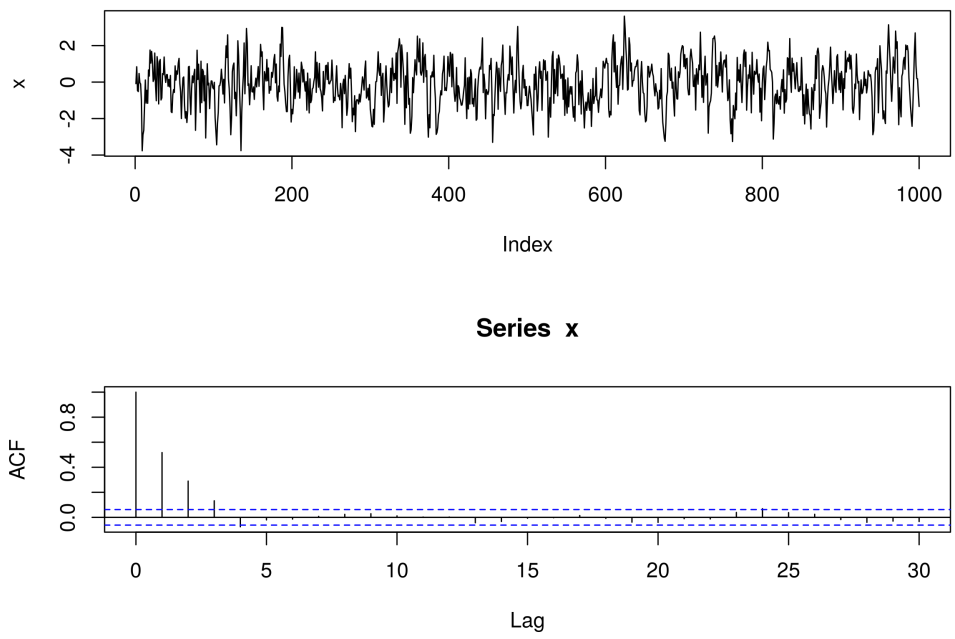

Figure 9.12: Realisation of MA(3) Model and Associated Correlogram

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 $k = q$.

Let's 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
```

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.

9.5.5 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's 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'll 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 9.13.

```
> acf(amznrt.ma$res[-1])
```

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's 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's plot the residuals once again, given in Figure 9.14.

```
> acf(amznrt.ma$res[-1])
```

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

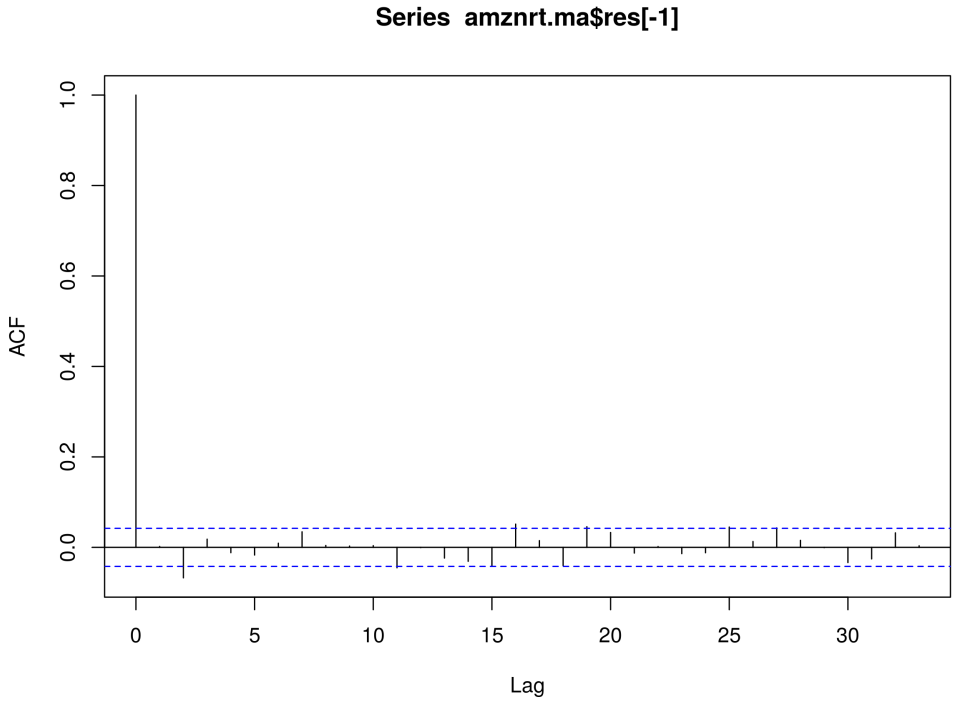

Figure 9.13: Residuals of MA(1) Model Fitted to AMZN Daily Log Prices

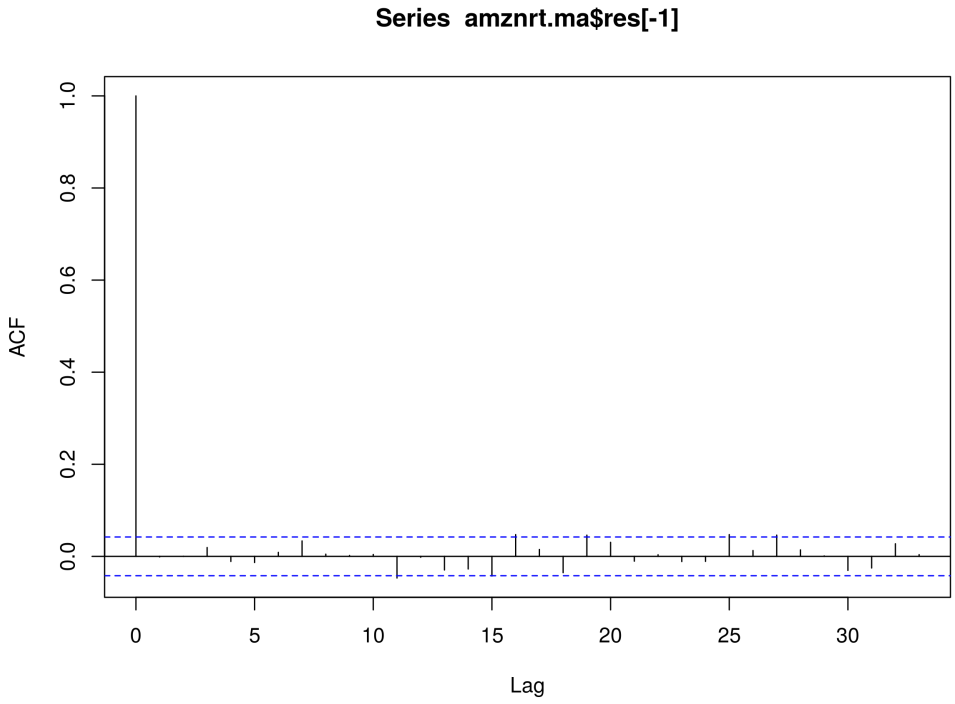

Figure 9.14: Residuals of MA(2) Model Fitted to AMZN Daily Log Prices

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 9.15.

```
> acf(amznrt.ma$res[-1])
```

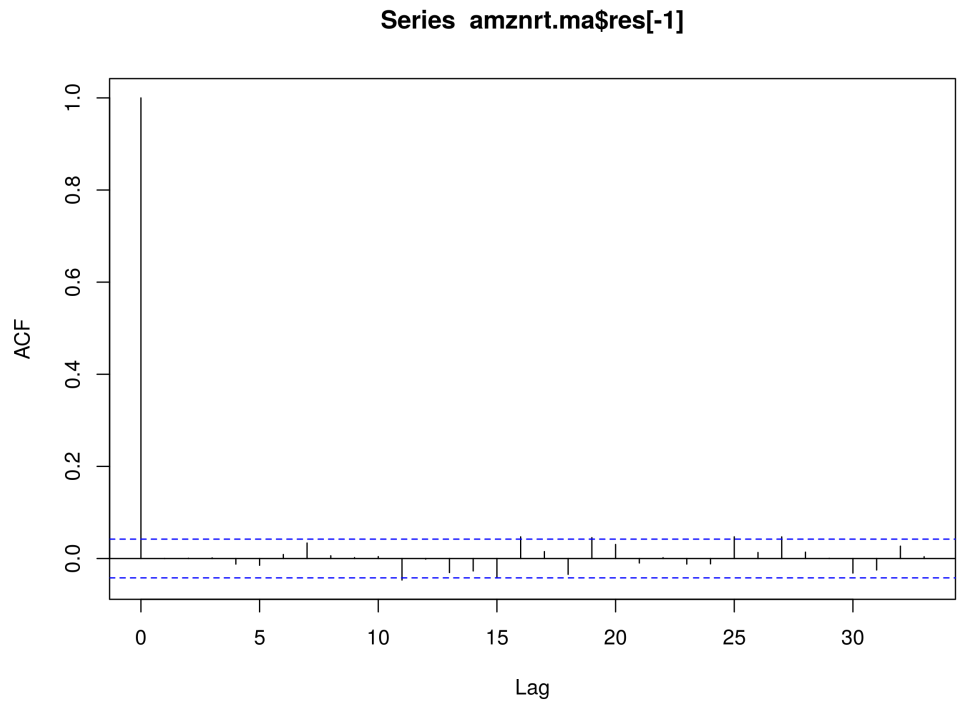

Figure 9.15: Residuals of MA(3) Model Fitted to AMZN Daily Log Prices

The MA(3) residuals plot looks almost identical to that of the MA(2) model. This is not surprising, as we're 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've 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 isn't a good fit, because this is the ultimate process we will be following when we come across much more sophisticated models, that are potentially harder to interpret.

Let's 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's 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's make a plot of the residuals of this fitted model, as given in Figure 9.16.

```
> 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
      -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, let's make a plot of the residuals of this fitted MA(2) model, as given in Figure 9.17.

```
> acf(gspcrt.ma$res[-1])
```

While the peak at $k = 2$ has disappeared (as we'd 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), but once again we should also expect no reduction in further lags.

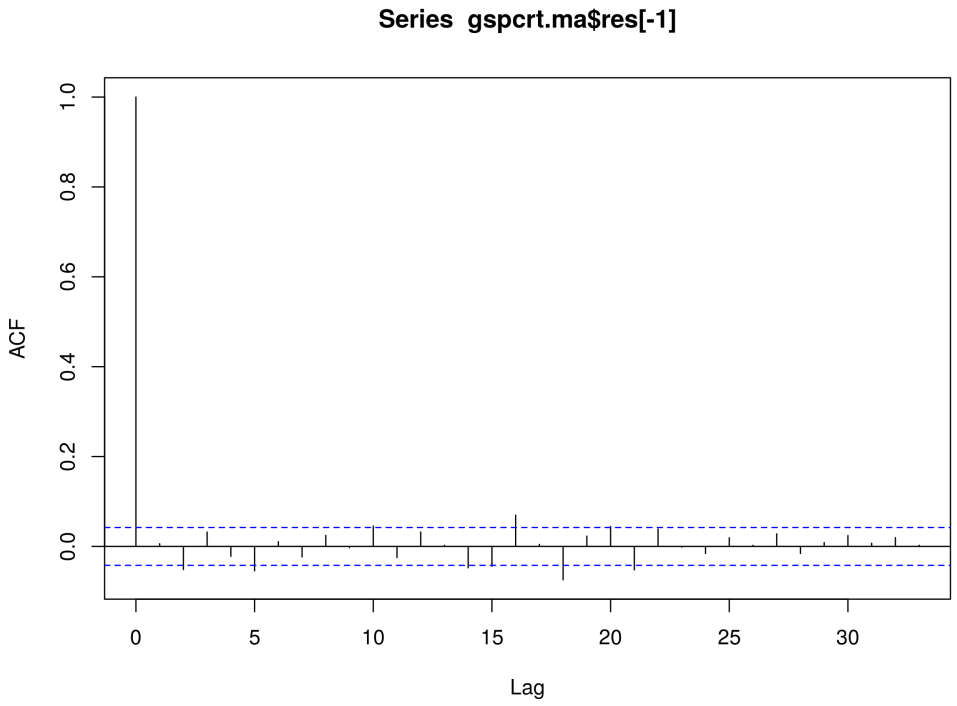

Figure 9.16: Residuals of MA(1) Model Fitted to S&P500 Daily Log Prices

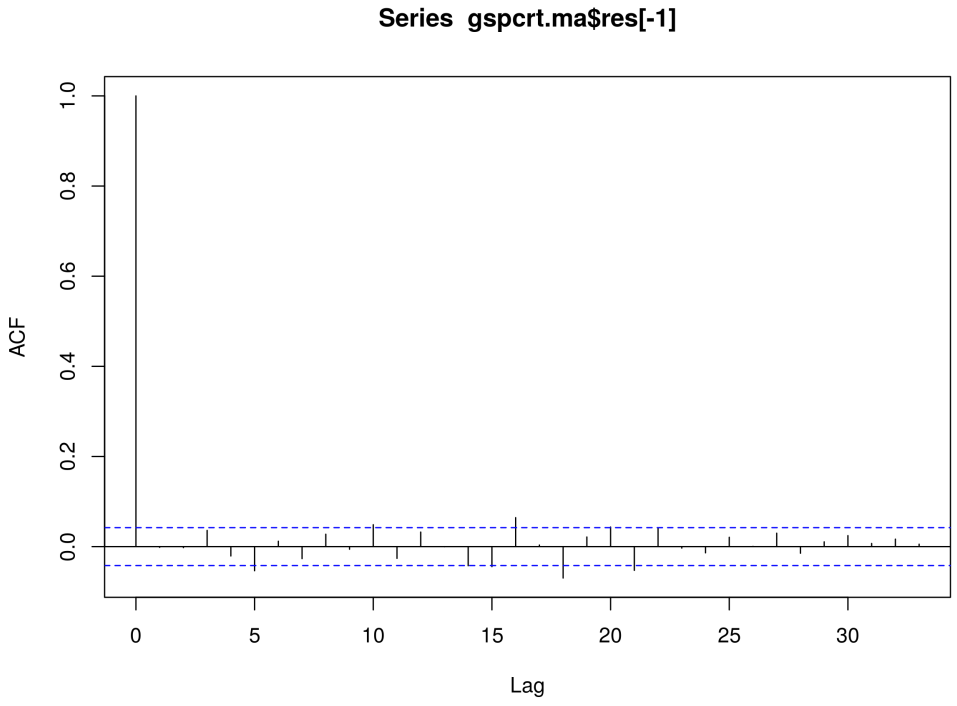

Figure 9.17: Residuals of MA(2) Model Fitted to S&P500 Daily Log Prices

```
> 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

sigma^2 estimated as 0.0001838:  log likelihood = 6253.9,  aic = -12497.81
```

Finally, let's make a plot of the residuals of this fitted MA(3) model, as given in Figure 9.18.

```
> acf(gspcrt.ma$res[-1])
```

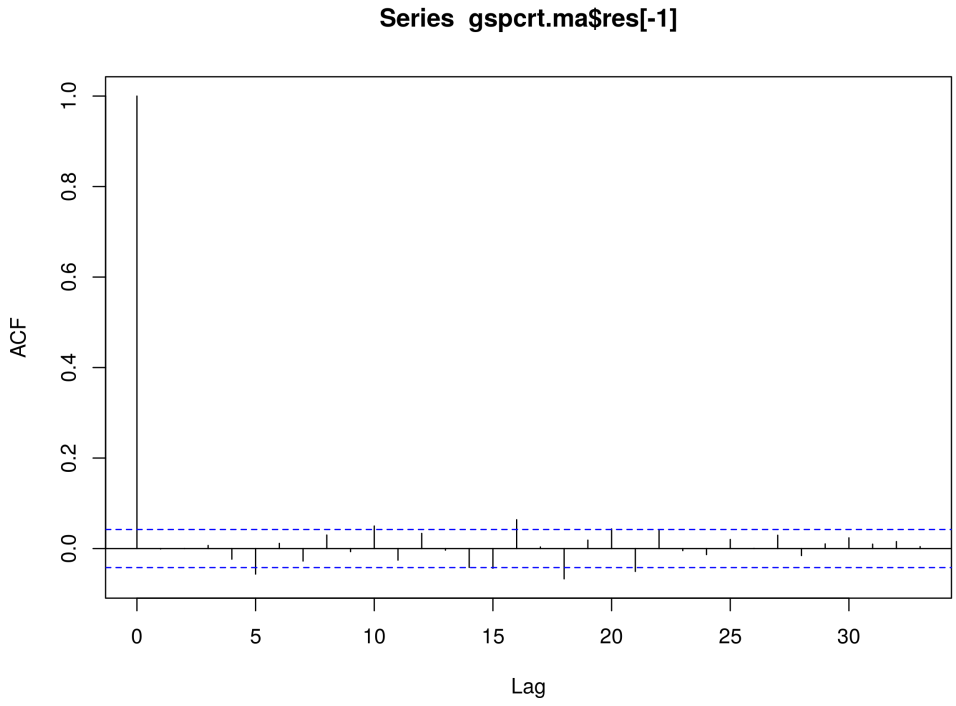

Figure 9.18: Residuals of MA(3) Model Fitted to S&P500 Daily Log Prices

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.

9.5.6 Next Steps

We've 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've seen that they're 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.

9.6 Autogressive Moving Average (ARMA) Models of order p, q

We've 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've 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.

9.6.1 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 9.6.1. 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:

$$BIC = -2\log(L) + k\log(n) \tag{9.16}$$

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.

9.6.2 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 9.6.2. 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:

$$Q = n(n+2)\sum_{k=1}^{h}\frac{\hat{\rho}_k^2}{n-k} \tag{9.17}$$

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 in fact use R to calculate the test for us, simplifying the procedure somewhat.

Now that we've discussed the BIC and the Ljung-Box test, we're ready to discuss our first mixed model, namely the Autoregressive Moving Average of order p, q, or ARMA(p,q).

9.6.3 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 and as such 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 announcement or other unexpected event. A good example of "shock" news would be the BP Deepwater Horizon oil spill.

Hence, 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, 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.

9.6.4 Definition

The ARMA(p,q) model is a linear combination of two linear models and thus is itself still linear:

Definition 9.6.3. 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:

$$x_t = \alpha_1 x_{t-1} + \alpha_2 x_{t-2} + \ldots + w_t + \beta_1 w_{t-1} + \beta_2 w_{t-2} \ldots + \beta_q w_{t-q} \tag{9.18}$$

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 $\mathbf{B}$:

$$\theta_p(\mathbf{B})x_t = \phi_q(\mathbf{B})w_t \tag{9.19}$$

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, thus leading to a simpler model.

9.6.5 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.

Let's start with the simplest possible non-trivial ARMA model, namely the ARMA(1,1) model. That 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. Such a model is given by:

$$x_t = \alpha x_{t-1} + w_t + \beta w_{t-1} \tag{9.20}$$

We need to specify the coefficients prior to simulation. Let's 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 9.19.

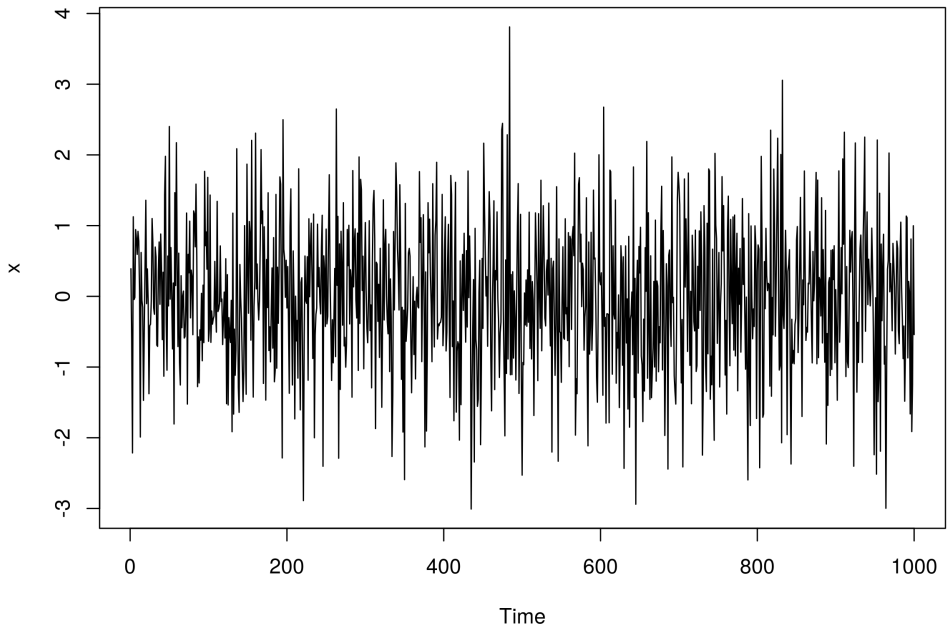

Figure 9.19: Realisation of an ARMA(1,1) Model, with $\alpha = 0.5$ and $\beta = -0.5$

Let's also plot the correlogram, as given in Figure 9.20.

```
> acf(x)
```

We can see that there is no significant autocorrelation, which is to be expected from an ARMA(1,1) model.

Finally, let's try and determine the coefficients and their standard errors using the `arima` function:

```
> arima(x, order=c(1, 0, 1))

Call:
```

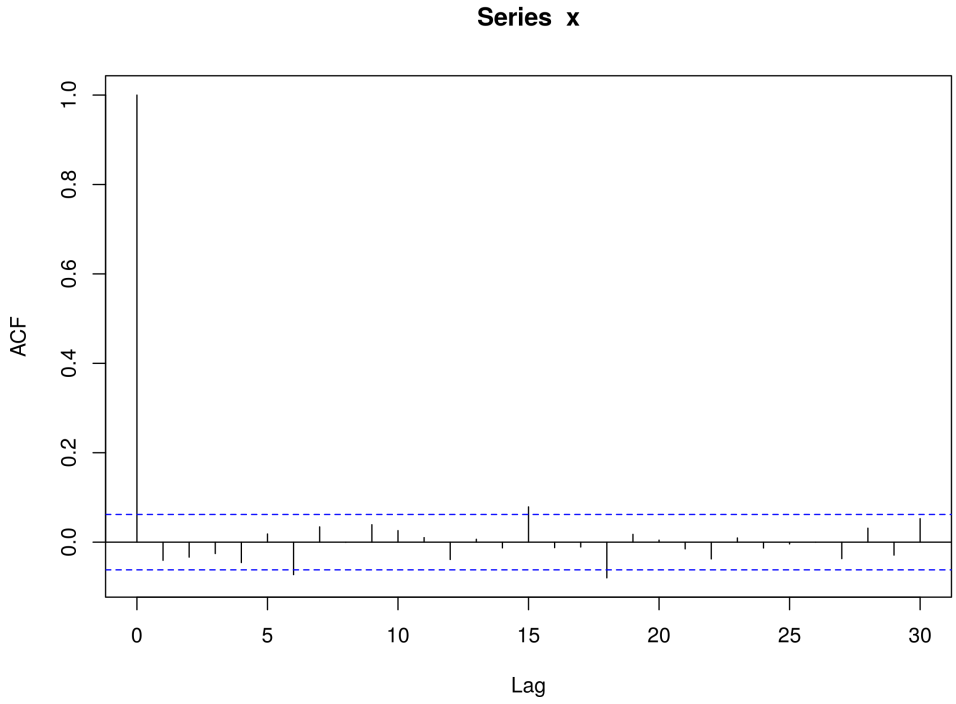

Figure 9.20: Correlogram of an ARMA(1,1) Model, with $\alpha = 0.5$ and $\beta = -0.5$

```
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, however we should note that the 95% confidence intervals are very wide (a consequence of the reasonably large standard errors).

Let's 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's 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 9.21.

And the corresponding autocorelation, as given in Figure 9.22.

```
> acf(x)
```

We can now try fitting an ARMA(2,2) model to the data:

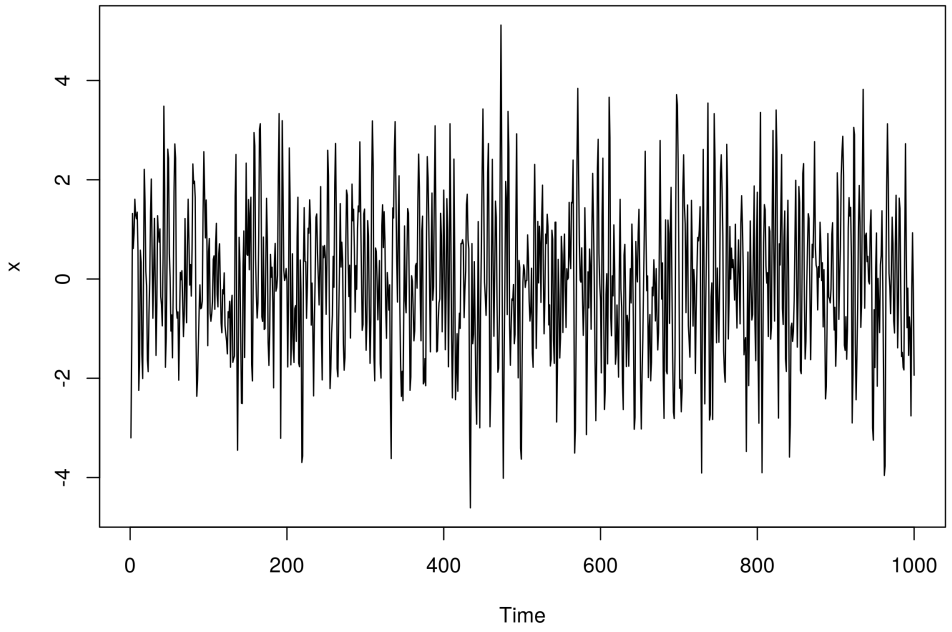

Figure 9.21: Realisation of an ARMA(2,2) Model, with $\alpha_1 = 0.5$, $\alpha_2 = -0.25$, $\beta_1 = 0.5$ and $\beta_2 = -0.3$

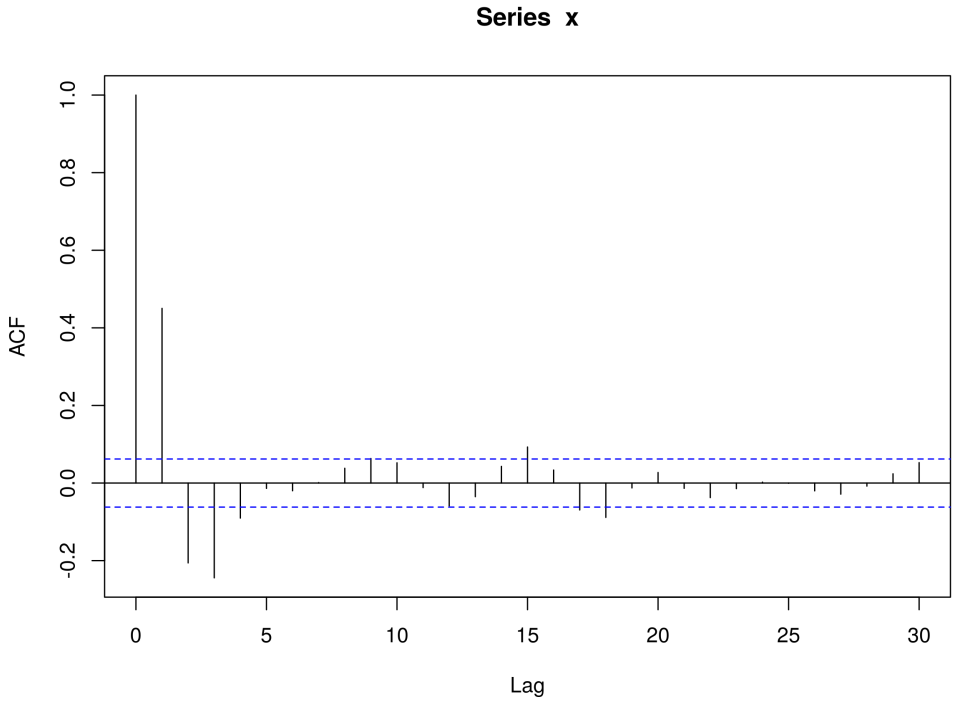

Figure 9.22: Correlogram of an ARMA(2,2) Model, with $\alpha_1 = 0.5$, $\alpha_2 = -0.25$, $\beta_1 = 0.5$ and $\beta_2 = -0.3$

```
> 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
[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 just need to have a predictive power that exceeds chance and produces enough profit above transaction costs, in order to be profitable in the long run.

Now that we've 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.

9.6.6 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's 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's 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 9.23.

```
> acf(resid(final.arma))
```

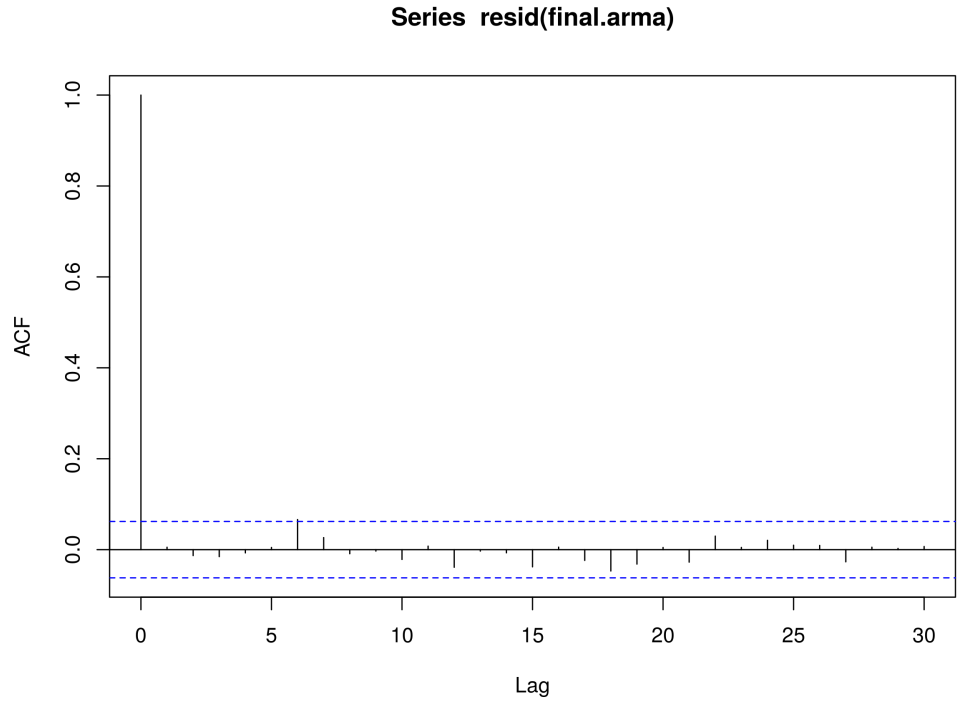

Figure 9.23: Correlogram of the residuals of the best fitting ARMA(p,q) Model, $p = 3$ and $q = 2$

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've 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.

9.6.7 Financial Data

Now that we've 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's download the daily closing prices using **quantmod** and then create the log returns stream:

```
> require(quantmod)
> getSymbols("^GSPC")
> sp = diff(log(Cl(GSPC)))
```

Let's 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's plot the residuals of the fitted model to the S&P500 log daily returns stream, as given in Figure 9.24:

```
> acf(resid(spfinal.arma), na.action=na.omit)
```

Notice that there are some significant peaks, especially at higher lags. This is indicative of a poor fit. Let's 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
```

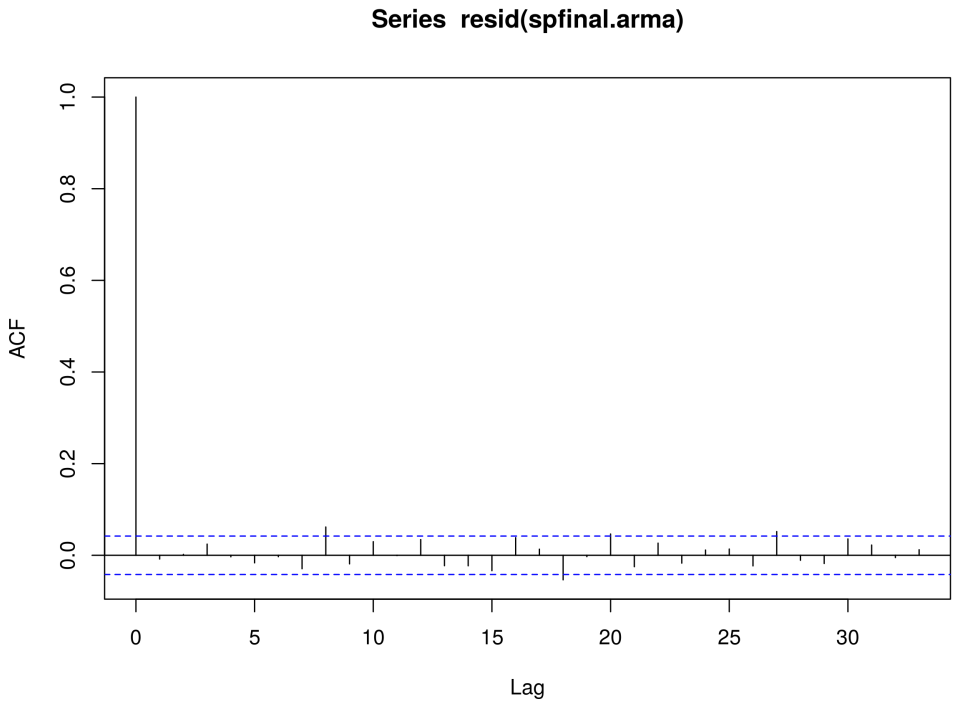

Figure 9.24: Correlogram of the residuals of the best fitting ARMA(p,q) Model, $p = 3$ and $q = 3$, to the S&P500 daily log returns stream

As we suspected, the p-value is *less* that 0.05 and as such 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.

9.7 Next Steps

As we've 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 10

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*.

10.1 Quick Recap

We have steadily built up our understanding of time series with concepts such as serial correlation, stationarity, linearity, residuals, correlograms, simulating, 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 (and even those currently in the research literature), we will have a significant knowledge base on which to draw, in order to effectively evaluate these models, instead of 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'd 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're very nearly ready to consider our first trading model, which will be a mixture of ARIMA and GARCH, so 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 long-memory processes, state-space models (i.e. the Kalman

Filter) and Vector Autoregressive (VAR) models, which will lead us to other, more sophisticated, trading strategies.

10.2 Autoregressive Integrated Moving Average (ARIMA) Models of order p, d, q

10.2.1 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):

$$\nabla x_t = x_t - x_{t-1} = w_t \tag{10.1}$$

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), however we won't be discussing SARIMA much in this book. Conditional heteroskedastic effects, such as volatility clustering in equities indexes, can be tackled with ARCH and GARCH, which we discuss later in this chapter.

In this chapter we will first be considering non-stationary series with stochastic trends and fit ARIMA models to these series. We will also finally produce forecasts for our financial series.

10.2.2 Definitions

Prior to defining ARIMA processes we need to discuss the concept of an **integrated** series:

Definition 10.2.1. Integrated Series of order d. A time series $\{x_t\}$ is *integrated of order* d, $I(d)$, if:

$$\nabla^d x_t = w_t \tag{10.2}$$

That is, if we difference the series d times we receive a discrete white noise series.

Alternatively, using the Backward Shift Operator $\mathbf{B}$ an equivalent condition is:

$$(1 - \mathbf{B}^d)x_t = w_t \tag{10.3}$$

Now that we have defined an integrated series we can define the ARIMA process itself:

Definition 10.2.2. 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, $\mathbf{B}$:

$$\theta_p(\mathbf{B})(1 - \mathbf{B})^d x_t = \phi_q(\mathbf{B})w_t \tag{10.4}$$

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.

10.2.3 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 10.1.

```
> set.seed(2)
> x <- arima.sim(list(order = c(1,1,1), ar = 0.6, ma=-0.5), n = 1000)
> plot(x)
```

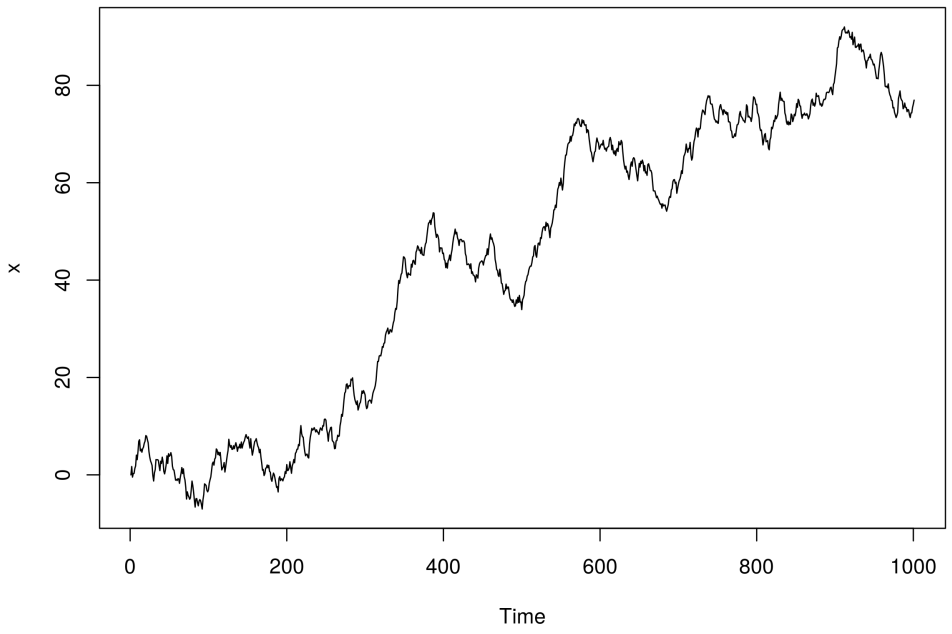

Figure 10.1: Plot of simulated ARIMA(1,1,1) model with $\alpha = 0.6$ and $\beta = -0.5$

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))

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 shouldn't be surprised to see the residuals looking like a realisation of discrete white noise, see Figure 10.2.

```
> acf(resid(x.arima))
```

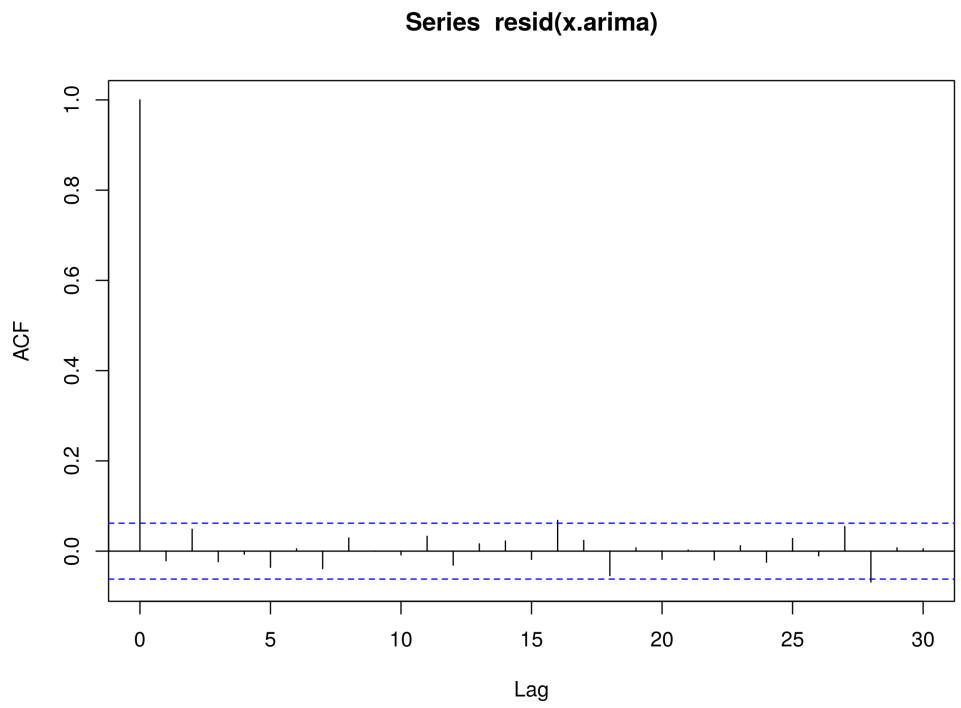

Figure 10.2: Correlogram of the residuals of the fitted ARIMA(1,1,1) model

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")

  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. Hence, the ARIMA(1,1,1) model is a good fit, as expected.

10.2.4 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[30].

Let's 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 10.3.

```
> 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's 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, see Figure 10.4.

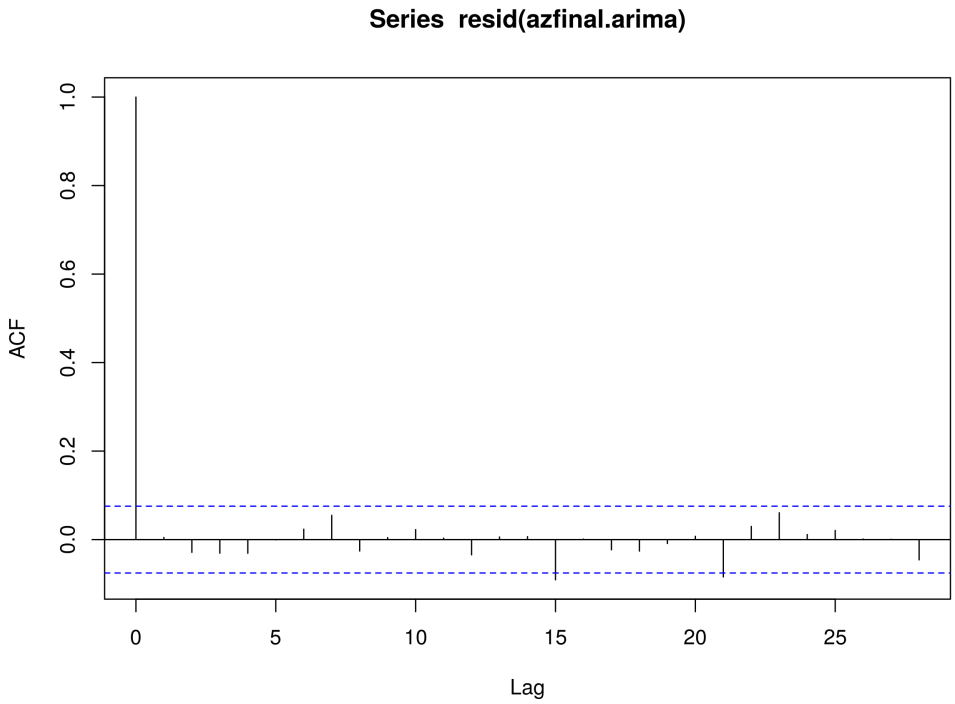

Figure 10.3: Correlogram of residuals of ARIMA(4,0,4) model fitted to AMZN daily log returns

```
> plot(forecast(azfinal.arima, h=25))
```

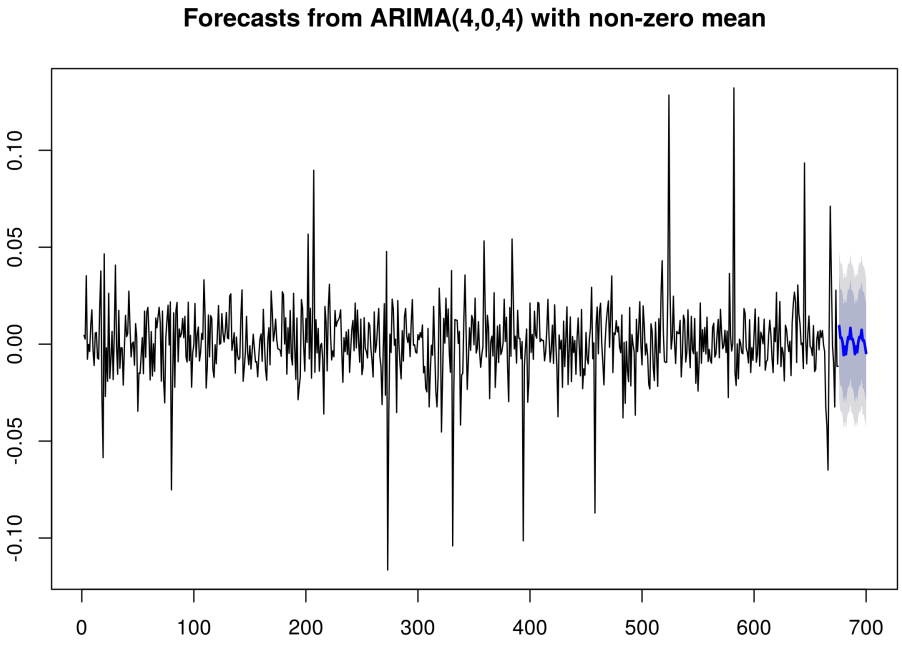

Figure 10.4: 25-day forecast of AMZN daily log returns

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's 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:

```
> 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 10.5.

```
> acf(resid(spfinal.arima), na.action=na.omit)
```

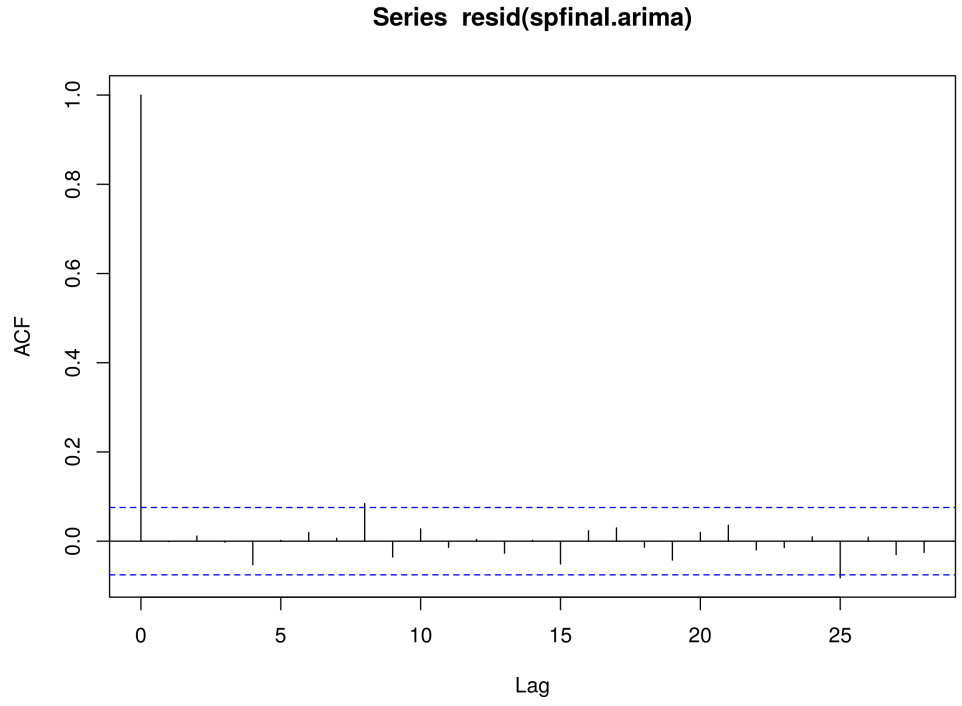

Figure 10.5: Correlogram of residuals of ARIMA(2,0,1) model fitted to S&P500 daily log returns

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's now plot a forecast for the next 25 days of the S&P500 daily log returns, see Figure 10.6.

```
> plot(forecast(spfinal.arima, h=25))
```

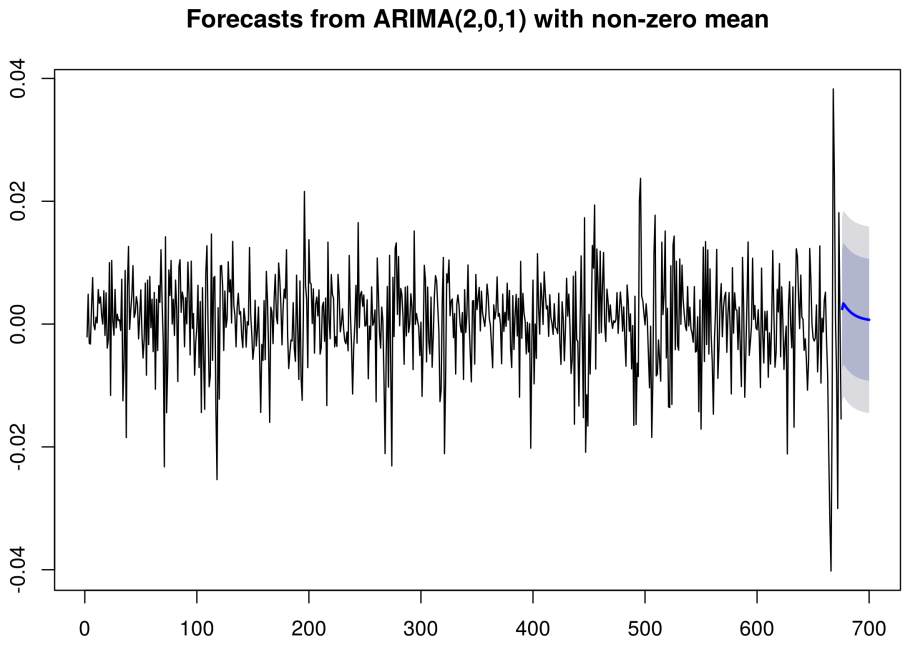

Figure 10.6: 25-day forecast of S&P500 daily log returns

Now that we have the ability to fit and forecast models such as ARIMA, we're very close to being able to create strategy indicators for trading.

10.2.5 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.

10.3 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

Hence, 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 come up with new strategies that trade volatility directly.

We're now going to turn our attention to conditional heteroskedasticity and discuss what it means.

10.4 Conditional Heteroskedasticity

Let's 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.

10.5 Autoregressive Conditional Heteroskedastic Models

We've 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'll begin with the simplest possible case, namely an ARCH model that depends solely on the previous variance value in the series.

10.5.1 ARCH Definition

Definition 10.5.1. Autoregressive Conditional Heteroskedastic Model of Order Unity.

A time series $\{\epsilon_t\}$ is given at each instance by:

$$\epsilon_t = \sigma_t w_t \tag{10.5}$$

Where $\{w_t\}$ is discrete white noise, with zero mean and unit variance, and σ_t^2 is given by:

$$\sigma_t^2 = \alpha_0 + \alpha_1 \epsilon_{t-1}^2 \tag{10.6}$$

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:

$$\epsilon_t = w_t \sqrt{\alpha_0 + \alpha_1 \epsilon_{t-1}^2} \tag{10.7}$$

10.5.2 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:

$$\begin{aligned}
\operatorname{Var}(\epsilon_t) &= \mathrm{E}[\epsilon_t^2] - (\mathrm{E}[\epsilon_t])^2 && (10.8)\\
&= \mathrm{E}[\epsilon_t^2] && (10.9)\\
&= \mathrm{E}[w_t^2]\,\mathrm{E}[\alpha_0 + \alpha_1 \epsilon_{t-1}^2] && (10.10)\\
&= \mathrm{E}[\alpha_0 + \alpha_1 \epsilon_{t-1}^2] && (10.11)\\
&= \alpha_0 + \alpha_1 \operatorname{Var}(\epsilon_{t-1}) && (10.12)
\end{aligned}$$

Where I have used the definitions of the variance $\operatorname{Var}(x) = \mathrm{E}[x^2] - (\mathrm{E}[x])^2$ 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:

$$x_t = \alpha_0 + \alpha_1 x_{t-1} + w_t \tag{10.13}$$

You can see that the models are similar in form (with the exception of the white noise term).

10.5.3 When Is It Appropriate To Apply ARCH(1)?

So what approach can we take 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*, i.e. that has 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.

10.5.4 ARCH(p) Models

It is straightforward to extend ARCH to higher order lags. An ARCH(p) process is given by:

$$\epsilon_t = w_t \sqrt{\alpha_0 + \sum_{i=1}^{p} \alpha_p \epsilon_{t-i}^2} \tag{10.14}$$

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.

10.6 Generalised Autoregressive Conditional Heteroskedastic Models

10.6.1 GARCH Definition

Definition 10.6.1. Generalised Autoregressive Conditional Heteroskedastic Model of Order p, q.

A time series $\{\epsilon_t\}$ is given at each instance by:

$$\epsilon_t = \sigma_t w_t \tag{10.15}$$

Where $\{w_t\}$ is discrete white noise, with zero mean and unit variance, and σ_t^2 is given by:

$$\sigma_t^2 = \alpha_0 + \sum_{i=1}^{q} \alpha_i \epsilon_{t-i}^2 + \sum_{j=1}^{p} \beta_j \sigma_{t-j}^2 \tag{10.16}$$

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.

10.6.2 Simulations, Correlograms and Model Fittings

As always we're going to begin with the simplest possible case of the model, namely 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:

$$\epsilon_t = \sigma_t w_t \tag{10.17}$$
$$\sigma_t^2 = \alpha_0 + \alpha_1 \epsilon_{t-1}^2 + \beta_1 \sigma_{t-1}^2 \tag{10.18}$$

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's 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 our original random walk simulations. That is, we need to create a vector `w` to store our random white noise values, then 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 10.7.

```
> 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 10.8,

```
> 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)
       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.

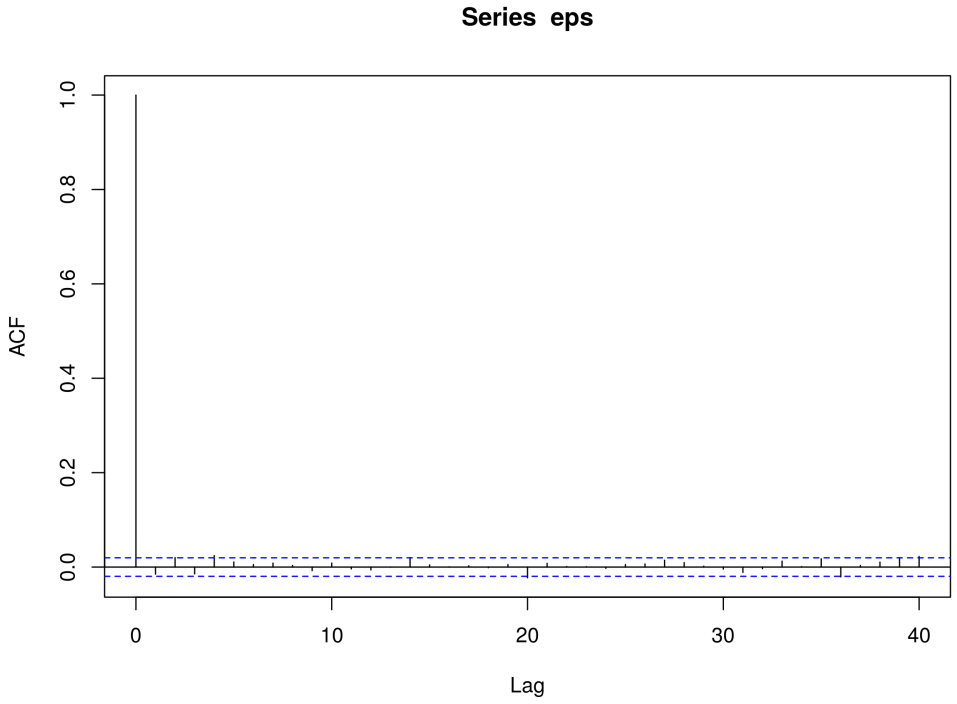

Figure 10.7: Correlogram of a simulated GARCH(1,1) model with $\alpha_0 = 0.2$, $\alpha_1 = 0.5$ and $\beta_1 = 0.3$

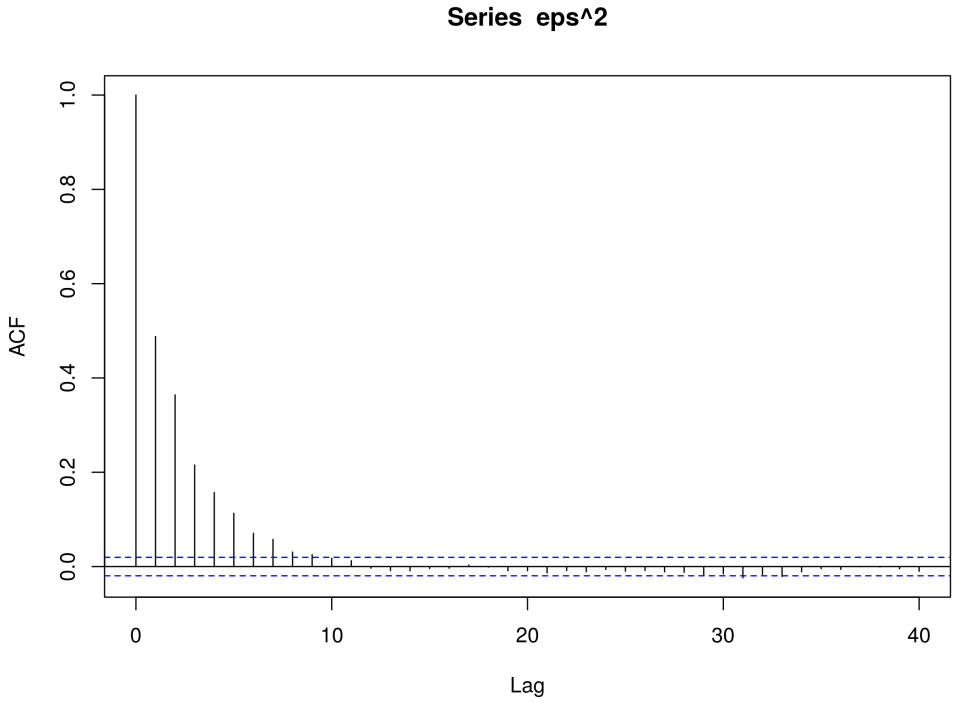

Figure 10.8: Correlogram of a simulated GARCH(1,1) models squared values with $\alpha_0 = 0.2$, $\alpha_1 = 0.5$ and $\beta_1 = 0.3$

10.6.3 Financial Data

Now that we know how to simulate and fit a GARCH model, we want to apply the procedure to some financial series. In particular, let's try fitting ARIMA and GARCH to the FTSE 100 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's plot the values, as given in Figure 10.9.

```
> 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:

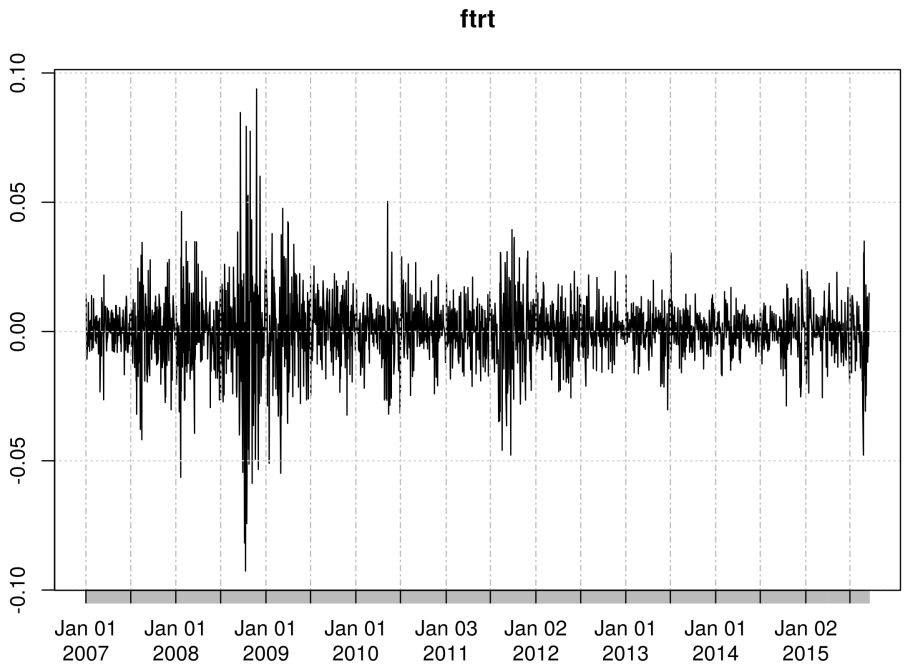

Figure 10.9: Differenced log returns of the daily closing price of the FTSE 100 UK stock index

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 I won't repeat the procedure here, I will simply provide the code:

```
> 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've 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, that is 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 10.10.

```
> acf(resid(ftfinal.arima))
```

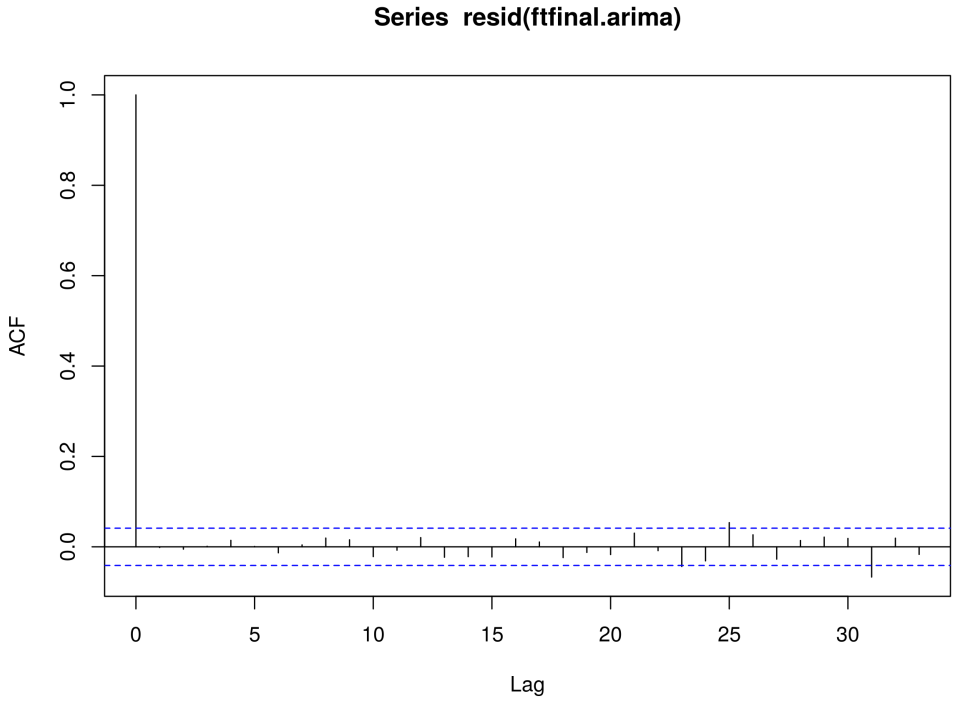

Figure 10.10: Residuals of an ARIMA(4,0,4) fit to the FTSE100 diff log returns

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 10.11.

```
> 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:

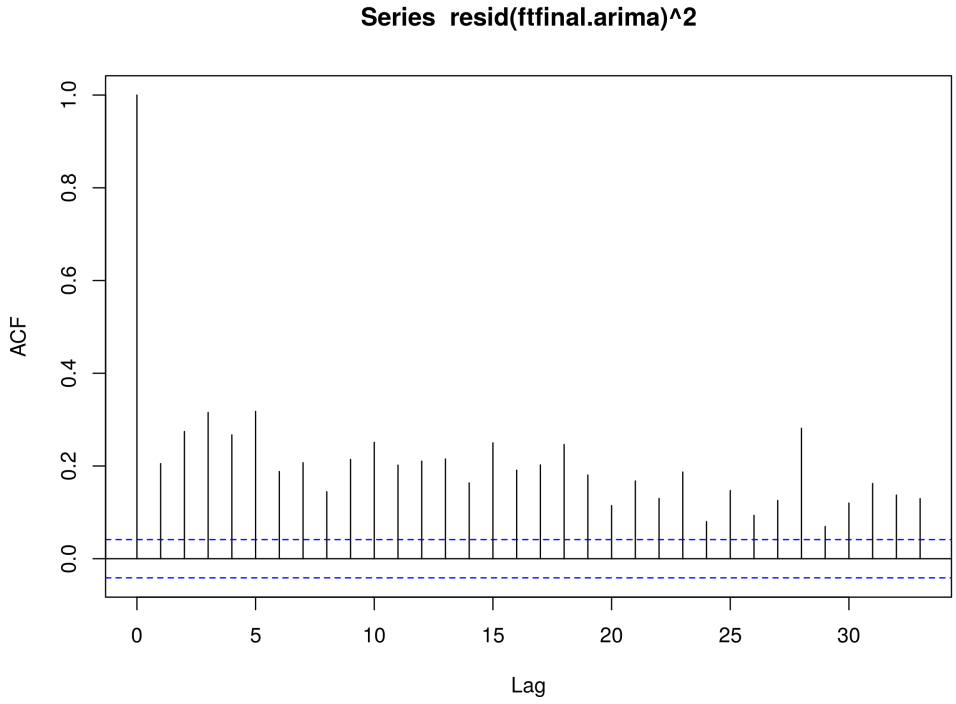

Figure 10.11: Squared residuals of an ARIMA(4,0,4) fit to the FTSE100 diff log returns

```
> ft.garch <- garch(ft, trace=F)
> ft.res <- ft.garch$res[-1]
```

Finally, to test for a good fit we can plot the correlogram of the GARCH residuals and the square GARCH residuals, as given in Figure 10.12.

```
> acf(ft.res)
```

The correlogram looks like a realisation of a discrete white noise process, indicating a good fit. Let's now try the squared residuals, given in Figure 10.13.

```
> 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).

10.7 Next Steps

We are now at the point in our time series education where we have studied ARIMA and GARCH, allowing us to fit a combination of these models to a stock market index, and to 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.

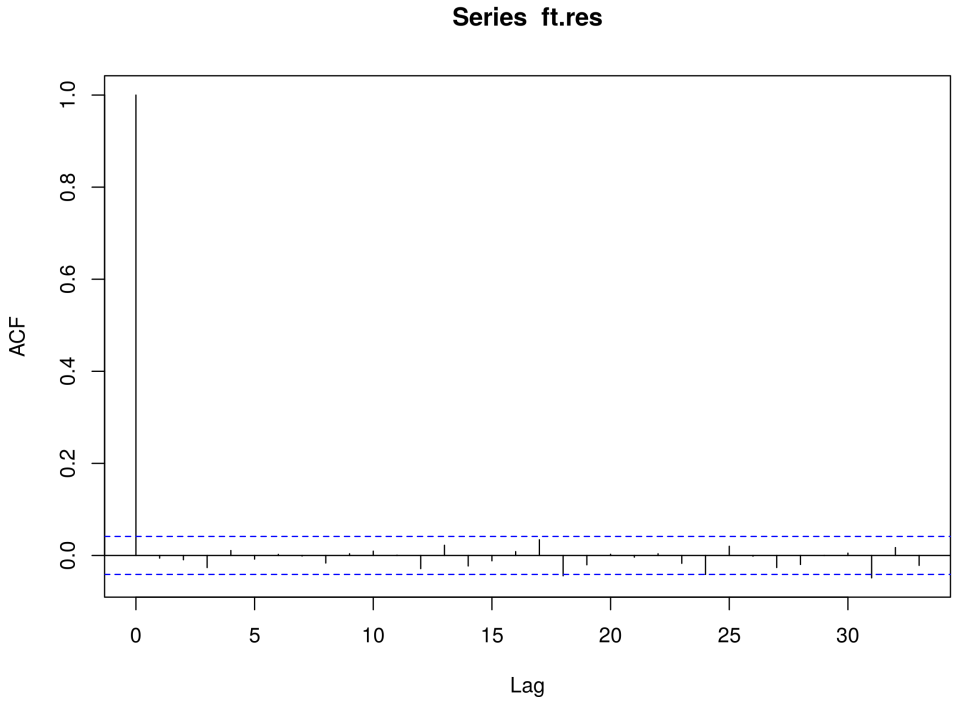

Figure 10.12: Residuals of a GARCH(p,q) fit to the ARIMA(4,0,4) fit of the FTSE100 diff log returns

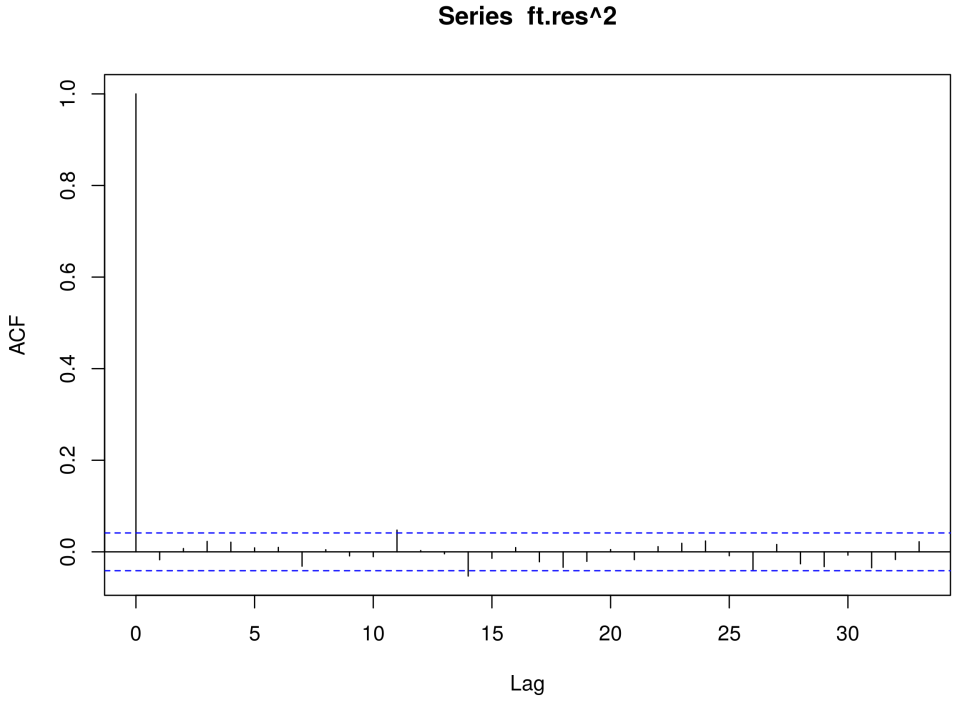

Figure 10.13: Squared residuals of a GARCH(p,q) fit to the ARIMA(4,0,4) fit of the FTSE100 diff log returns

Chapter 11

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 which 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.

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 also discuss in this article.

The Kalman Filter is ubiquitous in engineering control problems, including guidance & navigation, spacecraft trajectory analysis and manufacturing, but 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, but the algorithm is much more general than this and we will look at other use cases.

Generally, there are three types of inference that we are interested in 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 the Kalman Filter to trading situations, such as cointegrated pairs, as well as asset price prediction, later in the book.

We will be making use of a **Bayesian approach** to the problem, as this is a natural statistical framework for allowing us to readily update our beliefs in light of new information, which is precisely the desired behaviour of the Kalman Filter.

I want 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 R do the heavy lifting of solving the model for us, so the verbose notation will not be a problem in practice.

11.1 Linear State-Space Model

Let's 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, but of course, other distributions can be used.

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**:

$$\theta_t = G_t\theta_{t-1} + w_t \tag{11.1}$$

However, this is only half of the story. We also need to discuss the *observations*, that is, what we actually *see*, since the states are hidden to us by system noise.

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*, 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**:

$$y_t = F_t^T\theta_t + v_t \tag{11.2}$$

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:

$$\theta_0 \sim \mathcal{N}(m_0, C_0) \tag{11.3}$$
$$v_t \sim \mathcal{N}(0, V_t) \tag{11.4}$$
$$w_t \sim \mathcal{N}(0, W_t) \tag{11.5}$$

Clearly that is a lot of notation to specify the model. For completeness, I'll summarise all of the terms here to help you get to grips with the model:

- 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've 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.

11.2 The Kalman Filter

This section follows very closely the notation and analysis carried out in Pole et al[40]. I decided it wasn't 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.

11.2.1 A Bayesian Approach

If we recall from the prior chapters on Bayesian inference, Bayes' Rule is given by:

$$P(\theta|D) = P(D|\theta)P(\theta)/P(D) \tag{11.6}$$

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. Once again, 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. However, our current observations are given by y_t. Thus we can say that $D_t = (D_{t-1}, y_t)$. That is, our current knowledge is a mixture of our previous knowledge plus our most recent observation.

Applying Bayes' Rule to this situation gives the following:

$$P(\theta_t|D_{t-1}, y_t) = \frac{P(y_t|\theta_t)P(\theta_t|D_{t-1})}{P(y_t)} \tag{11.7}$$

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 (i.e. updated view 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?

Well, let's specify the terms that we'll be using, from Bayes' Rule above. Firstly, we specify the distributional form of the prior:

$$\theta_t | D_{t-1} \sim \mathcal{N}(a_t, R_t) \tag{11.8}$$

That is, 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's consider the likelihood:

$$y_t | \theta_t \sim \mathcal{N}(F_t^T \theta_t, V_t) \tag{11.9}$$

That is, the likelihood function of the current observation y at time t is distributed as a multivariate normal distribution, with mean $F_t^T \theta_t$ and variance-covariance V_t. We've already outlined these terms in our list above.

Finally we have the posterior of θ_t:

$$\theta_t | D_t \sim \mathcal{N}(m_t, C_t) \tag{11.10}$$

That is, our posterior view of the current state θ at time 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 won't derive where these values actually come from, but we will simply state them. Thankfully we can use library implementations in R to carry out the "heavy lifting" for us:

$$\begin{aligned}
a_t &= G_t m_{t-1} && (11.11)\\
R_t &= G_t C_{t-1} G_t^T + W_t && (11.12)\\
e_t &= y_t - f_t && (11.13)\\
m_t &= a_t + A_t e_t && (11.14)\\
f_t &= F_t^T a_t && (11.15)\\
Q_t &= F_t^T R_t F_t + V_t && (11.16)\\
A_t &= R_t F_t Q_t^{-1} && (11.17)\\
C_t &= R_t - A_t Q_t A_t^T && (11.18)
\end{aligned}$$

Clearly that is a lot of notation! As I said above, we need not worry about the excessive verboseness of the Kalman Filter, as we can simply use libraries in R to calculate the algorithm for us.

So 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.

11.2.2 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's take the expected value of the observation *tomorrow*, given our knowledge of the data *today*:

$$\begin{aligned} E[y_{t+1}|D_t] &= E[F_{t+1}^T\theta_t + v_{t+1}|D_t] && (11.19)\\ &= F_{t+1}^T E[\theta_{t+1}|D_t] && (11.20)\\ &= F_{t+1}^T a_{t+1} && (11.21)\\ &= f_{t+1} && (11.22) \end{aligned}$$

Where does this come from? Let's 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.

$$\begin{aligned} \text{Var}[y_{t+1}|D_t] &= \text{Var}[F_{t+1}^T\theta_t + v_{t+1}|D_t] && (11.23)\\ &= F_{t+1}^T \text{Var}[\theta_{t+1}|D_t]F_{t+1} + V_{t+1} && (11.24)\\ &= F_{t+1}^T R_{t+1}F_{t+1} + V_{t+1} && (11.25)\\ &= Q_{t+1} && (11.26) \end{aligned}$$

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:

$$y_{t+k}|D_t \sim \mathcal{N}(f_{t+k|t}, Q_{t+k|t}) \tag{11.27}$$

Note that I have used some odd notation here - what does it mean to have a subscript of $t+k|t$? Actually, it allows us to write a convenient shorthand for the following:

$$\begin{aligned} f_{t+k|t} &= F_{t+k}^T G^{k-1} a_{t+1} && (11.28)\\ Q_{t+k|t} &= F_{t+k}^T R_{t+k} F_{t+k} + V_{t+k} && (11.29)\\ R_{t+k|t} &= G^{k-1} R_{t+1} (G^{k-1})^T + \sum_{j=2}^{k} G^{k-j} W_{t+j} (G^{k-j})^T && (11.30) \end{aligned}$$

As I've mentioned repeatedly in this chapter, we should not concern ourselves too much with the verboseness of the Kalman Filter and its notation, rather we should think about the overall procedure and its Bayesian underpinnings.

Thus 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. In subsequent chapters later in the book we will actually carry this out for some real financial data and apply it to a predictive trading model.

We will also be able to use the "filter" aspect to provide us with continually updated views on a linear hedge ratio between two cointegrated pairs of assets, such as might be found in a stastical arbitrage strategy.

Part IV

Statistical Machine Learning

Chapter 12

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 unseen data. The latter is known as a models *generalisation performance*.

12.1 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 a classification setting would be attempting to assign a topic to a text document, from a finite set of topics. The bias-variance 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.

12.1.1 Machine Learning Models

As with most of our discussions in machine learning the basic model is given by the following:

$$Y = f(X) + \epsilon \tag{12.1}$$

This states that the response vector, Y, is given as a (potentially non-linear) function, f, of the predictor vector, X, with a set of normally distributed error terms that have mean zero and a standard deviation of one.

What does this mean in practice?

As an example, our vector X could represent a set of lagged financial prices, 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 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 Support Vector Machine or a spline-based method, for instance.

The error terms ϵ represent all of the factors that affect Y that we haven't 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.

In this section we are going to describe how to **measure the performance** of an estimate for the (unknown) function f. As with the time series models discussed previously, 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 make it more *flexible*. Flexibility describes the ability to increase 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 and 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.

12.1.2 Model Selection

When trying to ascertain which statistical machine learning method is best we need some 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.

Let's 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:

$$\tau = \{(X_1, Y_1), ..., (X_N, Y_N)\} \tag{12.2}$$

The 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 $L(Y, \hat{f}(X))$. Its job is to compare the predictions made by $\hat{f}$ at particular values of X to their true values given by Y. A common choice for L is the *absolute error*:

$$L(Y, \hat{f}(X)) = |Y - \hat{f}(X)| \tag{12.3}$$

Another popular choice is the *squared error*:

$$L(Y, \hat{f}(X)) = (Y - \hat{f}(X))^2 \tag{12.4}$$

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:

$$MSE := \frac{1}{N}\sum_{i=1}^{N}(Y_i - \hat{f}(X_i))^2 \tag{12.5}$$

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 $\hat{f}(X_i)$. 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*. It is what we are *really* interested in.

Mathematically, if we have a new prediction value 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**:

$$\text{Test MSE} := \mathbb{E}\left[(Y_0 - \hat{f}(X_0))^2\right] \tag{12.6}$$

Where the expectation is taken across all new unseen predictor-response pairs (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 statistical machine learning methods known as the **bias-variance tradeoff**.

12.1.3 The Bias-Variance Tradeoff

Let's consider a slightly contrived situation where we know the underlying "true" relationship between Y and X, which I will state 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 contrived 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 12.1. 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

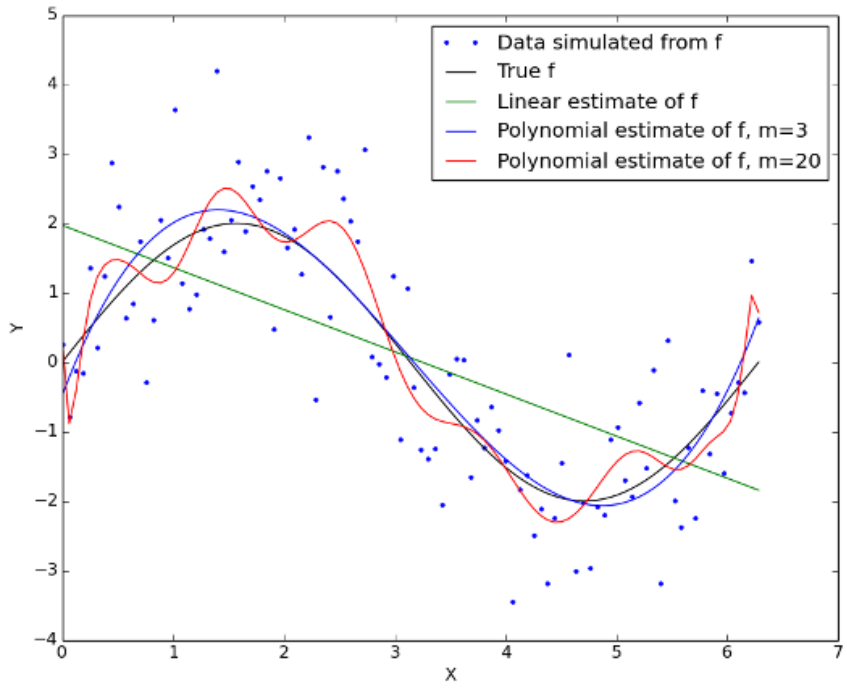

Figure 12.1: Various estimates of the underlying sinusoidal model, $f = \sin$

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 12.2 that the training MSE (given by the green curve) decreases monotonically as the flexibility of the model increases. This makes sense, since the polynomial fit can become as flexible as we need it to in order to minimise the difference between its values and those of the sinusoidal data.

However, if we plot the *test* MSE (given by the blue curve) the situation is vastly different. The test MSE initially decreases as we increase the flexibility of the model but eventually starts to increase again after we introduce a lot of flexibility. Why is this? By allowing the model to be extremely flexible we are letting it fit to "patterns" in the training data.

However, as soon as we introduce new unseen data points in the test set 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*.

In fact, this property of a "u-shaped" test MSE as a function of model flexibility is an intrinsic property of statistical 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 training sets, is given by:

$$\mathbb{E}(Y_0 - \hat{f}(X_0))^2 = \mathrm{Var}(\hat{f}(X_0)) + \left[\mathrm{Bias}\hat{f}(X_0)\right]^2 + \mathrm{Var}(\epsilon) \tag{12.7}$$

The first term on the right hand side is the *variance* of the estimate across many training sets. It determines how much the average model estimation deviates as different training data are tried. In particular, a model with high variance is suggestive that it is overfit to the training data.

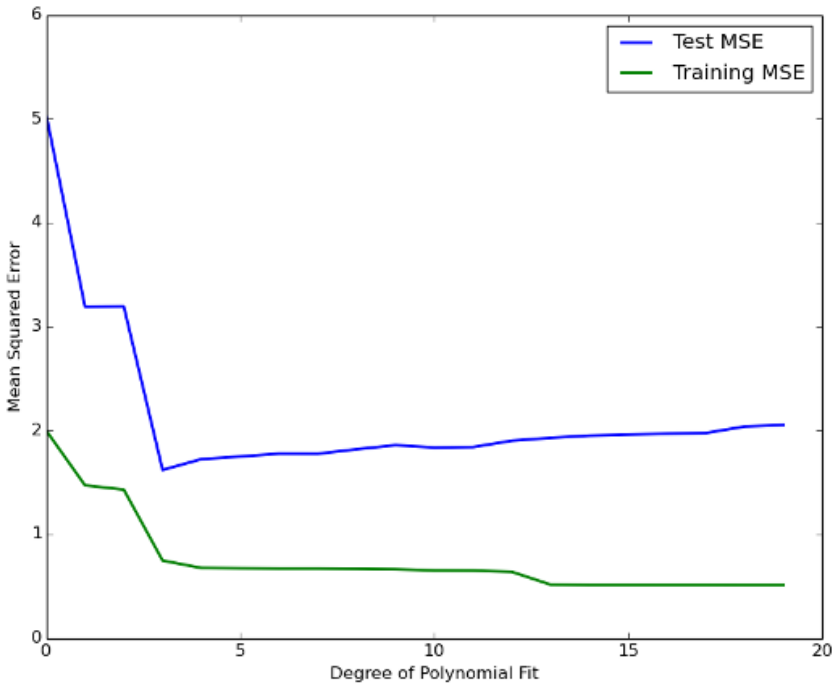

Figure 12.2: Training MSE and Test MSE as a function of model flexibility.

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 can't 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 machine learning is to try and minimise the *expected test MSE*, that is we must choose a statistical 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 next section.

A More Mathematical Explanation

We've now qualitatively outlined the issues surrounding model flexibility, bias and variance. In the following box we are going to carry out a mathematical decomposition of the expected prediction error for a particular model estimate, $\hat{f}(X)$ with prediction vector $X = x_0$ using the latter of our loss functions, the squared-error loss:

The definition of the squared error loss, at the prediction point X_0, is given by:

$$\text{Err}(X_0) = \mathbb{E}\left[\left(Y - \hat{f}(X_0)\right)^2 | X = X_0\right] \tag{12.8}$$

However, we can expand the expectation on the right hand side into three terms:

$$\text{Err}(X_0) = \sigma_\epsilon^2 + \left[\mathbb{E}\hat{f}(X_0) - f(X_0)\right]^2 + \mathbb{E}\left[\hat{f}(X_0) - \mathbb{E}\hat{f}(X_0)\right]^2 \tag{12.9}$$

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 X_0, across all possible training sets, and the true mean value of the underlying function at 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.

$$\begin{aligned} \text{Err}(X_0) &= \sigma_\epsilon^2 + \text{Bias}^2 + \text{Var}(\hat{f}(X_0)) & (12.10) \\ &= \text{Irreducible Error} + \text{Bias}^2 + \text{Variance} & (12.11) \end{aligned}$$

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.

12.2 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, 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.

12.2.1 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.

Again, we can recall from the section above 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 as well as implement the methods using Python on an example forecasting model based on prior historical data.

12.2.2 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 an equity index. Let's choose the FTSE100, which contains a weighted grouping of the one hundred largest (by market capitalisation) publicly traded firms on the London Stock Exchange (LSE). Equally we could pick the S&P500, as in the time series sections, or the DAX.

For this strategy 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 of *vectors*, X and y, which contain the predictor values and the response value respectively. If we consider a daily lag of p days, then X 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 contains only a single value, namely tomorrow's closing price, and is thus a *scalar*. Hence each observation is a tuple (X, y). We will consider a set of n observations corresponding to n days worth of historical pricing information about the FTSE100 (see Fig 12.3).

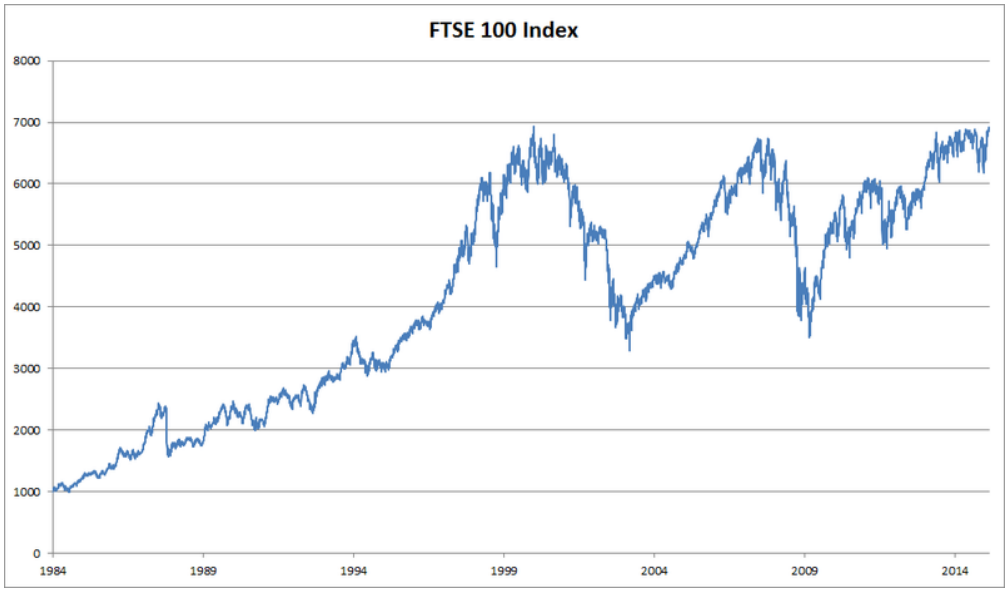

Figure 12.3: The FTSE100 Index - Image Credit: Wikipedia

Our goal is to find a statistical model that attempts to predict the price level of the FTSE100 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.

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.

12.2.3 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 what subtle problems can arises 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.

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**.

12.2.4 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:

$$\text{CV}_k = \frac{1}{k}\sum_{i=1}^{k}\text{MSE}_i \tag{12.12}$$

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, once again, 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.

While LOOCV is beneficial from the point of 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.

12.2.5 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've actually carried out this procedure in my previous book Successful Algorithmic Trading but I'd like to have this section as self-contained as possible, so 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 numpy as np
import pandas as pd
import sklearn

from pandas.io.data import DataReader


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 = 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 xrange(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 xrange(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 the FTSE100 daily prices along some suitable time frame. I have chosen Jan 1st 2004 to Dec 31st 2004. 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 `ftse_lags` we can use the following code:

```
if __name__ == "__main__":
    symbol = "^FTSE"
    start_date = datetime.datetime(2004, 1, 1)
    end_date = datetime.datetime(2004, 12, 31)
    ftse_lags = create_lagged_series(
        symbol, start_date, end_date, lags=5
    )
```

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 FTSE DataFrame that uses five of the prior lagging days 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')
    ax.set_ylim([0.0, 4.0])
```

```
    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 12.4 displays the ten different random splittings of the training and testing data along with the average test MSE (the black dashed line).

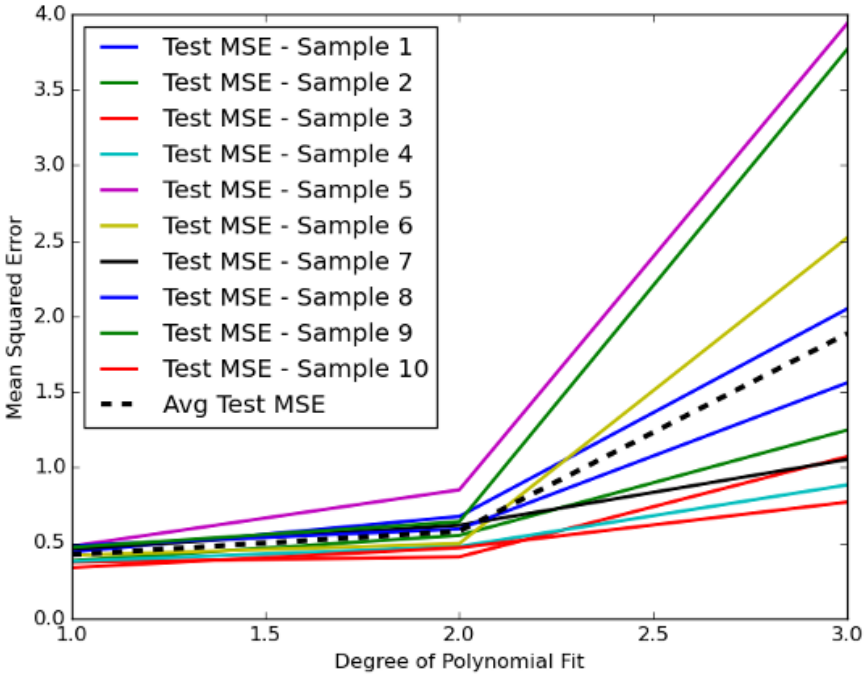

Figure 12.4: Test MSE curves for multiple training-validation splits for a Linear Regression with polynomial features of increasing degree

It is immediately apparent how much variation there is across different random splits into a training and validation set. Since there is not a great deal of predictive signal in using previous days historical close prices of the FTSE100, we see that as the degree of the polynomial features increases the test MSE actually *increases*.

In addition it is clear that the validation set suffers from high variance. The average test MSE for the validation set approach on the degree $d = 3$ model is approximately 1.9.

In order to minimise this issue we will now implement k-fold cross-validation on the same FTSE100 dataset.

12.2.6 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 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 transformation 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')
    ax.set_ylim([0.0, 4.0])
    fig.set_facecolor('white')
    plt.show()
..
..
```

The output is given in Figure 12.5.

Notice that the variation among the error curves is much lower than for the validation set approach. This is the desired effect of carrying out cross-validation. In particular, at $d = 3$ we have a reduced average test error of around 0.8.

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.

12.2.7 Full Python Code

The full Python code for `cross_validation.py` is given below:

```
# cross_validation.py

from __future__ import print_function

import datetime
import pprint

import numpy as np
```

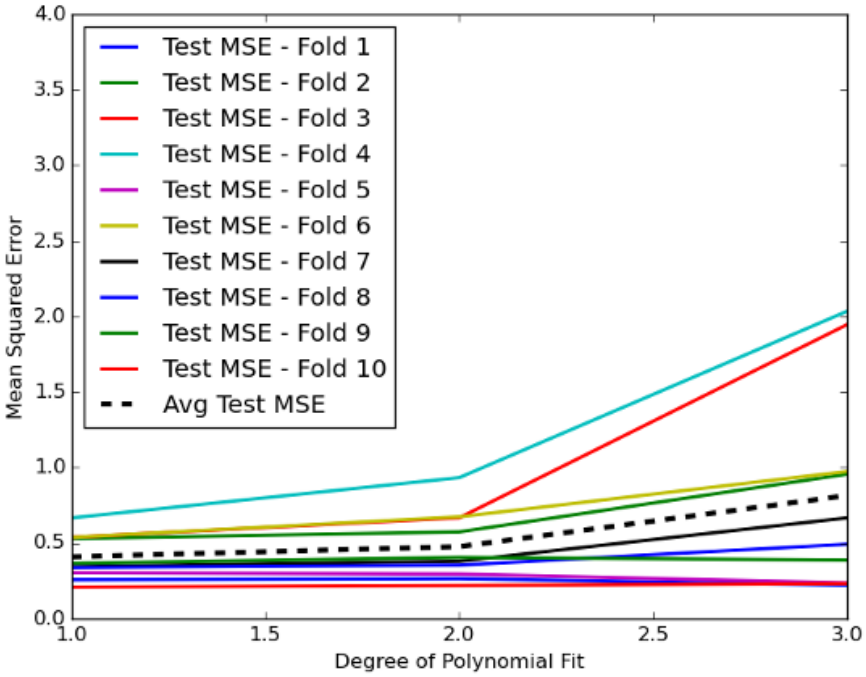

Figure 12.5: Test MSE curves for multiple k-fold cross-validation folds for a Linear Regression with polynomial features of increasing degree

```
import pandas as pd
from pandas.io.data import DataReader
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 = 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 xrange(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 xrange(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,
        label='Avg Test MSE'
    )
    ax.legend(loc=0)
    ax.set_xlabel('Degree of Polynomial Fit')
    ax.set_ylabel('Mean Squared Error')
    ax.set_ylim([0.0, 4.0])
    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')
    ax.set_ylim([0.0, 4.0])
    fig.set_facecolor('white')
    plt.show()


if __name__ == "__main__":
    symbol = "^FTSE"
    start_date = datetime.datetime(2004, 1, 1)
    end_date = datetime.datetime(2004, 12, 31)
    ftse_lags = create_lagged_series(
        symbol, start_date, end_date, lags=5
    )

    # Use five prior days of returns as predictor
    # values, with "Today" as the response
    # (Further days are commented, but can be
    # uncommented to allow extra predictors)
    X = ftse_lags[[
        "Lag1", "Lag2", "Lag3", "Lag4", "Lag5",
        #"Lag6", "Lag7", "Lag8", "Lag9", "Lag10",
        #"Lag11", "Lag12", "Lag13", "Lag14", "Lag15",
        #"Lag16", "Lag17", "Lag18", "Lag19", "Lag20"
```

```
]]
y = ftse_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 13

Kernel Methods and SVMs

13.1 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.

13.1.1 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 corpus of documents that have already been marked as spam or non-spam by humans. SVMs are highly applicable to such situations.

A Support Vector Machine models the situation by creating a *feature space*, which is a finite-dimensional vector space, each dimension of which represents a "feature" of a particular object. In the context of spam or document classification, each "feature" is the prevalence or importance of a particular word.

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 linear 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 "above" or "below" the separation plane, leading to a categorisation (e.g. spam or non-spam). This makes it an example of a *non-probabilistic linear classifier*. It is non-probabilistic because the features in the new objects fully determine its location in feature space and there is no stochastic element involved.

However, much of the benefit of SVMs comes from the fact that they are not restricted to being linear classifiers. Utilising a technique known as the **kernel trick** they can become much more flexible by introducing various types of non-linear decision boundaries.

Formally, in mathematical language, SVMs construct *linear separating hyperplanes* in large finite-dimensional vector spaces. Data points are viewed as $(\vec{x}, y)$ tuples, $\vec{x} = (x_1, \ldots, x_p)$ 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 (i.e. they 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 is clear, or at least clearer. 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.

13.1.2 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[4] but I have repeated and elaborated on them for completeness:

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.

Disadvantages

- $p > n$ - 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, leading 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've outlined the advantages and disadvantages we're 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've formulated the following treatment based on the references provided, making strong use of James et al[32], Hastie et al[27] and the Wikibooks article on SVMs[5]. As this is a textbook on trading strategies, I have made changes to the notation where appropriate and have adjusted the narrative to suit individuals interested in quantitative trading.

13.1.3 Linear Separating Hyperplanes

The linear separating hyperplane is the key geometric entity that is 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 13.1):

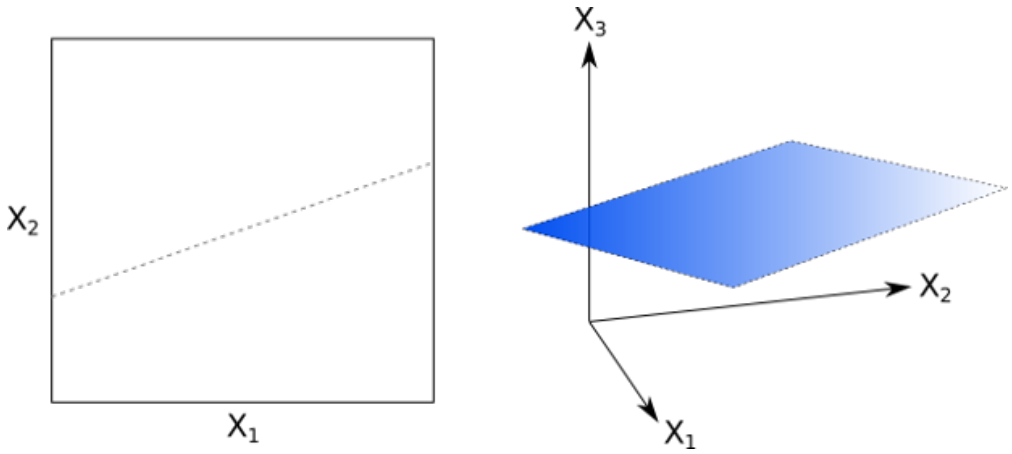

Figure 13.1: One- and two-dimensional hyperplanes

If we consider an element of our p-dimensional feature space, i.e. $\vec{x} = (x_1, ..., x_p) \in \mathbb{R}^p$, then we can mathematically define an affine hyperplane by the following equation:

$$b_0 + b_1 x_1 + ... + b_p x_p = 0 \tag{13.1}$$

$b_0 \neq 0$ gives us an affine plane (i.e. it does not pass through the origin). We can use a more succinct notation for this equation by introducing the summation sign:

$$b_0 + \sum_{j=1}^{p} b_j x_j = 0 \tag{13.2}$$

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:

$$\vec{b} \cdot \vec{x} + b_0 = 0 \tag{13.3}$$

If an element $\vec{x} \in \mathbb{R}^p$ satisfies this relation then it lives on the $p-1$-dimensional hyperplane. This hyperplane splits the p-dimensional feature space into two classification regions (see Figure 13.2):

Elements $\vec{x}$ above the plane satisfy:

$$\vec{b} \cdot \vec{x} + b_0 > 0 \tag{13.4}$$

While those below it satisfy:

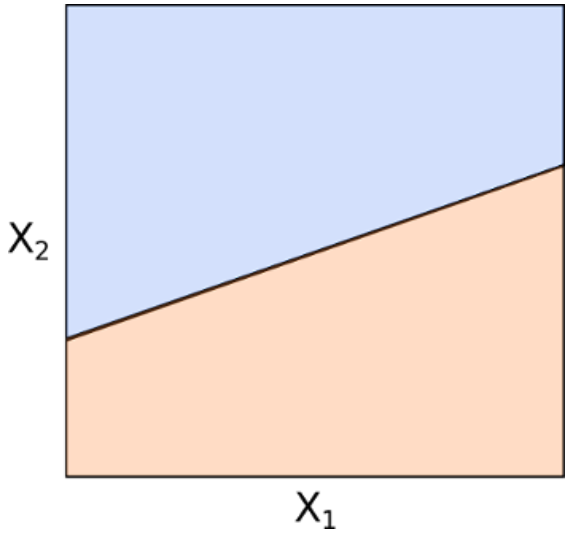

Figure 13.2: Separation of p-dimensional space by a hyperplane

$$\vec{b} \cdot \vec{x} + b_0 < 0 \tag{13.5}$$

The key point here is that it is possible for us to determine which side of the plane any element $\vec{x}$ will fall on by calculating the sign (i.e. whether it is positive or negative) of the expression $\vec{b} \cdot \vec{x} + b_0$. This concept will form the basis of a supervised classification technique.

13.1.4 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 (i.e. separating the words on spacing) that provide *features*. Hence if we take the set of all possible keywords from all of the emails (and remove duplicates), we will be left with p keywords in total.

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*, $\vec{x}_i$, 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 $(\vec{x}_i, y_i)$ representing the features and class labels (keyword lists and spam/non-spam). In addition to the training observations we can provide *test observations*, $\vec{x}^* = (x_1^*, ..., x_p^*)$ 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 solely based 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 13.3). This would mean cleanly separating spam emails from non-spam emails solely by using specific keywords. The following diagram is only showing $p = 2$, while for keyword lists we may have $p > 10^6$. Hence Figures 13.3 are only *representative* of the problem.

This translates into a mathematical separating property of:

$$\vec{b} \cdot \vec{x}_i + b_0 > 0, \text{ if } y_i = 1 \tag{13.6}$$

and

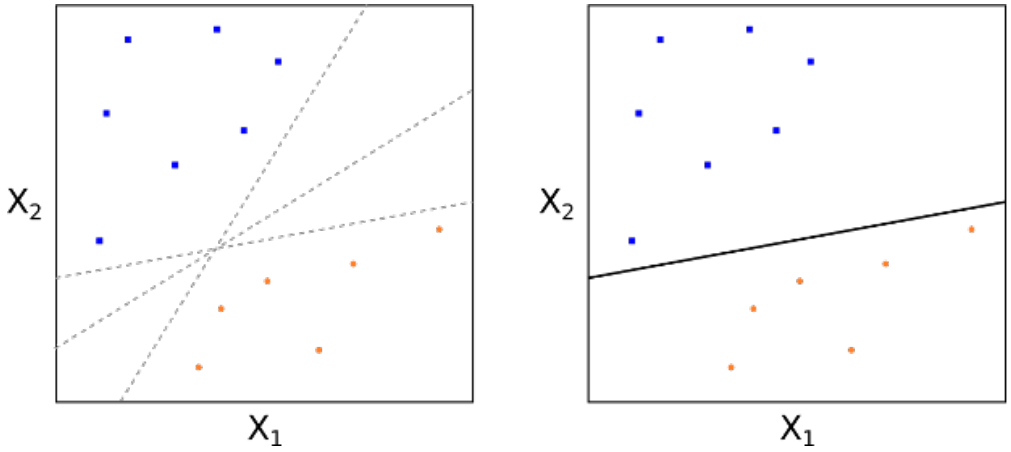

Figure 13.3: Multiple separating hyperplanes; Perfect separation of class data

$$\vec{b} \cdot \vec{x}_i + b_0 < 0, \text{ if } y_i = -1 \tag{13.7}$$

This basically 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 $f(\vec{x})$, with a test observation $\vec{x}^* = (x_1^*, ..., x_p^*)$:

$$f(\vec{x}^*) = \vec{b} \cdot \vec{x}^* + b_0 \tag{13.8}$$

If $f(\vec{x}^*) > 0$ then $y^* = +1$, whereas if $f(\vec{x}^*) < 0$ then $y^* = -1$.

However, this tells us nothing about *how* we go about finding the b_j components of $\vec{b}$, 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, as well as introducing the concept of the **maximal margin hyperplane** and a classifier built on it, known as the **maximal margin classifier**.

13.1.5 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 13.3).

So, 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 that is farthest 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 $\vec{x}_i$ 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 farthest 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 $f(\vec{x}^*)$. 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*. That is, 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 (e.g. in Natural Language Processing applications such as email spam classification). Overfitting here means that the MMH is a very good fit for the *training data* but can perform quite poorly when exposed to *testing data*. I discussed this issue in depth in the previous chapter, 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 $f(\vec{x}^*)$ for any test observation.

If we consider Figure 13.4, 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.

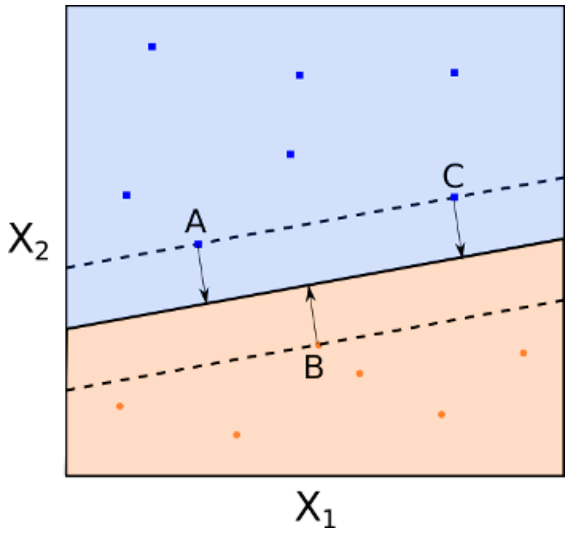

Figure 13.4: Maximal margin hyperplane with support vectors (A, B and C)

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 that lie directly on the margin (but not hyperplane) boundary (see points A, B and C in Figure 13.4). 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" (i.e. the b_j values are fixed).

13.1.6 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, the algorithmic solution to this problem is beyond the scope of the book. Thankfully these optimisation routines are implemented in scikit-learn (actually, via the LIBSVM library[16]). *If you wish to read more about the solution to these algorithmic problems, take a look at Hastie et al (2009)[27] and the Scikit-Learn page on Support Vector Machines[4].*

The procedure for determining a maximal margin hyperplane for a maximal margin classifier is as follows. Given n training observations $\vec{x}_1, ..., \vec{x}_n \in \mathbb{R}^p$ 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 $b_1, ..., b_p$ such that:

$$\sum_{j=1}^{p} b_j^2 = 1 \tag{13.9}$$

and

$$y_i \left(\vec{b} \cdot \vec{x} + b_0 \right) \geq M, \quad \forall i = 1, ..., n \tag{13.10}$$

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 13.5). 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?

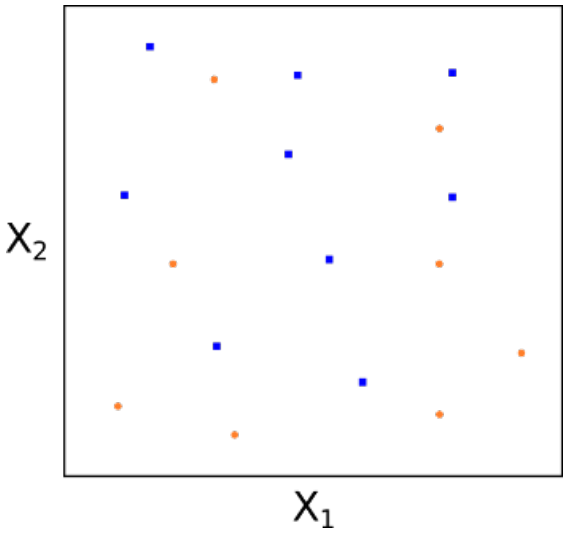

Figure 13.5: No possibility of a true separating hyperplane

Essentially we have to relax the requirement that a separating hyperplane will perfectly separate every training observation on the correct side of the line (i.e. guarantee that it is associated with its true class label), using what is called a **soft margin**. This motivates the concept of a **support vector classifier** (SVC).

13.1.7 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 13.6. 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 *over-fit*:

As we mentioned above also, we could consider a classifier based on a separating hyperplane that doesn't perfectly separate the two classes, but does have a greater robustness to the addition of *new* invididual observations and has 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), hence it provides a "soft" separation. Figure 13.7 demonstrate observations being on the wrong side of the margin and the wrong side of the hyperplane respectively:

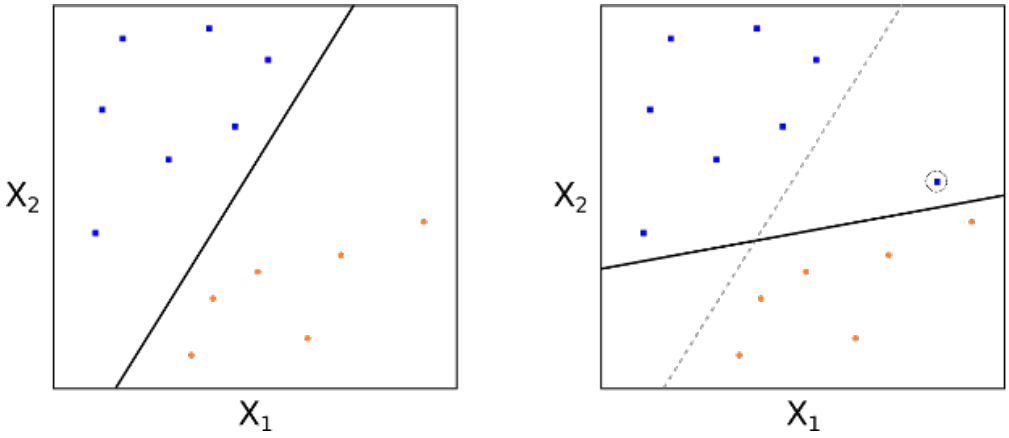

Figure 13.6: Addition of a single point dramatically changes the MMH line

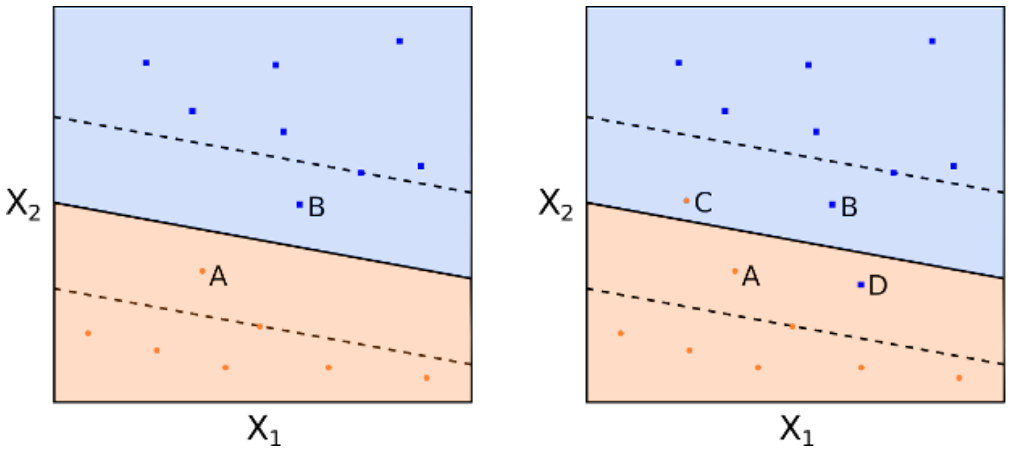

Figure 13.7: Observations on the wrong side of the margin and hyperplane, respectively

As before, an observation is 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 parameter C, known as the *budget*. We wish to maximise M, across $b_1, ..., b_p, \epsilon_1, .., \epsilon_n$ such that:

$$\sum_{j=1}^{p} b_j^2 = 1 \tag{13.11}$$

and

$$y_i \left(\vec{b} \cdot \vec{x} + b_0 \right) \geq M(1 - \epsilon_i), \quad \forall i = 1, ..., n \tag{13.12}$$

and

$$\epsilon_i \geq 0, \quad \sum_{i=1}^{n} \epsilon_i \leq C \tag{13.13}$$

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 13.8 for two differing values of C:

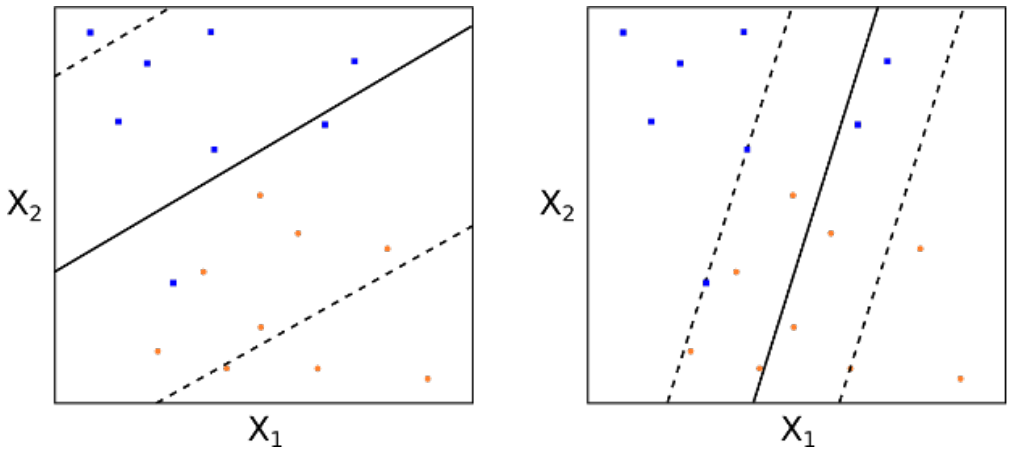

Figure 13.8: Different values of the tuning parameter C

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 x^* we simply calculate the sign of $f(\vec{x}^*) = \vec{b} \cdot \vec{x}^* + b_0$.

This is all well and good for classes that are linearly (or nearly linearly) separated. However, what about separation boundaries that are non-linear? How do we deal with those situations? This is where we can extend the concept of support vector classifiers to support vector machines.

13.1.8 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 13.9. In such a situation a purely linear SVC will have extremely poor performance, simply because the data has no clear linear separation:

Hence SVCs can be useless in highly non-linear class boundary problems.

In order to motivate how an SVM works, we can consider a standard "trick" in 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 $q(\vec{x}) = 0$ where q is a quadratic polynomial function of the original features and hence is a non-linear solution.

This is clearly not restricted to quadratic polynomials. Higher dimensional polynomials, interaction terms and other functional forms, could all be considered. Although the drawback is that it dramatically increases the dimension of the feature space to the point that some algorithms can become untractable.

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

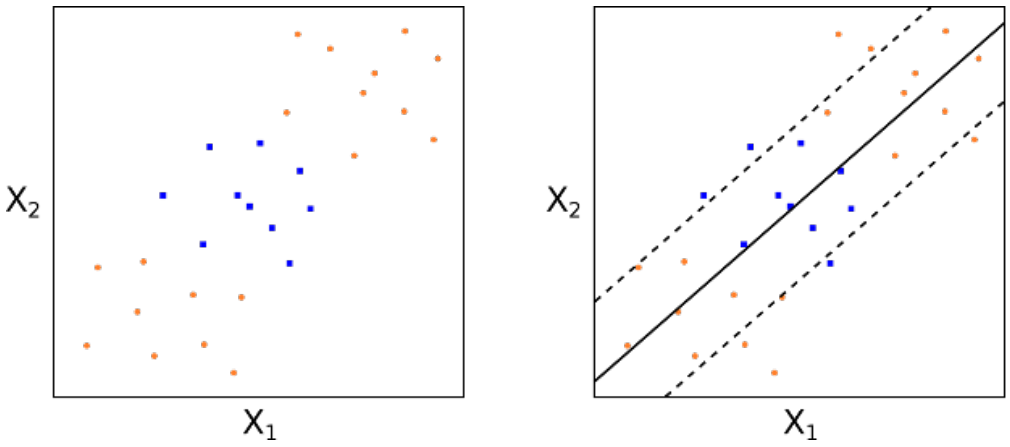

Figure 13.9: No clear linear separation between classes and thus poor SVC performance

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:

$$\langle \vec{u}, \vec{v} \rangle = \sum_{j=1}^{p} u_j v_j \tag{13.14}$$

Hence for two observations an inner product is defined as:

$$\langle \vec{x}_i, \vec{x}_k \rangle = \sum_{j=1}^{p} x_{ij} x_{kj} \tag{13.15}$$

While we won't dwell on the details (since they are beyond the scope of this book), it is possible to show that a linear support vector classifier for a particular observation $\vec{x}$ can be represented as a linear combination of inner products:

$$f(\vec{x}) = b_0 + \sum_{i=1}^{n} \alpha_i \langle \vec{x}, \vec{x}_i \rangle \tag{13.16}$$

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 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:

$$a_i = 0 \text{ if } \vec{x}_i \notin \mathscr{S} \tag{13.17}$$

Hence we can rewrite the representation formula as:

$$f(x) = b_0 + \sum_{i \in \mathscr{S}} a_i \langle \vec{x}, \vec{x}_i \rangle \tag{13.18}$$

This turns out to be a major advantage for computational efficiency.

This now motivates the extension to SVMs. If we consider the inner product $\langle \vec{x}_i, \vec{x}_k \rangle$ and replace it with a more general inner product "kernel" function $K = K(\vec{x}_i, \vec{x}_k)$, we can modify the SVC representation to use non-linear kernel functions and thus modify how we calculate "similarity" between two observations. For instance, to recover the SVC we just take K to be as follows:

$$K(\vec{x}_i, \vec{x}_k) = \sum_{j=1}^{p} x_{ij} x_{kj} \tag{13.19}$$

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:

$$K(\vec{x}_i, \vec{x}_k) = (1 + \sum_{j=1}^{p} x_{ij} x_{kj})^d \tag{13.20}$$

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 13.10).

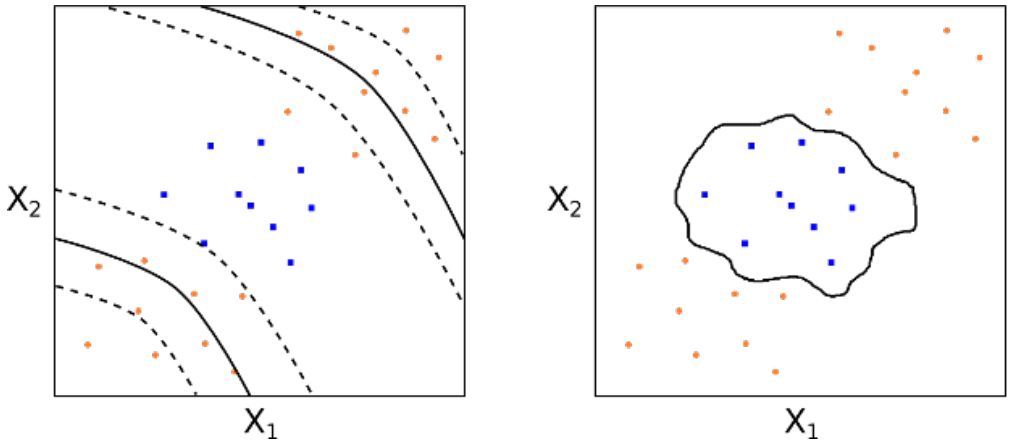

Figure 13.10: A d-degree polynomial kernel; A radial kernel

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 13.10):

$$K(\vec{x}_i, \vec{x}_k) = \exp\left(-\gamma \sum_{j=1}^{p} (x_{ij} - x_{kj})^2\right), \quad \gamma > 0 \tag{13.21}$$

So how do radial kernels work? They clearly differ from polynomial kernels. Essentially if our test observation $\vec{x}^*$ is far from a training observation $\vec{x}_i$ in standard Euclidean distance then the sum $\sum_{j=1}^{p} (x_j^* - x_{ij})^2$ will be large and thus $K(\vec{x}^*, \vec{x}_i)$ will be very small. Hence this particular training observation $\vec{x}_i$ will have almost no effect on where the test observation $\vec{x}^*$ is placed, via $f(\vec{x}^*)$.

Thus the radial kernel has extremely localised behaviour and only nearby training observations to $\vec{x}^*$ will have an impact on its class label.

13.2 Document Classification using Support Vector Machines

In this section we will apply Support Vector Machines to the domain of natural language processing (NLP) for the purposes of sentiment analysis and ultimately automated trade filter or signal generation. Our approach will be to use Support Vector Machines to classify text documents into mutually exclusive groups. Note that this is a *supervised* learning technique. Later in the book we will look at *unsupervised* techniques for similar tasks.

13.2.1 Overview

There are a significant number of steps to carry out between viewing a text document on a web site, say, and using its content as an input to an automated trading strategy to generate trade filters or signals. 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 Python's 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 (such as QSTrader), either by means of filtering other trade signals or generating new ones.
- Continually monitor the system and adjust it as necessary, 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.

In later chapters of the book we will discuss how to integrate such a classifier into a production-ready algorithmic trading system.

Hence, 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.

13.2.2 Supervised Document Classification

Consider a set of text documents. Each document has an associated set of words, which we will call "features". Each of these documents might be associated with a class label that describes 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 (animals) by "learning" from a set of "training" articles, which 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* 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 "supervision" of the training procedure occurs when a *model* is trained or fit to this particular data. This means that the classifier is fed feature vector-class label pairs and it "learns" how representative features are for particular class labels. In the following example we will use the SVM as our model and "train" it on a corpus (a collection of documents) which we will have previously generated.

13.3 Preparing a Dataset for Classification

A famous dataset that is 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" to be used in classification tests, since 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 (http://www.7-zip.org/).

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`[3].

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>
..
..
```

While it may be somewhat laborious to parse data in this manner, especially when compared to the actual machine learning, I can fully reassure you that a large part of a data scientist's or quant researcher's day is in actually getting the data into a format usable by the analysis

software! This particular activity is often jokingly referred to as "data wrangling". Hence I feel it is worth it for you to get 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. In later chapters we will consider this when we study extremely large data sets (particularly tick data).

As stated above, our first goal is to actually create the SGML Parser that will achieve this. 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`, which 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:

```
from __future__ import print_function

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'll 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:

```
..
..

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.

13.3.1 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, of course, include numbers from within the string as additional "words". Once this list of tokens has been created they can be assigned an integer identifier, which allows them to be listed.

Once the list of tokens have been generated, the number of tokens within a document are **counted**. Finally, these tokens are **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* is also fixed and identical.

This means that the classifier now has a set of *features* via the frequency of token occurance. In addition the document token-vector represents a **sample** for the classifier.

In essence, the entire corpus can be represented as a large matrix, each row of which 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 will, however, use this information to enhance the classification process.

13.3.2 Term-Frequency Inverse Document-Frequency

One of the major issues with **vectorisation**, via the **Bag Of Words** representation, is that there is a lot of "noise" in the form of **stop words**, such as "a", "the", "he", "she" etc. These words provide little context to the document but their relatively high frequency will mean that they can mask words that do provide 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 dwell on the calculation of TF-IDF, but if you are interested then you can read the Wikipedia article[6] 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.

Since Scikit-Learn classifiers take two separate data structures for training, namely, y, the vector of class labels or "responses" associated with an ordered set of documents, and, X, the sparse TF-IDF matrix of raw document text, we modify our two-tuple list to create y and 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.

13.4 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. However, we also need a means of evaluating the trained performance of the classifier subsequent to its training phase. We discussed approaches for this in the previous 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 that is retained for training, the "better" the classifier will be because it will have seen more data. 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 about 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, in this case 20%. The `random_state` keyword argument controls the random seed for selecting the partition randomly.

The next step is to actually create the Support Vector Machine and train it. In this instance we are going to use the `SVC` (Support Vector Classifier) class from Scikit-Learn. We give it the parameters $C = 1000000.0$, $\gamma = 0.0$ and choose a **radial kernel**. *To understand where these parameters come from, please take a look at the previous section on Support Vector Machines*

The following code imports the `SVC` class and then fits it on the training data:

```
from sklearn.svm import SVC
..
..
def train_svm(X, y):
    """
```

```
    Create and train the Support Vector Machine.
    """
    svm = SVC(C=1000000.0, gamma=0.0, 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.

13.4.1 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 aren't going to see the entire set of class labels and hence our confusion matrix is smaller in dimension than 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've 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.*

```
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).

In later chapters we will discuss such optimisation procedures and explain how a classifier such as this can be added to a production system in a data science or quantitative finance context.

13.5 Full Code Implementation in Python 3.4.x

Here is the full code for `reuters_svm.py` written in **Python 3.4.x**:

```
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=0.0, 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))
```

13.5.1 Biblographic Notes

Originally, SVMs were invented by Vapnik[47], while the current standard "soft margin" approach is due to Cortes[17]. My treatment of the material follows, and is strongly influenced by, the excellent statistical machine learning texts of James et al[32] and Hastie et al[27].

Part V

Quantitative Trading Strategies

Chapter 14

Introduction to QSTrader

In this chapter I want to briefly introduce the QSTrader library, which will eventually be used for all of the trading strategies outlined in this book.

14.1 Backtesting vs Live Trading

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.

Unfortunately, the performance of a deployed strategy is often significantly worse than that of a backtested system. There are many reasons for this:

- **Transaction costs** - These includes 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.
- **Alpha decay** - This describes the concept of the strategy being replicated by multiple users over time, thus "arbitraging away" the edge.

The most common reason for significant underperformance when compared to a backtest is due to incomplete transaction cost modelling in the backtest. Spread, slippage, fees and market impact all contribute to reduced profitability when a strategy is traded live.

In this book I want the historical backtested performance of trading strategies presented to be as realistic as possible, in order to allow you, the reader (and potential trader), to make the most informed decision possible as to what trading these strategies will be like.

For this reason I decided back in November 2015 to begin writing a new backtesting and live trading engine, called QSTrader, that would attempt to mitigate these issues. I wanted to make QSTrader freely available under a permissive open source license so that anyone could contribute to the codebase or use it for whatever purpose they wished. As with most software on QuantStart, QSTrader is written primarily in Python, which makes it straightforwardly cross-platform compatible.

QSTrader is strongly influenced by the previously developed QSForex software, also available under a permissive open-source license, with the exception that it will not only support the OANDA RESTful API, but also the Interactive Brokers API via swigpy. The eventual goal is to allow trading with Equities, ETFs and Forex all 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 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 myself and the community. As new modules are released, the book (and strategies) will be updated.

14.2 Design Considerations

The design of QSTrader is equivalent to the type of customised algorithmic trading stack that might be found in a small quantitative hedge fund manager. Thus, I consider the end goal of this project to be a fully open-source, but institutional grade, production-ready portfolio and order management system, with risk management layers across positions, portfolios and the infrastructure as a whole.

QSTrader will be end-to-end automated, meaning that minimal human intervention is necessary for the system to trade once it is set "live". It is impossible to completely eliminate human intervention, especially when it comes to input data quality, such as with erroneous ticks, but it is certainly possible to have the system running in an automated fashion most of the time.

14.2.1 Quantitative Trading Considerations

The design calls for the infrastructure to mirror that which might be found in a small quant fund or family office quant arm. It will be highly modular and loosely coupled. The main components are the data store (securities master), signal generator, portfolio/order management system, risk layer and brokerage interface.

The following is a list of "institutional grade" components that the system will contain:

- **Data Provider Integration** - The first major component involves interacting with a set of data providers, usually via some form of API. Typical providers of data include Quandl, DTN IQFeed and Interactive Brokers.

- **Data Ingestion and Cleaning** - In between data storage and download it is necessary to incorporate a filtration and cleansing layer that will only store data once it passes certain checks. It will flag "bad" data and note if data is unavailable.

- **Pricing Data Storage** - There will be a need for an intraday securities master database, storing symbols as well as price values obtained from a brokerage or data provider.

- **Trading Data Storage** - Orders, trades and portfolio states will need to be stored over time. Object serialisation and persistence can be used for this, such as with Python's `pickle` library.

- **Configuration Data Storage** - Time-dependent configuration information will need to be stored for historical reference in a database, either in tabular format or, once again, in pickled format.

- **Research/Backtesting Environment** - The research and backtesting environments will need to hook into the securities master and ultimately use the same trading logic as live trading in order to generate realistic backtests.

- **Signal Generation** - The techniques of Bayesian statistics, time series analysis and machine learning will be used within a "signal generator" class to produce trading recommendations to our portfolio engine.

- **Portfolio/Order Management** - The "heart" of the system will be the portfolio and order management system (OMS) which will receive signals from the signal generator and use them as "recommendations" for constructing orders. The OMS will communicate directly with the risk management component in order to determine how these orders should be constructed.

- **Risk Management** - The risk manager will provide a "veto" or modification mechanism for the OMS, such that sector-specific weightings, leverage constraints, broker margin availability and average daily volume limits are kept in place. The risk layer will also provide "umbrella hedge" situations, providing market- or sector-wide hedging capability to the portfolio.
- **Brokerage Interface** - The brokerage interface will consist of the raw interface code to the the broker API (in this case the C++ API of Interactive Brokers) as well as the implementation of multiple order types such as market, limit, stop etc.
- **Algorithmic Execution** -Automated execution algorithms will eventually be developed in order to mitigate market impact effects for strategies that trade larger amounts or on small-cap stocks.
- **Accounting and P&L** - Accounting is a tricky concept in quantitative trading. A lot of time will be spent considering how to correctly account for PnL in a professional trading system.

14.3 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`.

Chapter 15

ARIMA+GARCH Trading Strategy on Stock Market Indexes Using R

In this chapter we will apply the knowledge gained in the chapters on linear 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.

15.1 Strategy Overview

If you've skipped ahead directly to the strategy section, then great, you can dive in! However, despite the fact that the idea of 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, as I provided in the previous book, *Successful Algorithmic Trading*. Hence the performance achieved in a real trading system would likely be slightly less than you might achieve here, due to commission and slippage.

15.2 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 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 don't 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!) and as such we are really fitting an ARMA model, rather than an ARIMA.

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:

```
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.*

15.3 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". Finally, we 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 15.1:

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, let's consider the fact that the ARMA model was only published in 1951. It wasn't really widely utilised until the 1970's when Box & Jenkins[12] discussed it in their book.

Secondly, the ARCH model wasn't discovered (publicly!) until the early 80s, by Engle[21], and GARCH itself was published by Bollerslev[11] 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 15.2.

As you can see the equity curve remains below a Buy & Hold strategy for almost three years, but during the stock market crash of 2008/2009 it does exceedingly well. This makes sense

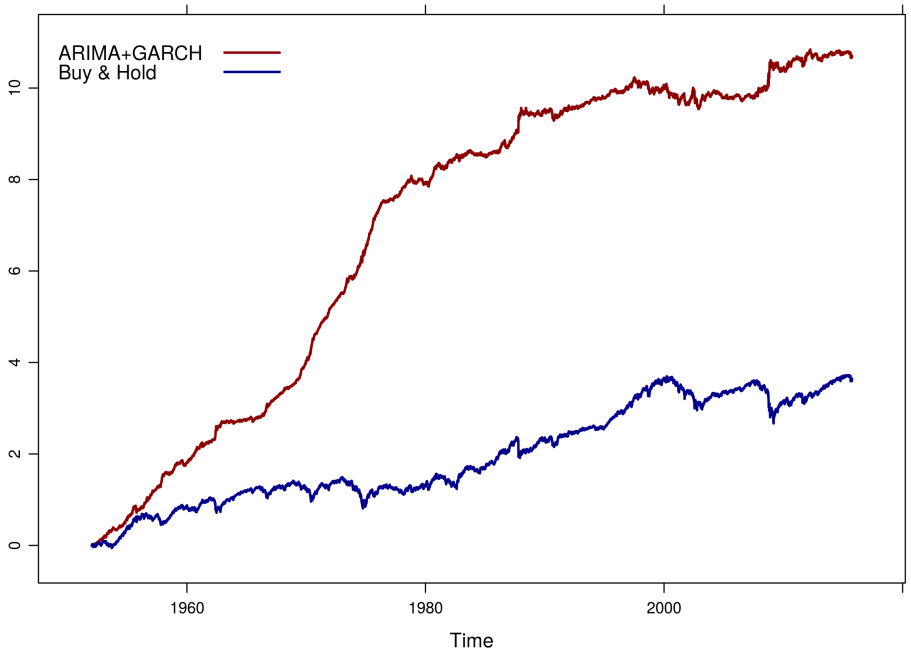

Figure 15.1: Equity curve of ARIMA+GARCH strategy vs "Buy & Hold" for the S&P500 from 1952

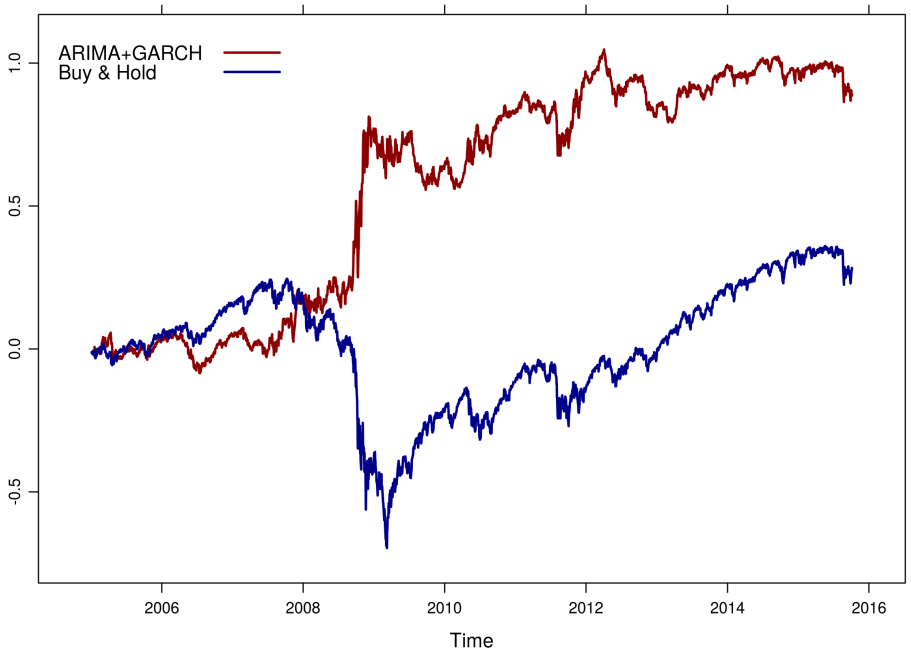

Figure 15.2: Equity curve of ARIMA+GARCH strategy vs "Buy & Hold" for the S&P500 from 2005 until today

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.

15.4 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)
```

Bibliography

[1] Wikipedia: Standard generalized markup language. http://en.wikipedia.org/wiki/Standard_Generalized_Markup_Language, 2015.

[2] Pymc3: Probabilistic programming in python. https://github.com/pymc-devs/pymc3, 2016.

[3] Python htmlparser. https://docs.python.org/2/library/htmlparser.html, 2016.

[4] Scikit-learn: Support vector machines. http://scikit-learn.org/stable/modules/svm.html, 2016.

[5] Wikibooks: Support vector machines. https://en.wikibooks.org/wiki/Support_Vector_Machines, 2016.

[6] Wikipedia: Term frequency-inverse document frequency. http://en.wikipedia.org/wiki/Tf%E2%80%93idf, 2016.

[7] Arnold, G. *Financial Times Guide to the Financial Markets.* Financial Times/Prentice Hall, 2011.

[8] Barber, D. *Bayesian Reasoning and Machine Learning.* Cambridge University Press, 2012.

[9] Bird, S., Klein, E., and Loper, E. *Natural Language Processing with Python.* O'Reilly Media, 2009.

[10] Bollen, J., Mao, H., and Zeng, X. Twitter mood predicts the stock market. *CoRR*, abs/1010.3003, 2010.

[11] Bollerslev, T. Generalized autoregressive conditional heteroskedasticity. *Journal of Econometrics*, 31(3):307–327, 1986.

[12] Box, G., Jenkins, G., Reinsel, G., and Ljung, G. *Time Series Analysis: Forecasting and Control, 5th Ed.* Wiley-Blackwell, 2015.

[13] Brockwell, P. and Davis, R. *Time Series: Theory and Methods.* Springer, 2009.

[14] Chan, E. P. *Quantitative Trading: How to Build Your Own Algorithmic Trading Business.* John Wiley & Sons, 2009.

[15] Chan, E. P. *Algorithmic Trading: Winning Strategies And Their Rationale.* John Wiley & Sons, 2013.

[16] Chang, C. and Lin, C. Libsvm: A library for support vector machines. http://www.csie.ntu.edu.tw/~cjlin/papers/libsvm.pdf, 2013.

[17] Cortes, C. and Vapnik, V. Support vector networks. *Machine Learning*, 20(3):273, 1995.

[18] Cowpertwait, P. and Metcalfe, A. *Introductory Time Series with R.* Springer, 2009.

[19] Davidson-Pilon, C. *Probabilistic Programming & Bayesian Methods for Hackers*, 2016.

[20] Duane, S. and et al. Hybrid monte carlo. *Physics Letters B*, 195(2):216–222, 1987.

[21] Engle, R. F. Autoregressive conditional heteroscedasticity with estimates of the variance of united kingdom inflation. *Econometrica*, 50(4):987–1007, 1982.

[22] Gelfand, A. E. and Smith, A. F. M. Sampling-based approaches to calculating marginal densities. *J. Amer. Statist. Assoc.*, 85(140):398–409, 1990.

[23] Gelman, A., Carlin, J., Stern, H., Dunson, D., Vehtari, A., and Rubin, D. *Bayesian Data Analysis, 3rd Ed.* Chapman and Hall/CRC, 2013.

[24] 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.

[25] Hamada, M., Wilson, A., Reese, C. S., and Martz, H. *Bayesian Reliability.* Springer, 2008.

[26] Harris, L. *Trading and Exchanges: Market Microstructure for Practitioners.* Oxford University Press, 2002.

[27] Hastie, T., Tibshirani, R., and Friedman, J. *The Elements of Statistical Learning: Data Mining, Inference and Prediction, 2nd Ed.* Springer, 2011.

[28] Hastings, W. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57:97–109, 1970.

[29] Hoffman, M. D. and Gelman, A. The no-u-turn sampler: Adaptively setting path lengths in hamiltonian monte carlo.

[30] Hyndman, R. J. and Khandakar, Y. Automatic time series forecasting: the forecast package for R. *Journal of Statistical Software*, 26(3):1–22, 2008.

[31] Hyndman, R. J. *forecast: Forecasting functions for time series and linear models*, 2015. R package version 6.2.

[32] James, G., Witten, D., Hastie, T., and Tibshirani, R. *An Introduction to Statistical Learning: with applications in R.* Springer, 2013.

[33] Johnson, B. *Algorithmic Trading & DMA: An introduction to direct access trading strategies.* 4Myeloma Press, 2010.

[34] Kruschke, J. *Doing Bayesian Data Analysis: A Tutorial with R, JAGS, and Stan, 2nd Ed.* Academic Press, 2015.

[35] McKinney, W. *Python for Data Analysis.* O'Reilly Media, 2012.

[36] Metropolis, N. and et al. Equations of state calculations by fast computing machines. *J. Chem. Phys.*, 21:1087–1092, 1953.

[37] Narang, R. K. *Inside The Black Box: The Simple Truth About Quantitative and High-Frequency Trading, 2nd Ed.* John Wiley & Sons, 2013.

[38] Pardo, R. *The Evaluation and Optimization of Trading Strategies, 2nd Ed.* John Wiley & Sons, 2008.

[39] 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.

[40] Pole, A., West, M., and Harrison, J. *Applied Bayesian Forecasting and Time Series Analysis, 2nd Ed.* Chapman and Hall/CRC, 2011.

[41] 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.

[42] Russell, M. A. *21 Recipes for Mining Twitter.* O'Reilly Media, 2011.

[43] Russell, M. A. *Mining the Social Web, 2nd Ed.* O'Reilly Media, 2013.

[44] Sedar, J. Bayesian inference with pymc3 - part 1. http://blog.applied.ai/bayesian-inference-with-pymc3-part-1/, 2016.

[45] Sinclair, E. *Volatility Trading, 2nd Ed.* John Wiley & Sons, 2013.

[46] Tsay, R. *Analysis of Financial Time Series, 3rd Ed.* Wiley-Blackwell, 2010.

[47] Vapnik, V. *The Nature of Statistical Learning Theory.* Springer, 1996.

[48] Wiecki, T. The inference button: Bayesian glms made easy with pymc3. http://twiecki.github.io/blog/2013/08/12/bayesian-glms-1/, 2013.

[49] 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.

[50] Wiecki, T. Mcmc sampling for dummies. http://twiecki.github.io/blog/2015/11/10/mcmc-sampling/, 2015.

[51] Wilmott, P. *Paul Wilmott Introduces Quantitative Finance, 2nd Ed.* John Wiley & Sons, 2007.