A STEP-BY-STEP GUIDE TO
QUANTITATIVE STRATEGIES

SUCCESSFUL ALGORITHMIC TRADING

Applying the scientific method
for profitable trading results.

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

Introducing Algorithmic Trading

Chapter 1

Introduction to the Book

1.1 Introduction to QuantStart

QuantStart was founded by Michael Halls-Moore, in 2010, to help junior quantitative analysts (QAs) find jobs in the tough economic climate. Since then the site has evolved to become a substantial resource for quantitative finance. The site now concentrates on algorithmic trading, but also discusses quantitative development, in both Python and C++.

Since March 2010, QuantStart has helped over 200,000 visitors improve their quantitative finance skills. You can always contact QuantStart by sending an email to mike@quantstart.com.

1.2 What is this Book?

Successful Algorithmic Trading has been written to teach retail discretionary traders and trading professionals, with basic programming skills, how to create fully automated *profitable* and *robust* algorithmic trading systems using the Python programming language. The book describes the nature of an algorithmic trading system, how to obtain and organise financial data, the concept of backtesting and how to implement an execution system. The book is designed to be extremely practical, with liberal examples of Python code throughout the book to demonstrate the principles and practice of algorithmic trading.

1.3 Who is this Book For?

This book has been written for both retail traders and professional quants who have some basic exposure to programming and wish to learn how to apply modern languages and libraries to algorithmic trading. It is designed for those who enjoy self-study and can learn by example. The book is aimed at individuals interested in actual programming and implementation, as I believe that real success in algorithmic trading comes from fully understanding the implementation details.

Professional quantitative traders will also find the content useful. Exposure to new libraries and implementation methods may lead to more optimal execution or more accurate backtesting.

1.4 What are the Prerequisites?

The book is relatively self-contained, but does assume a familiarity with the basics of trading in a discretionary setting. The book does not require an extensive programming background, but basic familiarity with a programming language is assumed. You should be aware of elementary programming concepts such as variable declaration, flow-control (if-else) and looping (for/while).

Some of the trading strategies make use of statistical machine learning techniques. In addition, the portfolio/strategy optimisation sections make extensive use of search and optimisation

algorithms. While a deep understanding of mathematics is not absolutely necessary, it will make it easy to understand how these algorithms work on a conceptual level.

If you are rusty on this material, or it is new to you, have a look at the QuantStart reading list.

1.5 Software/Hardware Requirements

Quantitative trading applications in Python can be developed in Windows, Mac OSX or Linux. This book is agnostic to the operating system so it is best to use whatever system you are comfortable with. I do however recommend Mac OSX or Linux (I use Ubuntu), as I have found installation and data management to be far more straightforward.

In order to write Python programs you simply need access to a text editor (preferably with syntax highlighting). On Windows I tend to use Notepad++. On Mac OSX I make use of SublimeText. On Ubuntu I tend to use emacs, but of course, you can use vim.

The code in this book will run under both Python version 2.7.x (specifically 2.7.6 on my Ubuntu 14.04 machine) and Python 3.4.x (specifically 3.4.0 on my Ubuntu 14.04 machine).

In terms of hardware, you will probably want at least 1GB RAM, but more is always better. You'll also want to use a relatively new CPU and plenty of hard disk storage for historical data, depending upon the frequency you intend to trade at. A 200Gb hard disk should be sufficient for smaller data, while 1TB is useful for a wide symbol universe of tick data.

1.6 Book Structure

The book is designed to create a set of algorithmic trading strategies from idea to automated execution. The process followed is outlined below.

- **Why Algorithmic Trading?** - The benefits of using a systematic/algorithmic approach to trading are discussed as opposed to a discretionary methodology. In addition the different approaches taken to algorithmic trading are shown.
- **Trading System Development** - The process of developing an algorithmic trading system is covered, from hypothesis through to live trading and continual assessment.
- **Trading System Design** - The actual components forming an algorithmic trading system are covered. In particular, signal generation, risk management, performance measurement, position sizing/leverage, portfolio optimisation and execution.
- **Trading System Environment** - The installation procedure of all Python software is carried out and historical data is obtained, cleaned and stored in a local database system.
- **Time Series Analysis** - Various time series methods are used for forecasting, mean-reversion, momentum and volatility identification. These statistical methods later form the basis of trading strategies.
- **Optimisation** - Optimisation/search algorithms are discussed and examples of how they apply to strategy optimisation are considered.
- **Performance Measurement** - Implementations for various measures of risk/reward and other performance metrics are described in detail.
- **Risk Management** - Various sources of risk affecting an algorithmic trading system are outlined and methods for mitigating this risk are provided.
- **Trading Strategy Implementation** - Examples of trading strategies based off statistical measures and technical indicators are provided, along with details of how to optimise a portfolio of such strategies.
- **Execution** - Connecting to a brokerage, creating an automated event-based trading infrastructure and monitoring/resilience tools are all discussed.

1.7 What the Book does not Cover

This is not a beginner book on discretionary trading, nor a book filled with "technical analysis" trading strategies. If you have not carried out any trading (discretionary or otherwise), I would suggest reading some of the books on the QuantStart reading list.

It is also not a Python tutorial book, although once again the QuantStart reading list can be consulted. While every effort has been made to introduce the Python code as each example warrants it, a certain familiarity with Python will be extremely helpful.

1.8 Where to Get Help

The best place to look for help is the articles list on QuantStart.com, found at QuantStart.com/articles or by contacting me at mike@quantstart.com. I've written over 140 articles about quantitative finance (and algorithmic trading in particular), so you can brush up by reading some of these.

I also want to say thank you for purchasing the book and helping to support me while I write more content - it is very much appreciated. Good luck with your algorithmic strategies! Now onto some trading...

Chapter 2

What Is Algorithmic Trading?

Algorithmic trading, as defined here, is the use of an automated system for carrying out trades, which are executed in a pre-determined manner via an algorithm specifically *without any human intervention*. The latter emphasis is important. Algorithmic strategies are designed prior to the commencement of trading and are executed without *discretionary* input from human traders.

In this book "algorithmic trading" refers to the retail practice of automated, systematic and quantitative trading, which will all be treated as synonyms for the purpose of this text. In the financial industry "algorithmic trading" generally refers to a class of execution algorithms (such as Volume Weighted Average Price, VWAP) used to optimise the costs of larger trading orders.

2.1 Overview

Algorithmic trading differs substantially from discretionary trading. In this section the benefits and drawbacks of a *systematic* approach will be outlined.

2.1.1 Advantages

Algorithmic trading possesses numerous advantages over discretionary methods.

Historical Assessment

The most important advantage in creating an automated strategy is that its performance can be ascertained on historical market data, which is (hopefully) representative of future market data. This process is known as **backtesting** and will be discussed in significant depth within this book. Backtesting allows the (prior) statistical properties of the strategy to be determined, providing insight into whether a strategy is likely to be profitable in the future.

Efficiency

Algorithmic trading is substantially more efficient than a discretionary approach. With a fully automated system there is no need for an individual or team to be constantly monitoring the markets for price action or news input. This frees up time for the developer(s) of the trading strategy to carry out more research and thus, depending upon capital constraints, deploy more strategies into a portfolio.

Furthermore by automating the risk management and position sizing process, by considering a stable of systematic strategies, it is necessary to automatically adjust leverage and risk factors dynamically, directly responding to market dynamics in real-time. This is not possible in a discretionary world, as a trader is unable to continuously compute risk and must take occasional breaks from monitoring the market.

No Discretionary Input

One of the primary advantages of an automated trading system is that there is (theoretically) no subsequent discretionary input. This refers to modification of trades at the point of execution or while in a position. Fear and greed can be overwhelming motivators when carrying out discretionary trading. In the context of systematic trading it is rare that discretionary input improves the performance of a strategy.

That being said, it is certainly possible for systematic strategies to stop being profitable due to regime shifts or other external factors. In this instance judgement is required to modify parameters of the strategy or to retire it. Note that this process is still devoid of interfering with individual trades.

Comparison

Systematic strategies provide statistical information on both historical and current performance. In particular it is possible to determine equity growth, risk (in various forms), trading frequency and a myriad of other metrics. This allows an "apples to apples" comparison between various strategies such that capital can be allocated optimally. This is in contrast to the case where only profit & loss (P&L) information is tracked in a discretionary setting, since it masks potential drawdown risk.

Higher Frequencies

This is a corollary of the efficiency advantage discussed above. Strategies that operate at higher frequencies over many markets become possible in an automated setting. Indeed, some of the most profitable trading strategies operate at the ultra-high frequency domain on limit order book data. These strategies are simply impossible for a human to carry out.

2.1.2 Disadvantages

While the advantages of algorithmic trading are numerous there are some disadvantages.

Capital Requirements

Algorithmic trading generally requires a far larger capital base than would be utilised for retail discretionary trading, this is simply due to the fact that there are few brokers who support automated trade execution that do not also require large account minimums. The most prolific brokerage in the retail automated space is Interactive Brokers, who require an account balance of $10,000. The situation is slowly changing, especially as other brokerages are allowing direct connection via the FIX protocol. Further, the Pattern Day Trader requirements as defined by the Securities and Exchange Commission require a minimum of $25,000 in account equity to be maintained at all times, in certain margin situations. These issues will be discussed at length in the section on Execution.

In addition, obtaining data feeds for intraday quantitative strategies, particularly if using futures contracts, is not cheap for the retail trader. Common retail intraday feeds are often priced in the $300-$500 per month range, with commercial feeds an order of magnitude beyond that. Depending upon your latency needs it may be necessary to co-locate a server in an exchange, which increases the monthly costs. For the interday retail trader this is not necessarily an issue, but it is worth considering. There are also ancillaries such as a more robust internet connection and powerful (and thus expensive) desktop machines to be purchased.

Programming/Scientific Expertise

While certain systematic trading platforms exist, such as Quantopian, QuantConnect and TradeStation, that alleviate the majority of the programming difficulty, some do not yet (as of the time of writing) support live execution. TradeStation is clearly an exception in this case. Thus it is a requirement for the algorithmic trader to be relatively proficient both in programming and scientific modelling.

I have attempted to demonstrate a wide variety of strategies, the basis of which are nearly always grounded in a manner that is straightforward to understand. However, if you do possess numerical modelling skills then you will likely find it easier to make use of the statistical time series methods present in the Modelling section. The majority of the techniques demonstrated have already been implemented in external Python libraries, which saves us a substantial amount of development work. Thus we are "reduced" to tying together our data analysis and execution libraries to produce an algorithmic trading system.

2.2 Scientific Method

The design of trading strategies within this book is based solely on the principles of the *scientific method*. The process of the scientific method begins with the formulation of a **question**, based on observations. In the context of trading an example would be "Is there a relationship between the SPDR Gold Shares ETF (GLD) and the Market Vectors Gold Miners ETF (GDX)?". This allows a **hypothesis** to be formed that may explain the observed behaviour. In this instance a hypothesis may be "Does the spread between GLD and GDX have mean-reverting behaviour?". The *null hypothesis* is that there is no mean-reverting behaviour, i.e. the price spread is a *random walk*.

After formulation of a hypothesis it is up to the scientist to disprove the null hypothesis and demonstrate that there is indeed mean reverting behaviour. To carry this out a **prediction** must be defined. Returning to the GLD-GDX example the prediction is that the time series representing the spread of the two ETFs is *stationary*. In order to prove or disprove the hypothesis the prediction is subject to **testing**. In the case of GLD-GDX this means applying statistical stationarity tests such as the Augmented Dickey-Fuller, Hurst Exponent and Variance-Ratio Tests (described in detail in subsequent chapters).

The results of the testing procedure will provide a *statistical* answer upon whether the null hypothesis can be rejected at a certain level of confidence. If the null hypothesis is unable to be rejected, which implies that there was no discernible relationship between the two ETFs, it is still possible that the hypothesis is (partially) true. A larger set of data, incorporation of additional information (such as a third ETF affecting the price) can also be tested. This is the process of **analysis**. It often leads to rejection of the null hypothesis, after refinement.

The primary advantage of using the scientific method for trading strategy design is that if the strategy "breaks down" after a prior period of profitability, it is possible to revisit the initial hypothesis and re-evaluate it, potentially leading to a new hypothesis that leads to regained profitability for a strategy.

This is in direct contrast to the *data mining* or *black box* approach where a large quantity of parameters or "indicators" are applied to a time series. If such a "strategy" is initially profitable and then performance deteriorates it is difficult (if not impossible) to determine why. It often leads to arbitrary application of new information, indicators or parameters that may temporarily lead to profitability but subsequently lead to further performance degradation. In this instance the strategy is usually discarded and the process of "research" continues again.

In this book all trading strategies will be developed with an observation-hypothesis approach.

2.3 Why Python?

The above sections have outlined the benefits of algorithmic trading and the scientific method. It is now time to turn attention to the language of implementation for our trading systems. For this book I have chosen Python. Python is a high-level language designed for speed of development. It possesses a wide array of libraries for nearly any computational task imaginable. It is also gaining wider adoption in the the asset management and investment bank communities.

Here are the reasons why I have chosen Python as a language for trading system research and implementation:

- **Learning** - Python is extremely easy to learn compared to other languages such as C++. You can be extremely productive in Python after only a few weeks (some say days!) of

usage.

- **Libraries** - The main reason to use Python is that it comes with a staggering array of libraries, which significantly reduce time to implementation and the chance of introducing bugs into our code. In particular, we will make use of NumPy (vectorised operations), SciPy (optimisation algorithms), pandas (time series analysis), statsmodel (statistical modelling), scikit-learn (statistical/machine learning), IPython (interactive development) and matplotlib (visualisation).

- **Speed of Development** - Python excels at development speed to the extent that some have commented that it is like writing in "pseudocode". The interactive nature of tools like IPython make strategy research extremely rapid, without sacrificing robustness.

- **Speed of Execution** - While not quite as fast as C++, Python provides scientific computing components which are heavily optimised (via vectorisation). If speed of execution becomes an issue one can utilise Cython and obtain execution speeds similar to C, for a small increase in code complexity.

- **Trade Execution** - Python plugins exist for larger brokers, such as Interactive Brokers (IBypy). In addition Python can easily make use of the FIX protocol where necessary.

- **Cost/License** - Python is free, open source and cross-platform. It will run happily on Windows, Mac OSX or Linux.

While Python is extremely applicable to nearly all forms of algorithmic trading, it cannot compete with C (or lower level languages) in the Ultra-High Frequency Trading (UHFT) realm. However, these types of strategies are well outside the scope of this book!

2.4 Can Retail Traders Still Compete?

It is common, as a beginning algorithmic trader practising at retail level, to **question whether it is still possible to compete** with the large institutional quant funds. In this section it will be argued that due to the nature of the institutional regulatory environment, the organisational structure and a need to maintain investor relations, that funds suffer from certain disadvantages that do not concern retail algorithmic traders.

The capital and regulatory constraints imposed on funds lead to certain predictable behaviours, which are able to be exploited by a retail trader. "Big money" moves the markets, and as such one can dream up many strategies to take advantage of such movements. Some of these strategies will be discussed in later chapters. The comparative advantages enjoyed by the algorithmic trader over many larger funds will now be outlined.

2.4.1 Trading Advantages

There are many ways in which a retail algo trader can compete with a fund on their trading process alone, but there are also some disadvantages:

- **Capacity** - A retail trader has greater freedom to play in smaller markets. They can generate significant returns in these spaces, even while institutional funds can't.

- **Crowding the trade** - Funds suffer from "technology transfer", as staff turnover can be high. Non-Disclosure Agreements and Non-Compete Agreements mitigate the issue, but it still leads to many quant funds "chasing the same trade". Whimsical investor sentiment and the "next hot thing" exacerbate the issue. Retail traders are not constrained to follow the same strategies and so can remain uncorrelated to the larger funds.

- **Market impact** - When playing in highly liquid, non-OTC markets, the low capital base of retail accounts reduces market impact substantially.

- **Leverage** - A retail trader, depending upon their legal setup, is constrained by margin/leverage regulations. Private investment funds do not suffer from the same disadvantage, although they are equally constrained from a risk management perspective.

- **Liquidity** - Having access to a prime brokerage is out of reach of the average retail algo trader. They have to "make do" with a retail brokerage such as Interactive Brokers. Hence there is reduced access to liquidity in certain instruments. Trade order-routing is also less clear and is one way in which strategy performance can diverge from backtests.

- **Client news flow** - Potentially the most important disadvantage for the retail trader is lack of access to client news flow from their prime brokerage or credit-providing institution. Retail traders have to make use of non-traditional sources such as meet-up groups, blogs, forums and open-access financial journals.

2.4.2 Risk Management

Retail algo traders often take a different approach to risk management than the larger quant funds. It is often advantageous to be "small and nimble" in the context of risk.

Crucially, there is no risk management budget imposed on the trader beyond that which they impose themselves, nor is there a compliance or risk management department enforcing oversight. This allows the retail trader to deploy custom or preferred risk modelling methodologies, without the need to follow "industry standards" (an implicit investor requirement).

However, the alternative argument is that this flexibility can lead to retail traders to becoming "sloppy" with risk management. Risk concerns may be built-in to the backtest and execution process, without external consideration given to portfolio risk as a whole. Although "deep thought" might be applied to the alpha model (strategy), risk management might not achieve a similar level of consideration.

2.4.3 Investor Relations

Outside investors are the key difference between retail shops and large funds. This drives all manner of incentives for the larger fund - issues which the retail trader need not concern themselves with:

- **Compensation structure** - In the retail environment the trader is concerned only with absolute return. There are no high-water marks to be met and no capital deployment rules to follow. Retail traders are also able to suffer more volatile equity curves since nobody is watching their performance who might be capable of redeeming capital from their fund.

- **Regulations and reporting** - Beyond taxation there is little in the way of regulatory reporting constraints for the retail trader. Further, there is no need to provide monthly performance reports or "dress up" a portfolio prior to a client newsletter being sent. This is a big time-saver.

- **Benchmark comparison** - Funds are not only compared with their peers, but also "industry benchmarks". For a long-only US equities fund, investors will want to see returns in excess of the S&P500, for example. Retail traders are not enforced in the same way to compare their strategies to a benchmark.

- **Performance fees** - The downside to running your own portfolio as a retail trader are the lack of management and performance fees enjoyed by the successful quant funds. There is no "2 and 20" to be had at the retail level!

2.4.4 Technology

One area where the retail trader is at a significant advantage is in the choice of technology stack for the trading system. Not only can the trader pick the "best tools for the job" as they see fit, but there are no concerns about legacy systems integration or firm-wide IT policies. Newer

languages such as Python or R now possess packages to construct an end-to-end backtesting, execution, risk and portfolio management system with far fewer lines-of-code (LOC) than may be needed in a more verbose language such as C++.

However, this flexibility comes at a price. One either has to build the stack themselves or outsource all or part of it to vendors. This is expensive in terms of time, capital or both. Further, a trader must debug all aspects of the trading system - a long and potentially painstaking process. All desktop research machines and any co-located servers must be paid for directly out of trading profits as there are no management fees to cover expenses.

In conclusion, it can be seen that retail traders possess significant comparative advantages over the larger quant funds. Potentially, there are many ways in which these advantages can be exploited. Later chapters will discuss some strategies that make use of these differences.

Part II

Trading Systems

Chapter 3

Successful Backtesting

Algorithmic backtesting requires knowledge of many areas, including psychology, mathematics, statistics, software development and market/exchange microstructure. I couldn't hope to cover all of those topics in one chapter, so I'm going to split them into two or three smaller pieces. What will we discuss in this section? I'll begin by defining backtesting and then I will describe the basics of how it is carried out. Then I will elucidate upon the biases we touched upon in previous chapters.

In subsequent chapters we will look at the details of strategy implementations that are often barely mentioned or ignored elsewhere. We will also consider how to make the backtesting process more realistic by including the idiosyncrasies of a *trading exchange*. Then we will discuss transaction costs and how to correctly model them in a backtest setting. We will end with a discussion on the *performance* of our backtests and finally provide detailed examples of common quant strategies.

Let's begin by discussing what backtesting is and why we should carry it out in our algorithmic trading.

3.1 Why Backtest Strategies?

Algorithmic trading stands apart from other types of investment classes because we can more reliably provide expectations about future performance from past performance, as a consequence of abundant data availability. The process by which this is carried out is known as **backtesting**.

In simple terms, backtesting is carried out by exposing your particular strategy algorithm to a stream of historical financial data, which leads to a set of **trading signals**. Each *trade* (which we will mean here to be a 'round-trip' of two signals) will have an associated profit or loss. The accumulation of this profit/loss over the duration of your strategy backtest will lead to the total profit and loss (also known as the 'P & L' or 'PnL'). That is the essence of the idea, although of course the "devil is always in the details"!

What are key reasons for backtesting an algorithmic strategy?

- **Filtration** - If you recall from the previous chapter on Strategy Identification, our goal at the initial research stage was to set up a strategy pipeline and then filter out any strategy that did not meet certain criteria. Backtesting provides us with another filtration mechanism, as we can eliminate strategies that do not meet our performance needs.

- **Modelling** - Backtesting allows us to (safely!) test new models of certain market phenomena, such as transaction costs, order routing, latency, liquidity or other *market microstructure* issues.

- **Optimisation** - Although *strategy optimisation* is fraught with biases, backtesting allows us to increase the performance of a strategy by modifying the quantity or values of the parameters associated with that strategy and recalculating its performance.

- **Verification** - Our strategies are often sourced externally, via our *strategy pipeline*. Backtesting a strategy ensures that it has not been incorrectly implemented. Although we will rarely have access to the signals generated by external strategies, we will often have access to the performance metrics such as the Sharpe Ratio and Drawdown characteristics. Thus we can compare them with our own implementation.

Backtesting provides a host of advantages for algorithmic trading. However, it is not always possible to straightforwardly backtest a strategy. In general, as the frequency of the strategy increases, it becomes harder to correctly model the microstructure effects of the market and exchanges. This leads to less reliable backtests and thus a trickier evaluation of a chosen strategy. This is a particular problem where the execution system is the key to the strategy performance, as with ultra-high frequency algorithms.

Unfortunately, backtesting is fraught with biases of all types and we will now discuss them in depth.

3.2 Backtesting Biases

There are many biases that can affect the performance of a backtested strategy. Unfortunately, these biases have a tendency to inflate the performance rather than detract from it. Thus you should always consider a backtest to be an idealised upper bound on the actual performance of the strategy. It is almost impossible to eliminate biases from algorithmic trading so it is our job to minimise them as best we can in order to make informed decisions about our algorithmic strategies.

There are four major biases that I wish to discuss: *Optimisation Bias*, *Look-Ahead Bias*, *Survivorship Bias* and *Cognitive Bias*.

3.2.1 Optimisation Bias

This is probably the most insidious of all backtest biases. It involves adjusting or introducing additional trading parameters until the strategy performance on the backtest data set is very attractive. However, once live the performance of the strategy can be markedly different. Another name for this bias is "curve fitting" or "data-snooping bias".

Optimisation bias is hard to eliminate as algorithmic strategies often involve many parameters. "Parameters" in this instance might be the entry/exit criteria, look-back periods, averaging periods (i.e the moving average smoothing parameter) or volatility measurement frequency. Optimisation bias can be minimised by keeping the number of parameters to a minimum and increasing the quantity of data points in the training set. In fact, one must also be careful of the latter as older training points can be subject to a prior *regime* (such as a regulatory environment) and thus may not be relevant to your current strategy.

One method to help mitigate this bias is to perform a *sensitivity analysis*. This means varying the parameters incrementally and plotting a "surface" of performance. Sound, fundamental reasoning for parameter choices should, with all other factors considered, lead to a smoother parameter surface. If you have a very jumpy performance surface, it often means that a parameter is not reflecting a phenomena and is an artefact of the test data. There is a vast literature on multi-dimensional optimisation algorithms and it is a highly active area of research. I won't dwell on it here, but keep it in the back of your mind when you find a strategy with a fantastic backtest!

3.2.2 Look-Ahead Bias

Look-ahead bias is introduced into a backtesting system when future data is accidentally included at a point in the simulation where that data would not have actually been available. If we are running the backtest chronologically and we reach time point N, then look-ahead bias occurs if data is included for any point $N + k$, where $k > 0$. Look-ahead bias errors can be incredibly subtle. Here are three examples of how look-ahead bias can be introduced:

- **Technical Bugs** - Arrays/vectors in code often have iterators or index variables. Incorrect *offsets* of these indices can lead to a look-ahead bias by incorporating data at $N + k$ for non-zero k.
- **Parameter Calculation** - Another common example of look-ahead bias occurs when calculating optimal strategy parameters, such as with linear regressions between two time series. If the whole data set (including future data) is used to calculate the regression coefficients, and thus retroactively applied to a trading strategy for optimisation purposes, then future data is being incorporated and a look-ahead bias exists.
- **Maxima/Minima** - Certain trading strategies make use of extreme values in any time period, such as incorporating the high or low prices in OHLC data. However, since these maximal/minimal values can only be calculated at the end of a time period, a look-ahead bias is introduced if these values are used -during- the current period. It is always necessary to lag high/low values by at least one period in any trading strategy making use of them.

As with optimisation bias, one must be extremely careful to avoid its introduction. It is often the main reason why trading strategies underperform their backtests significantly in "live trading".

3.2.3 Survivorship Bias

Survivorship bias is a particularly dangerous phenomenon and can lead to significantly inflated performance for certain strategy types. It occurs when strategies are tested on datasets that do not include the full universe of prior assets that may have been chosen at a particular point in time, but only consider those that have "survived" to the current time.

As an example, consider testing a strategy on a random selection of equities before and after the 2001 market crash. Some technology stocks went bankrupt, while others managed to stay afloat and even prospered. If we had restricted this strategy only to stocks which made it through the market drawdown period, we would be introducing a survivorship bias because they have already demonstrated their success to us. In fact, this is just another specific case of look-ahead bias, as future information is being incorporated into past analysis.

There are two main ways to mitigate survivorship bias in your strategy backtests:

- **Survivorship Bias Free Datasets** - In the case of equity data it is possible to purchase datasets that include delisted entities, although they are not cheap and only tend to be utilised by institutional firms. In particular, Yahoo Finance data is NOT survivorship bias free, and this is commonly used by many retail algo traders. One can also trade on asset classes that are not prone to survivorship bias, such as certain commodities (and their future derivatives).
- **Use More Recent Data** - In the case of equities, utilising a more recent data set mitigates the possibility that the stock selection chosen is weighted to "survivors", simply as there is less likelihood of overall stock delisting in shorter time periods. One can also start building a personal survivorship-bias free dataset by collecting data from current point onward. After 3-4 years, you will have a solid survivorship-bias free set of equities data with which to backtest further strategies.

We will now consider certain psychological phenomena that can influence your trading performance.

3.2.4 Cognitive Bias

This particular phenomena is not often discussed in the context of *quantitative* trading. However, it is discussed extensively in regard to more discretionary trading methods. When creating backtests over a period of 5 years or more, it is easy to look at an upwardly trending equity curve, calculate the compounded annual return, Sharpe ratio and even drawdown characteristics and be satisfied with the results. As an example, the strategy might possess a maximum relative

drawdown of 25% and a maximum drawdown duration of 4 months. This would not be atypical for a momentum strategy. It is straightforward to convince oneself that it is easy to tolerate such periods of losses because the overall picture is rosy. However, in practice, it is far harder!

If historical drawdowns of 25% or more occur in the backtests, then in all likelihood you will see periods of similar drawdown in live trading. These periods of drawdown are psychologically difficult to endure. I have observed first hand what an extended drawdown can be like, in an institutional setting, and it is not pleasant - even if the backtests suggest such periods will occur. The reason I have termed it a "bias" is that often a strategy which would otherwise be successful is stopped from trading during times of extended drawdown and thus will lead to significant underperformance compared to a backtest. Thus, even though the strategy is algorithmic in nature, psychological factors can still have a heavy influence on profitability. The takeaway is to ensure that if you see drawdowns of a certain percentage and duration in the backtests, then you should expect them to occur in live trading environments, and will need to persevere in order to reach profitability once more.

3.3 Exchange Issues

3.3.1 Order Types

One choice that an algorithmic trader must make is how and when to make use of the different *exchange orders* available. This choice usually falls into the realm of the *execution system*, but we will consider it here as it can greatly affect strategy backtest performance. There are two types of order that can be carried out: **market orders** and **limit orders**.

A market order executes a trade immediately, irrespective of available prices. Thus large trades executed as market orders will often get a mixture of prices as each subsequent limit order on the opposing side is filled. Market orders are considered *aggressive* orders since they will almost certainly be filled, albeit with a potentially unknown cost.

Limit orders provide a mechanism for the strategy to determine the worst price at which the trade will get executed, with the caveat that the trade may not get filled partially or fully. Limit orders are considered *passive* orders since they are often unfilled, but when they are a price is guaranteed. An individual exchange's collection of limit orders is known as the **limit order book**, which is essentially a queue of buy and sell orders at certain sizes and prices.

When backtesting, it is essential to model the effects of using market or limit orders correctly. For high-frequency strategies in particular, backtests can significantly outperform live trading if the effects of market impact and the limit order book are not modelled accurately.

3.3.2 Price Consolidation

There are particular issues related to backtesting strategies when making use of daily data in the form of Open-High-Low-Close (OHLC) figures, especially for equities. Note that this is precisely the form of data given out by Yahoo Finance, which is a very common source of data for retail algorithmic traders!

Cheap or free datasets, while suffering from survivorship bias (which we have already discussed above), are also often composite price feeds from multiple exchanges. This means that the extreme points (i.e. the open, close, high and low) of the data are very susceptible to "outlying" values due to small orders at regional exchanges. Further, these values are also sometimes more likely to be tick-errors that have yet to be removed from the dataset.

This means that if your trading strategy makes extensive use of any of the OHLC points specifically, backtest performance can differ from live performance as orders might be routed to different exchanges depending upon your broker and your available access to liquidity. The only way to resolve these problems is to make use of higher frequency data or obtain data directly from an individual exchange itself, rather than a cheaper composite feed.

3.3.3 Forex Trading and ECNs

The backtesting of foreign exchange strategies is somewhat trickier to implement than that of equity strategies. Forex trading occurs across multiple venues and Electronic Communication Networks (ECN). The bid/ask prices achieved on one venue can differ substantially from those on another venue. One must be extremely careful to make use of pricing information from the particular venue you will be trading on in the backtest, as opposed to a consolidated feed from multiple venues, as this will be significantly more indicative of the prices you are likely to achieve going forward.

Another idiosyncrasy of the foreign exchange markets is that brokers themselves are not obligated to share *trade* prices/sizes with every trading participant, since this is their proprietary information[6]. Thus it is more appropriate to use bid-ask quotes in your backtests and to be extremely careful of the variation of transaction costs between brokers/venues.

3.3.4 Shorting Constraints

When carrying out short trades in the backtest it is necessary to be aware that some equities may not have been available (due to the lack of availability in that stock to borrow) or due to a market constraint, such as the US SEC banning the shorting of financial stocks during the 2008 market crisis.

This can severely inflate backtesting returns so be careful to include such short sale constraints within your backtests, or avoid shorting at all if you believe there are likely to be liquidity constraints in the instruments you trade.

3.4 Transaction Costs

One of the most **prevalent beginner mistakes** when implementing trading models is to neglect (or grossly underestimate) the effects of *transaction costs* on a strategy. Though it is often assumed that transaction costs only reflect broker commissions, there are in fact many other ways that costs can be accrued on a trading model. The three main types of costs that must be considered include:

3.4.1 Commission

The most direct form of transaction costs incurred by an algorithmic trading strategy are commissions and fees. All strategies require some form of access to an *exchange*, either directly or through a brokerage intermediary ("the broker"). These services incur an incremental cost with each trade, known as *commission*.

Brokers generally provide many services, although quantitative algorithms only really make use of the exchange infrastructure. Hence brokerage commissions are often small on per trade basis. Brokers also charge *fees*, which are costs incurred to clear and settle trades. Further to this are *taxes* imposed by regional or national governments. For instance, in the UK there is a *stamp duty* to pay on equities transactions. Since commissions, fees and taxes are generally fixed, they are relatively straightforward to implement in a backtest engine (see below).

3.4.2 Slippage

Slippage is the difference in price achieved between the time when a trading system decides to transact and the time when a transaction is actually carried out at an exchange. Slippage is a considerable component of transaction costs and can make the difference between a very profitable strategy and one that performs poorly. Slippage is a function of the underlying asset volatility, the latency between the trading system and the exchange and the type of strategy being carried out.

An instrument with higher volatility is more likely to be moving and so prices between signal and execution can differ substantially. Latency is defined as the time difference between signal generation and point of execution. Higher frequency strategies are more sensitive to latency

issues and improvements of milliseconds on this latency can make all the difference towards profitability. The type of strategy is also important. Momentum systems suffer more from slippage on average because they are trying to purchase instruments that are already moving in the forecast direction. The opposite is true for mean-reverting strategies as these strategies are moving in a direction opposing the trade.

3.4.3 Market Impact

Market impact is the cost incurred to traders due to the supply/demand dynamics of the exchange (and asset) through which they are trying to trade. A large order on a relatively illiquid asset is likely to *move the market* substantially as the trade will need to access a large component of the current supply. To counter this, large block trades are broken down into smaller "chunks" which are transacted periodically, as and when new liquidity arrives at the exchange. On the opposite end, for highly liquid instruments such as the S&P500 E-Mini index futures contract, low volume trades are unlikely to adjust the "current price" in any great amount.

More illiquid assets are characterised by a larger **spread**, which is the difference between the current bid and ask prices on the **limit order book**. This spread is an additional transaction cost associated with any trade. Spread is a very important component of the total transaction cost - as evidenced by the myriad of UK spread-betting firms whose advertising campaigns express the "tightness" of their spreads for heavily traded instruments.

3.5 Backtesting vs Reality

In summary there are a staggering array of factors that can be simulated in order to generate a realistic backtest. The dangers of overfitting, poor data cleansing, incorrect handling of transaction costs, market regime change and trading constraints often lead to a backtest performance that differs substantially from a live strategy deployment.

Thus one must be very aware that future performance is very unlikely to match historical performance directly. We will discuss these issues in further detail when we come to implement an event-driven backtesting engine near the end of the book.

Chapter 4

Automated Execution

Automated execution is the process of letting the strategy automatically generate execution signals that are the sent to the broker without any human intervention. This is the purest form of algorithmic trading strategy, as it minimises issues due to human intervention. It is the type of system that we will consider most often during this book.

4.1 Backtesting Platforms

The software landscape for strategy backtesting is vast. Solutions range from fully-integrated institutional grade sophisticated software through to programming languages such as C++, Python and R where nearly everything must be written from scratch (or suitable 'plugins' obtained). As quant traders we are interested in the balance of being able to "own" our trading technology stack versus the speed and reliability of our development methodology. Here are the key considerations for software choice:

- **Programming Skill** - The choice of environment will in a large part come down to your ability to program software. I would argue that being in control of the total stack will have a greater effect on your long term PnL than outsourcing as much as possible to vendor software. This is due to the downside risk of having external bugs or idiosyncrasies that you are unable to fix in vendor software, which would otherwise be easily remedied if you had more control over your "tech stack". You also want an environment that strikes the right balance between productivity, library availability and speed of execution. I make my own personal recommendation below.
- **Execution Capability/Broker Interaction** - Certain backtesting software, such as Tradestation, ties in directly with a brokerage. I am not a fan of this approach as reducing transaction costs are often a big component of getting a higher Sharpe ratio. If you're tied into a particular broker (and Tradestation "forces" you to do this), then you will have a harder time transitioning to new software (or a new broker) if the need arises. Interactive Brokers provide an API which is robust, albeit with a slightly obtuse interface.
- **Customisation** - An environment like MATLAB or Python gives you a great deal of flexibility when creating algo strategies as they provide fantastic libraries for nearly any mathematical operation imaginable, but also allow extensive customisation where necessary.
- **Strategy Complexity** - Certain software just isn't cut out for heavy number crunching or mathematical complexity. Excel is one such piece of software. While it is good for simpler strategies, it cannot really cope with numerous assets or more complicated algorithms, at speed.
- **Bias Minimisation** - Does a particular piece of software or data lend itself more to trading biases? You need to make sure that if you want to create all the functionality yourself,

that you don't introduce bugs which can lead to biases. An example here is look-ahead bias, which Excel minimises, while a vectorised research backtester might lend itself to accidentally.

- **Speed of Development** - One shouldn't have to spend months and months implementing a backtest engine. Prototyping should only take a few weeks. Make sure that your software is not hindering your progress to any great extent, just to grab a few extra percentage points of execution speed.

- **Speed of Execution** - If your strategy is completely dependent upon execution timeliness (as in HFT/UHFT) then a language such as C or C++ will be necessary. However, you will be verging on Linux kernel optimisation and FPGA usage for these domains, which is outside the scope of the book.

- **Cost** - Many of the software environments that you can program algorithmic trading strategies with are completely free and open source. In fact, many hedge funds make use of open source software for their entire algo trading stacks. In addition, Excel and MATLAB are both relatively cheap and there are even free alternatives to each.

Different strategies will require different software packages. HFT and UHFT strategies will be written in C/C++. These days such strategies are often carried out on GPUs and FPGAs. Conversely, low-frequency directional equity strategies are easy to implement in TradeStation, due to the "all in one" nature of the software/brokerage.

4.1.1 Programming

Custom development of a backtesting language within a first-class programming language provides the most flexibility when testing a strategy. Conversely, a vendor-developed integrated backtesting platform will always have to make assumptions about how backtests are carried out. The choice of available programming languages is large and diverse. It is not obvious before development which language would be suitable.

Once a strategy has been codified into systematic rules it is necessary to backtest it in such a manner that the quantitative trader is confident that its future performance will be reflective of its past performance. There are generally two forms of backtesting system that are utilised to test this hypothesis. Broadly, they are categorised as *research back testers* and *event-driven back testers*.

4.1.2 Research Tools

The simpler form of a backtesting tool, the research tool, is usually considered first. The research tool is used to quickly ascertain whether a strategy is likely to have any performance. Such tools often make unrealistic assumptions about transaction costs, likely fill prices, shorting constraints, venue dependence, risk management and a host of other issues that were outlined in the previous chapter. Common tools for research include MATLAB, R, Python and Excel.

The research stage is useful because the software packages provide significant *vectorised* capability, which leads to good execution speed and straightforward implementation (less lines of code). Thus it is possible to test multiple strategies, combinations and variants in a rapid, iterative manner.

While such tools are often used for both backtesting and execution, such research environments are generally not suitable for strategies that approach intraday trading at higher frequencies (sub-minute). This is because these environments do not often possess the necessary libraries to connect to real-time market data vendor servers or can interface with brokerage APIs in a clean fashion.

Despite these executional shortcomings, research environments are heavily used within the professional quantitative environment. They are the "firs test" for all strategy ideas before promoting them to a more rigourous check within a realistic backtesting environment.

4.1.3 Event-Driven Backtesting

Once a strategy has been deemed suitable on a research basis it must be tested in a more realistic fashion. Such realism attempts to account for the majority (if not all) of the issues described in the previous chapter. The ideal situation is to be able to use the same trade generation code for historical backtesting as well as live execution. This can be achieved using an *event-driven backtester*.

Event-driven systems are widely used in software engineering, commonly for handling graphical user interface (GUI) input within window-based operating systems. They are also ideal for algorithmic trading. Such systems are often written in high-performance languages such as C++, C# and Java.

Consider a situation where an automated trading strategy is connected to a real-time market feed and a broker (these two may be one and the same). New market information will be sent to the system, which triggers and *event* to generate a new trading signal and thus an execution *event*. Thus such a system is in a continuous loop waiting to receive events and handle them appropriately.

It is possible to generate sub-components such as a historic data handler and brokerage simulator, which can mimic their live counterparts. This allows backtesting strategies in a manner that is extremely similar to live execution.

The disadvantage of such systems is that they are far more complicated to design and implement than a simpler research tool. Thus the "time to market" is longer. They are more prone to bugs and require a reasonable knowledge of programming and, to a degree, software development methodology.

4.1.4 Latency

In engineering terms, *latency* is defined as the time interval between a simulation and a response. For our purposes it will generally refer to the round-trip time delay between the generation of an execution signal and the receipt of the fill information from a broker that is carrying out the execution.

Such latency is rarely an issue on low-frequency interday strategies since the likely price movement during the latency period will not affect the strategy to any great extent. Unfortunately, the same is not true of higher-frequency strategies. At these frequencies latency becomes important. The ultimate goal is to reduce latency as much as possible in order to minimise *slippage*, as discussed in the previous chapter.

Decreasing latency involves minimising the "distance" between the algorithmic trading system and the ultimate exchange on which an order is being executed. This can involve shortening the geographic distance between systems (and thus reducing travel down network cabling), reducing the processing carried out in networking hardware (important in HFT strategies) or choosing a brokerage with more sophisticated infrastructure.

Decreasing latency becomes exponentially more expensive as a function of "internet distance" (i.e. the network distance between two servers). Thus, for a high-frequency trader, a compromise must be reached between expenditure of latency-reduction vs the gain from minimising slippage. These issues will be discussed in the section on *Colocation* below.

4.1.5 Language Choices

Some issues that drive language choice have already been outlined. Now we will consider the benefits and drawbacks of individual programming languages. I have broadly categorised the languages into high-performance/harder development vs lower-performance/easier development. These are subjective terms and some will disagree depending upon their background.

One of the most important aspects of programming a custom backtesting environment is that the programmer is familiar with the tools being used. That is probably a more important criterion than speed of development. However, for those that are new to the programming language landscape, the following should clarify what tends to be utilised within algorithmic trading.

C++, C# and Java

C++, C# and Java are all examples of *general purpose* object-oriented programming languages. That is, they can be used without a corresponding IDE, are all cross-platform (can be run on Windows, Mac OSX or Linux), have a wide range of libraries for nearly any imaginable task and possess have rapid execution speed.

If ultimate execution speed is desired then C++ (or C) is likely to be the best choice. It offers the most flexibility for managing memory and optimising execution speed. This flexibility comes at a price. C++ is notoriously tricky to learn well and can often lead to subtle bugs. Development time can take much longer than in other languages.

C# and Java are similar in that they both require all components to be objects, with the exception of primitive data types such as floats and integers. They differ from C++ in that they both perform automatic *garbage collection*. This means that memory does not have to be manually de-allocated upon destruction of an object. Garbage collection adds a performance overhead but it makes development substantially more rapid. These languages are both good choices for developing a backtester as they have native GUI capabilities, numerical analysis libraries and fast execution speed.

Personally, I would make use of C++ for creating an event-driven backtester that needs extremely rapid execution speed, such as for HFT. This is only if I feel that a Python event-driven system was becoming a bottleneck, as the latter language would be my first choice for such a system.

MATLAB, R and Python

MATLAB is a commercial integrated development environment (IDE) for numerical computation. It has gained wide acceptance in the academic, engineering and financial sectors. It has a huge array of numerical libraries for many scientific computing tasks and possesses a rapid execution speed, assuming that any algorithm being developed is subject to *vectorisation* or *parallelisation*. It is expensive and this sometimes makes it less appealing to retail traders on a budget. MATLAB is sometimes used for direct execution through to a brokerage such as Interactive Brokers.

R is less of a general purpose programming language and more of a statistics scripting environment. It is free, open-source, cross-platform and contains a huge array of freely-available statistical packages for carrying out extremely advanced analysis. R is very widely used in the quantitative hedge fund industry for statistical/strategy research. While it is possible to connect R to a brokerage, is not well suited to the task and should be considered more of a research tool. In addition, it lacks execution speed unless operations are vectorised.

I've grouped Python under this heading, although it sits somewhere between MATLAB, R and the aforementioned general-purpose languages. It is free, open-source and cross-platform. It is *interpreted* as opposed to *compiled*, which makes it natively slower than C++. Despite this it contains a library for carrying out nearly any task imaginable, from scientific computing through to low-level web server design. In particular, it contains NumPy, SciPy, pandas, matplotlib and scikit-learn, which provide a robust numerical research environment that, when vectorised, is comparable to compiled language execution speed.

Further, it has mature libraries for connecting to brokerages. This makes it a "one-stop shop" for creating an event-driven backtesting and live execution environment without having to step into other languages. Execution speed is more than sufficient for intraday traders trading on the time scale of minutes. Finally, it is very straightforward to pick up and learn, when compared to lower-level languages like C++. For these reasons we make extensive use of Python within this book.

4.1.6 Integrated Development Environments

The term IDE has multiple meanings within algorithmic trading. Software developers use it to mean a GUI that allows programming with syntax highlighting, file browsing, debugging and code execution features. Algorithmic traders use it to mean a fully-integrated backtesting/trading environment, including historical or real-time data download, charting, statistical evaluation and

live execution. For our purposes, I use the term to mean any environment (often GUI-based) that is *not* a general purpose programming language, such as C++ or Python. MATLAB is considered an IDE, for instance.

Excel

While some purer quants may look down on Excel, I have found it to be extremely useful for "sanity checking" of results. The fact that all of the data is directly available, rather than hidden behind objects, makes it straightforward to implement very basic signal/filter strategies. Brokerages, such as Interactive Brokers, also allow DDE plugins that allow Excel to receive real-time market data and execute trading orders.

Despite the ease of use, Excel is extremely slow for any reasonable scale of data or level of numerical computation. I only use it to error-check when developing against other strategies and to make sure I've avoided look-ahead bias, which is easy to see in Excel due to the spreadsheet nature of the software.

If you are uncomfortable with programming languages and are carrying out an interday strategy, then Excel may be the perfect choice.

Commercial/Retail Backtesting Software

The market for retail charting, "technical analysis" and backtesting software is extremely competitive. Features offersd by such software include real-time charting of prices, a wealth of technical indicators, customised backtesting langauges and automated execution.

Some vendors provide an all-in-one solution, such as TradeStation. TradeStation are an online brokerage who produce trading software (also known as TradeStation) that provides electronic order execution across multiple asset classes. I don't currently believe that they offer a direct API for automated execution, rather orders must be placed through the software. This is in contrast to Interactive Brokers, who have a more stripped down charting/trading interface (Trader WorkStation), but offer both their proprietary real-time market/order execution APIs and a FIX interface.

Another extremely popular platform is MetaTrader, which is used in foreign exchange trading for creating 'Expert Advisors'. These are custom scripts written in a proprietary language that can be used for automated trading. I haven't had much experience with either TradeStation or MetaTrader so I won't spend too much time discussing their merits.

Such tools are useful if you are not comfortable with in depth software development and wish a lot of the details to be taken care of. However, with such systems a lot of flexibility is sacrificed and you are often tied to a single brokerage.

Web-Based Tools

The two current popular web-based backtesting systems are Quantopian (https://www.quantopian.com/) and QuantConnect (https://www.quantconnect.com/). The former makes use of Python (and ZipLine, see below) while the latter utilises C#. Both provide a wealth of historical data. Quantopian currently supports live trading with Interactive Brokers, while QuantConnect is working towards live trading.

Open Source Backtesting Software

In addition to the commercial offerings, there are open source alternatives for backtesting software.

Algo-Trader is a Swiss-based firm that offer both an open-source and a commercial license for their system. From what I can gather the offering seems quite mature and they have many institutional clients. The system allows full historical backtesting and *complex event processing* and they tie into Interactive Brokers. The Enterprise edition offers substantially more high performance features.

Marketcetera provide a backtesting system that can tie into many other languages, such as Python and R, in order to leverage code that you might have already written. The 'Strategy

Studio' provides the ability to write backtesting code as well as optimised execution algorithms and subsequently transition from a historical backtest to live paper trading.

ZipLine is the Python library that powers the Quantopian service mentioned above. It is a fully event-driven backtest environment and currently supports US equities on a minutely-bar basis. I haven't made extensive use of ZipLine, but I know others who feel it is a good tool. There are still many areas left to improve, but the team are constantly working on the project so it is very actively maintained.

There are also some Github/Google Code hosted projects that you may wish to look into. I have not spent any great deal of time investigating them. Such projects include OpenQuant (http://code.google.com/p/openquant/), TradeLink (https://code.google.com/p/tradelink/) and PyAlgoTrade (http://gbeced.github.io/pyalgotrade/).

Institutional Backtesting Software

Institutional-grade backtesting systems, such as Deltix and QuantHouse, are not often utilised by retail algorithmic traders. The software licenses are generally well outside the budget for infrastructure. That being said, such software is widely used by quant funds, proprietary trading houses, family offices and the like.

The benefits of such systems are clear. They provide an all-in-one solution for data collection, strategy development, historical backtesting and live execution across single instruments or portfolios, up to the ultra-high frequency level. Such platforms have had extensive testing and plenty of "in the field" usage and so are considered robust.

The systems are event-driven and as such the backtesting environment can often simulate the live environment well. The systems also support optimised execution algorithms, which attempt to minimise transaction costs.

I have to admit that I have not had much experience of Deltix or QuantHouse beyond some cursory overviews. That being said, the budget alone puts them out of reach of most retail traders, so I won't dwell on these systems.

4.2 Colocation

The software landscape for algorithmic trading has now been surveyed. It is now time to turn attention towards implementation of the hardware that will execute our strategies.

A retail trader will likely be executing their strategy from home during market hours, turning on their PC, connecting to the brokerage, updating their market software and then allowing the algorithm to execute automatically during the day. Conversely, a professional quant fund with significant *assets under management* (AUM) will have a dedicated exchange-colocated server infrastructure in order to reduce latency as far as possible to execute their high speed strategies.

4.2.1 Home Desktop

The simplest approach to hardware deployment is simply to carry out an algorithmic strategy with a home desktop computer connected to the brokerage via a broadband (or similar) connection.

While this approach is straightforward to get started it does suffers from many drawbacks. Primarily, the desktop machine is subject to power failure, unless backed up by a UPS. In addition, a home internet connection is also at the mercy of the ISP. Power loss or internet connectivity failure could occur at a crucial moment in trading, leaving the algorithmic trader with open positions that are unable to be closed.

Secondly, a desktop machine must occasionally be restarted, often due to the reliability of the operating system. This means that the strategy suffers from a degree of indirect manual intervention. If this occurs outside of trading hours the problem is mitigated. However, if a computer needs a restart during trading hours the problem is similar to a power loss. Unclosed positions may still be subject to risk.

Component failure also leads to the same set of "downtime" problems. A failure in the hard disk, monitor or motherboard often occurs at precisely the wrong time. For all of these reasons

I hesitate to recommend a home desktop approach to algorithmic trading. If you do decide to pursue this approach, make sure to have both a backup computer AND a backup internet connection (e.g. a 3G dongle) that you can use to close out positions under a downtime situation.

4.2.2 VPS

The next level up from a home desktop is to make use of a **virtual private server** (VPS). A VPS is a remote server system often marketed as a "cloud" service. They are far cheaper than a corresponding dedicated server, since a VPS is actually a partition of a much larger server, with a virtual isolated operating system environment solely available to you. CPU load is shared between multiple servers and a portion of the systems RAM is allocated to the VPS.

Common VPS providers include Amazon EC2 and Rackspace Cloud. They provide entry-level systems with low RAM and basic CPU usage, through to enterprise-ready high RAM, high CPU servers. For the majority of algorithmic retail traders, the entry level systems suffice for low-frequency intraday or interday strategies and smaller historical data databases.

The benefits of a VPS-based system include 24/7 availability (with a certain realistic down-time!), more robust monitoring capabilities, easy "plugins" for additional services, like file storage or managed databases and a flexible architecture. The drawbacks include expense as the system grows, since dedicated hardware becomes far cheaper per performance, assuming colocation away from an exchange, as well as handling failure scenarios (i.e. by creating a second identical VPS, for instance).

In addition, latency is not always improved by choosing a VPS/cloud provider. Your home location may be closer to a particular financial exchange than the data centres of your cloud provider. This is somewhat mitigated by choosing a firm that provide VPS geared specifically for algorithmic trading which are located at or near exchanges, however these will likely cost more than a "traditional" VPS provider such as Amazon or Rackspace.

4.2.3 Exchange

In order to get the best latency minimisation and fastest systems, it is necessary to colocate a dedicated server (or set of servers) directly at the exchange data centre. This is a prohibitively expensive option for nearly all retail algorithmic traders (unless they're very well capitalised). It is really the domain of the professional quantitative fund or brokerage.

As I mentioned above, a more realistic option is to purchase a VPS system from a provider that is located near an exchange. I won't dwell too heavily on exchange colocation, as the topic is somewhat outside the scope of the book.

Chapter 5

Sourcing Strategy Ideas

In this chapter I want to introduce you to the methods by which I myself identify profitable algorithmic trading strategies. We will discuss how to find, evaluate and select such systems. I'll explain how identifying strategies is as much about personal preference as it is about strategy performance, how to determine the type and quantity of historical data for testing, how to dispassionately evaluate a trading strategy and finally how to proceed towards the backtesting phase and strategy implementation.

5.1 Identifying Your Own Personal Preferences for Trading

In order to be a successful trader - either discretionally or algorithmically - it is necessary to ask yourself some honest questions. Trading provides you with the ability to lose money at an alarming rate, so it is necessary to "know thyself" as much as it is necessary to understand your chosen strategy.

I would say the most important consideration in trading is **being aware of your own personality**. Trading, and algorithmic trading in particular, requires a significant degree of discipline, patience and emotional detachment. Since you are letting an algorithm perform your trading for you, it is necessary to be resolved not to interfere with the strategy when it is being executed. This can be extremely difficult, especially in periods of extended drawdown. However, many strategies that have been shown to be highly profitable in a backtest can be ruined by simple interference. Understand that if you wish to enter the world of algorithmic trading you will be emotionally tested and that in order to be successful, it is necessary to work through these difficulties!

The next consideration is one of **time**. Do you have a full time job? Do you work part time? Do you work from home or have a long commute each day? These questions will help determine the *frequency* of the strategy that you should seek. For those of you in full time employment, an intraday futures strategy may not be appropriate (at least until it is fully automated!). Your time constraints will also dictate the methodology of the strategy. If your strategy is frequently traded and reliant on expensive news feeds (such as a Bloomberg terminal) you will clearly have to be realistic about your ability to successfully run this while at the office! For those of you with a lot of time, or the skills to automate your strategy, you may wish to look into a more technical high-frequency trading (HFT) strategy.

My belief is that it is necessary to carry out **continual research** into your trading strategies to maintain a consistently profitable portfolio. Few strategies stay "under the radar" forever. Hence a significant portion of the time allocated to trading will be in carrying out ongoing research. Ask yourself whether you are prepared to do this, as it can be the difference between strong profitability or a slow decline towards losses.

You also need to consider your **trading capital**. The generally accepted ideal minimum amount for a quantitative strategy is 50,000 USD (approximately £35,000 for us in the UK). If I was starting again, I would begin with a larger amount, probably nearer 100,000 USD (approximately £70,000). This is because transaction costs can be extremely expensive for mid- to high-frequency strategies and it is necessary to have sufficient capital to absorb them in times

of drawdown. If you are considering beginning with less than 10,000 USD then you will need to restrict yourself to low-frequency strategies, trading in one or two assets, as transaction costs will rapidly eat into your returns. *Interactive Brokers, which is one of the friendliest brokers to those with programming skills, due to its API, has a retail account minimum of 10,000 USD.*

Programming skill is an important factor in creating an automated algorithmic trading strategy. Being knowledgeable in a programming language such as C++, Java, C#, Python or R will enable you to create the end-to-end data storage, backtest engine and execution system yourself. This has a number of advantages, chief of which is the ability to be completely aware of all aspects of the trading infrastructure. It also allows you to explore the higher frequency strategies as you will be in full control of your "technology stack". While this means that you can test your own software and eliminate bugs, it also means more time spent coding up infrastructure and less on implementing strategies, at least in the earlier part of your algo trading career. You may find that you are comfortable trading in Excel or MATLAB and can outsource the development of other components. I would not recommend this however, particularly for those trading at high frequency.

You need to ask yourself **what you hope to achieve** by algorithmic trading. Are you interested in a regular income, whereby you hope to draw earnings from your trading account? Or, are you interested in a long-term capital gain and can afford to trade without the need to drawdown funds? Income dependence will dictate the frequency of your strategy. More regular income withdrawals will require a higher frequency trading strategy with less volatility (i.e. a higher Sharpe ratio). Long-term traders can afford a more sedate trading frequency.

Finally, do not be deluded by the notion of becoming extremely wealthy in a short space of time! **Algo trading is NOT a get-rich-quick scheme** - if anything it can be a become-poor-quick scheme. It takes significant discipline, research, diligence and patience to be successful at algorithmic trading. It can take months, if not years, to generate consistent profitability.

5.2 Sourcing Algorithmic Trading Ideas

Despite common perceptions to the contrary, it is actually quite straightforward to locate profitable trading strategies in the public domain. Never have trading ideas been more readily available than they are today. Academic finance journals, pre-print servers, trading blogs, trading forums, weekly trading magazines and specialist texts provide thousands of trading strategies with which to base your ideas upon.

Our goal as *quantitative trading researchers* is to establish a **strategy pipeline** that will provide us with a stream of ongoing trading ideas. Ideally we want to create a methodical approach to sourcing, evaluating and implementing strategies that we come across. The aims of the *pipeline* are to generate a consistent quantity of new ideas and to provide us with a framework for rejecting the majority of these ideas with the minimum of emotional consideration.

We must be extremely careful not to let cognitive biases influence our decision making methodology. This could be as simple as having a preference for one asset class over another (gold and other precious metals come to mind) because they are perceived as more exotic. Our goal should always be to find consistently profitable strategies, with positive expectation. The choice of asset class should be based on other considerations, such as trading capital constraints, brokerage fees and leverage capabilities.

5.2.1 Textbooks

If you are completely unfamiliar with the concept of a *trading strategy* and financial markets in general then the first place to look is with established textbooks. Classic texts provide a wide range of simpler, more straightforward ideas, with which to familiarise yourself with quantitative trading. Here is a selection that I recommend for those who are new to quantitative trading, which gradually become more sophisticated as you work through the list.

Financial Markets and Participants

The following list details books that outline how capital markets work and describe modern electronic trading.

- **Financial Times Guide to the Financial Markets** by Glen Arnold[1] - This book is designed for the novice to the financial markets and is extremely useful for gaining insight into all of the market participants. For our purposes, it provides us with a list of markets on which we might later form algorithmic trading strategies.
- **Trading and Exchanges: Market Microstructure for Practitioners** by Larry Harris[7] - This is an extremely informative book on the participants of financial markets and how they operate. It provides significant detail in how trades are carried out, the various motives of the players and how markets are regulated. While some may consider it "dry reading" I believe it is absolutely essential to understand these concepts to be a good algorithmic trader.
- **Algorithmic Trading and DMA: An introduction to direct access trading strategies** by Barry Johnson[10] - Johnson's book is geared more towards the technological side of markets. It discusses order types, optimal execution algorithms, the types of exchanges that accept algorithmic trading as well as more sophisticated strategies. As with Harris' book above, it explains in detail how electronic trading markets work, the knowledge of which I also believe is an essential prerequisite for carrying out systematic strategies.

Quantitative Trading

The next set of books are about algorithmic/quantitative trading directly. They outline some of the basic concepts and describe particular strategies that can be implemented.

- **Quantitative Trading: How to Build Your Own Algorithmic Trading Business** by Ernest Chan[5] - Ernest Chan's first book is a "beginner's guide" to quantitative trading strategies. While it is not heavy on strategy ideas, it does present a framework for how to setup a trading business, with risk management ideas and implementation tools. This is a great book if you are completely new to algorithmic trading. The book makes use of MATLAB.
- **Algorithmic Trading: Winning Strategies and Their Rationale** by Ernest Chan[6] - Chan's second book is very heavy on strategy ideas. It essentially begins where the previous book left off and is updated to reflect "current market conditions". The book discusses mean reversion and momentum based strategies at the interday and intraday frequencies. it also briefly touches on high-frequency trading. As with the prior book it makes extensive use of MATLAB code.
- **Inside The Black Box: The Simple Truth About Quantitative and High-Frequency Trading, 2nd Ed** by Rishi Narang[12] - Narang's book provides an overview of the components of a trading system employed by a quantitative hedge fund, including alpha generators, risk management, portfolio optimisation and transaction costs. The second edition goes into significant detail on high-frequency trading techniques.
- **Volatility Trading** by Euan Sinclair[16] - Sinclair's book concentrates solely on volatility modelling/forecasting and options strategies designed to take advantage of these models. If you plan to trade options in a quantitative fashion then this book will provide many research ideas.

5.2.2 The Internet

After gaining a grounding in the process of algorithmic trading via the classic texts, additional strategy ideas can be sourced from the internet. Quantitative finance blogs, link aggregators and trading forums all provide rich sources of ideas to test.

However, a note of caution: Many internet trading resources rely on the concept of **technical analysis**. Technical analysis involves utilising basic signals analysis *indicators* and *behavioural psychology* to determine trends or reversal patterns in asset prices.

Despite being extremely popular in the overall trading space, technical analysis is considered somewhat controversial in the quantitative finance community. Some have suggested that it is no better than reading a horoscope or studying tea leaves in terms of its predictive power! In reality there are successful individuals making extensive use of technical analysis in their trading.

As quants with a more sophisticated mathematical and statistical toolbox at our disposal, we can easily evaluate the effectiveness of such "TA-based" strategies. This allows us to make decisions driven by data analysis and hypothesis testing, rather than base such decisions on emotional considerations or preconceptions.

Quant Blogs

I recommend the following quant blogs for good trading ideas and concepts about algorithmic trading in general:

- **MATLAB Trading** - http://matlab-trading.blogspot.co.uk/
- **Quantitative Trading (Ernest Chan)** - http://epchan.blogspot.com
- **Quantivity** - http://quantivity.wordpress.com
- **Quantopian** - http://blog.quantopian.com
- **Quantpedia** - http://quantpedia.com

Aggregators

It has become fashionable in the last few years for topical links to be aggregated and then discussed. I read the following aggregators:

- **Quantocracy** - http://www.quantocracy.com
- **Quant News** - http://www.quantnews.com
- **Algo Trading Sub-Reddit** - http://www.reddit.com/r/algotrading

Forums

The next place to find additional strategy ideas is with trading forums. Do not be put off by more "technical analysis" oriented strategies. These strategies often provide good ideas that can be statistically tested:

- **Elite Trader Forums** - http://www.elitetrader.com
- **Nuclear Phynance** - http://www.nuclearphynance.com
- **QuantNet** - http://www.quantnet.com
- **Wealth Lab** - http://www.wealth-lab.com/Forum
- **Wilmott Forums** - http://www.wilmott.com

5.2.3 Journal Literature

Once you have had some experience at evaluating simpler strategies, it is time to look at the more sophisticated academic offerings. Some academic journals will be difficult to access, without high subscriptions or one-off costs. If you are a member or alumnus of a university, you should be able to obtain access to some of these financial journals. Otherwise, you can look at *pre-print servers*, which are internet repositories of late drafts of academic papers that are undergoing peer review. Since we are only interested in strategies that we can successfully replicate, backtest and obtain profitability for, a peer review is of less importance to us.

The major downside of academic strategies is that they can often either be out of date, require obscure and expensive historical data, trade in illiquid asset classes or do not factor in fees, slippage or spread. It can also be unclear whether the trading strategy is to be carried out with market orders, limit orders or whether it contains stop losses etc. Thus it is absolutely essential to replicate the strategy yourself as best you can, backtest it and add in realistic transaction costs that include as many aspects of the asset classes that you wish to trade in.

Here is a list of the more popular pre-print servers and financial journals that you can source ideas from:

- **arXiv** - http://arxiv.org/archive/q-fin
- **SSRN** - http://www.ssrn.com
- **Journal of Investment Strategies** - http://www.risk.net/type/journal/source/journal-of-investment-strategies
- **Journal of Computational Finance** - http://www.risk.net/type/journal/source/journal-of-computational-finance
- **Mathematical Finance** - http://onlinelibrary.wiley.com/journal/10.1111/%28ISSN%291467-9965

5.2.4 Independent Research

What about forming your own quantitative strategies? This generally requires (but is not limited to) expertise in one or more of the following categories:

- **Market microstructure** - For higher frequency strategies in particular, one can make use of *market microstructure*, i.e. understanding of the *order book dynamics* in order to generate profitability. Different markets will have various technology limitations, regulations, market participants and constraints that are all open to exploitation via specific strategies. This is a very sophisticated area and retail practitioners will find it hard to be competitive in this space, particularly as the competition includes large, well-capitalised quantitative hedge funds with strong technological capabilities.
- **Fund structure** - Pooled investment funds, such as pension funds, private investment partnerships (hedge funds), commodity trading advisors and mutual funds are constrained both by heavy regulation and their large capital reserves. Thus certain consistent behaviours can be exploited with those who are more nimble. For instance, large funds are subject to *capacity constraints* due to their size. Thus if they need to rapidly offload (sell) a quantity of securities, they will have to stagger it in order to avoid "moving the market". Sophisticated algorithms can take advantage of this, and other idiosyncrasies, in a general process known as *fund structure arbitrage*.
- **Machine learning/artificial intelligence** - Machine learning algorithms have become more prevalent in recent years in financial markets. Classifiers (such as Naive-Bayes, et al.) non-linear function matchers (neural networks) and optimisation routines (genetic algorithms) have all been used to predict asset paths or optimise trading strategies. If you have a background in this area you may have some insight into how particular algorithms might be applied to certain markets.

By continuing to monitor the above sources on a weekly, or even daily, basis you are setting yourself up to receive a consistent list of strategies from a diverse range of sources. The next step is to determine how to reject a large subset of these strategies in order to minimise wasting your time and backtesting resources on strategies that are likely to be unprofitable.

5.3 Evaluating Trading Strategies

The first, and arguably most obvious consideration is whether you actually **understand the strategy**. Would you be able to explain the strategy concisely or does it require a string of caveats and endless parameter lists? In addition, does the strategy have a good, solid basis in reality? For instance, could you point to some behavioural rationale or fund structure constraint that might be causing the pattern(s) you are attempting to exploit? Would this constraint hold up to a regime change, such as a dramatic regulatory environment disruption? Does the strategy rely on complex statistical or mathematical rules? Does it apply to any financial time series or is it specific to the asset class that it is claimed to be profitable on? You should constantly be thinking about these factors when evaluating new trading methods, otherwise you may waste a significant amount of time attempting to backtest and optimise unprofitable strategies.

Once you have determined that you understand the basic principles of the strategy you need to decide whether it fits with your aforementioned personality profile. This is not as vague a consideration as it sounds! Strategies will differ substantially in their performance characteristics. There are certain personality types that can handle more significant periods of drawdown, or are willing to accept greater risk for larger return. Despite the fact that we, as quants, try and eliminate as much cognitive bias as possible and should be able to evaluate a strategy dispassionately, biases will always creep in. Thus we need a consistent, unemotional means through which to assess the performance of strategies. Here is the list of criteria that I judge a potential new strategy by:

- **Methodology** - Is the strategy momentum based, mean-reverting, market-neutral, directional? Does the strategy rely on sophisticated (or complex!) statistical or machine learning techniques that are hard to understand and require a PhD in statistics to grasp? Do these techniques introduce a significant quantity of parameters, which might lead to optimisation bias? Is the strategy likely to withstand a *regime change* (i.e. potential new regulation of financial markets)?

- **Sharpe Ratio** - The Sharpe ratio heuristically characterises the reward/risk ratio of the strategy. It quantifies how much return you can achieve for the level of volatility endured by the equity curve. Naturally, we need to determine the period and frequency that these returns and volatility (i.e. standard deviation) are measured over. A higher frequency strategy will require greater sampling rate of standard deviation, but a shorter overall time period of measurement, for instance.

- **Leverage** - Does the strategy require significant leverage in order to be profitable? Does the strategy necessitate the use of leveraged derivatives contracts (futures, options, swaps) in order to make a return? These leveraged contracts can have heavy volatility characterises and thus can easily lead to *margin calls*. Do you have the trading capital and the temperament for such volatility?

- **Frequency** - The frequency of the strategy is intimately linked to your technology stack (and thus technological expertise), the Sharpe ratio and overall level of transaction costs. All other issues considered, higher frequency strategies require more capital, are more sophisticated and harder to implement. However, assuming your backtesting engine is sophisticated and bug-free, they will often have far higher Sharpe ratios.

- **Volatility** - Volatility is related strongly to the "risk" of the strategy. The Sharpe ratio characterises this. Higher volatility of the underlying asset classes, if unhedged, often leads to higher volatility in the equity curve and thus smaller Sharpe ratios. I am of course assuming that the positive volatility is approximately equal to the negative volatility. Some strategies may have greater downside volatility. You need to be aware of these attributes.

- **Win/Loss, Average Profit/Loss** - Strategies will differ in their win/loss and average profit/loss characteristics. One can have a very profitable strategy, even if the number of losing trades exceed the number of winning trades. Momentum strategies tend to have this pattern as they rely on a small number of "big hits" in order to be profitable. Mean-reversion strategies tend to have opposing profiles where more of the trades are "winners", but the losing trades can be quite severe.

- **Maximum Drawdown** - The maximum drawdown is the largest overall peak-to-trough percentage drop on the equity curve of the strategy. Momentum strategies are well known to suffer from periods of extended drawdowns (due to a string of many incremental losing trades). Many traders will give up in periods of extended drawdown, even if historical testing has suggested this is "business as usual" for the strategy. You will need to determine what percentage of drawdown (and over what time period) you can accept before you cease trading your strategy. This is a highly personal decision and thus must be considered carefully.

- **Capacity/Liquidity** - At the retail level, unless you are trading in a highly illiquid instrument (like a small-cap stock), you will not have to concern yourself greatly with strategy *capacity*. Capacity determines the scalability of the strategy to further capital. Many of the larger hedge funds suffer from significant capacity problems as their strategies increase in capital allocation.

- **Parameters** - Certain strategies (especially those found in the machine learning community) require a large quantity of parameters. Every extra parameter that a strategy requires leaves it more vulnerable to optimisation bias (also known as "curve-fitting"). You should try and target strategies with as few parameters as possible or make sure you have sufficient quantities of data with which to test your strategies on.

- **Benchmark** - Nearly all strategies (unless characterised as "absolute return") are measured against some performance benchmark. The benchmark is usually an *index* that characterises a large sample of the underlying asset class that the strategy trades in. If the strategy trades large-cap US equities, then the S&P500 would be a natural benchmark to measure your strategy against. You will hear the terms "alpha" and "beta", applied to strategies of this type.

Notice that we have not discussed the actual *returns* of the strategy. Why is this? In isolation, the returns actually provide us with limited information as to the effectiveness of the strategy. They don't give you an insight into leverage, volatility, benchmarks or capital requirements. Thus strategies are rarely judged on their returns alone. Always consider the risk attributes of a strategy before looking at the returns.

At this stage many of the strategies found from your pipeline will be rejected out of hand, since they won't meet your capital requirements, leverage constraints, maximum drawdown tolerance or volatility preferences. The strategies that do remain can now be considered for *backtesting*. However, before this is possible, it is necessary to consider one final rejection criteria - that of available historical data on which to test these strategies.

5.4 Obtaining Historical Data

Nowadays, the breadth of the technical requirements across asset classes for historical data storage is substantial. In order to remain competitive, both the buy-side (funds, prop-desks) and sell-side (broker/dealers) invest heavily in their technical infrastructure. It is imperative to consider its importance. In particular, we are interested in timeliness, accuracy and storage requirements. We will be discussing data storage in later chapters of the book.

In the previous section we had set up a strategy pipeline that allowed us to reject certain strategies based on our own personal rejection criteria. In this section we will filter more strategies based on our own preferences for obtaining historical data. The chief considerations (especially at retail practitioner level) are the costs of the data, the storage requirements and your level of

technical expertise. We also need to discuss the different types of available data and the different considerations that each type of data will impose on us.

Let's begin by discussing the types of data available and the key issues we will need to think about, with the understanding that we will explore these issues in significant depth in the remainder of the book:

- **Fundamental Data** - This includes data about macroeconomic trends, such as interest rates, inflation figures, corporate actions (dividends, stock-splits), SEC filings, corporate accounts, earnings figures, crop reports, meteorological data etc. This data is often used to value companies or other assets on a *fundamental* basis, i.e. via some means of expected future cash flows. It does not include stock price series. Some fundamental data is freely available from government websites. Other long-term historical fundamental data can be extremely expensive. Storage requirements are often not particularly large, unless thousands of companies are being studied at once.

- **News Data** - News data is often qualitative in nature. It consists of articles, blog posts, microblog posts ("tweets") and editorial. Machine learning techniques such as *classifiers* are often used to interpret *sentiment*. This data is also often freely available or cheap, via subscription to media outlets. The newer "NoSQL" document storage databases are designed to store this type of unstructured, qualitative data.

- **Asset Price Data** - This is the traditional data domain of the quant. It consists of time series of asset prices. Equities (stocks), fixed income products (bonds), commodities and foreign exchange prices all sit within this class. Daily historical data is often straightforward to obtain for the simpler asset classes, such as equities. However, once accuracy and cleanliness are included and statistical biases removed, the data can become expensive. In addition, time series data often possesses significant storage requirements especially when intraday data is considered.

- **Financial Instruments** - Equities, bonds, futures and the more exotic derivative options have very different characteristics and parameters. Thus there is no "one size fits all" database structure that can accommodate them. Significant care must be given to the design and implementation of database structures for various financial instruments.

- **Frequency** - The higher the frequency of the data, the greater the costs and storage requirements. For low-frequency strategies, daily data is often sufficient. For high frequency strategies, it might be necessary to obtain tick-level data and even historical copies of particular trading exchange *order book* data. Implementing a storage engine for this type of data is very technologically intensive and only suitable for those with a strong programming/technical background.

- **Benchmarks** - The strategies described above will often be compared to a *benchmark*. This usually manifests itself as an additional financial time series. For equities, this is often a national stock benchmark, such as the S&P500 index (US) or FTSE100 (UK). For a fixed income fund, it is useful to compare against a basket of bonds or fixed income products. The "risk-free rate" (i.e. appropriate interest rate) is also another widely accepted benchmark. All asset class categories possess a favoured benchmark, so it will be necessary to research this based on your particular strategy, if you wish to gain interest in your strategy externally.

- **Technology** - The technology stacks behind a financial data storage centre are complex. However, it does generally centre around a database cluster engine, such as a Relational Database Management System (RDBMS), such as MySQL, SQL Server, Oracle or a Document Storage Engine (i.e. "NoSQL"). This is accessed via "business logic" application code that queries the database and provides access to external tools, such as MATLAB, R or Excel. Often this business logic is written in C++, Java or Python. You will also need to host this data somewhere, either on your own personal computer, or remotely via internet servers. Products such as Amazon Web Services have made this simpler and cheaper in

recent years, but it will still require significant technical expertise to achieve in a robust manner.

As can be seen, once a strategy has been identified via the pipeline it will be necessary to evaluate the availability, costs, complexity and implementation details of a particular set of historical data. You may find it is necessary to reject a strategy based solely on historical data considerations. This is a big area and teams of PhDs work at large funds making sure pricing is accurate and timely. Do not underestimate the difficulties of creating a robust data centre for your backtesting purposes!

I do want to say, however, that many backtesting platforms can provide this data for you automatically - at a cost. Thus it will take much of the implementation pain away from you, and you can concentrate purely on strategy implementation and optimisation. Tools like TradeStation possess this capability. However, my personal view is to implement as much as possible internally and avoid outsourcing parts of the stack to software vendors. I prefer higher frequency strategies due to their more attractive Sharpe ratios, but they are often tightly coupled to the technology stack, where advanced optimisation is critical.

Part III

Data Platform Development

Chapter 6

Software Installation

This chapter will discuss in detail how to install an algorithmic trading environment. Operating system choice is considered as a necessary first step, with the three major choices outlined. Subsequently Linux is chosen as the system of choice (Ubuntu in particular) and Python is installed with all of the necessary libraries.

Package/library installation is often glossed over in additional books but I personally feel that it can be a stumbling block for many so I have devoted an entire chapter to it. Unfortunately the reality is that the chapter will become dated the moment it is released. Newer versions of operating systems emerge and packages are constantly updated. Hence there are likely to be specific implementation details.

If you do have trouble installing or working with these packages, make sure to check the versions installed and upgrade if necessary. If you still have trouble, feel free to email me at mike@quantstart.com and I'll try and help you out.

6.1 Operating System Choice

The first major choice when deciding on an algorithmic trading platform is that of the operating system. In some sense this will be dictated by the primary programming language or the means of connecting to the brokerage. These days the majority of software, particularly open source, is cross-platform and so the choice is less restricted.

6.1.1 Microsoft Windows

Windows is probably the "default" option of many algorithmic traders. It is extremely familiar and, despite criticism to the contrary, in certain forms is rather robust. Windows 8 has not been hugely well received but the prior version, Windows 7, is considered a solid operating system.

Certain tools in the algorithmic trading space will only function on Windows, in particular the IQFeed server, necessary to download tick data from DTN IQFeed. In addition Windows is the native platform of the Microsoft .NET framework, on which a vast quantity of financial software is written, utilising C++ and C#.

If you do not wish to use Windows then it is sometimes possible to run Windows-based software under a UNIX based system using the WINE emulator (http://www.winehq.org/).

6.1.2 Mac OSX

Mac OSX combines the graphical ease of Windows (some say it improves substantially upon it!) with the robustness of a UNIX based system (FreeBSD). While I use a MacBook Air for all of my "day to day" work, such as web/email and developing the QuantStart site, I have found it to be extremely painful to install a full algorithmic research stack, based on Python, under Mac OSX.

The package landscape of Mac OSX is significantly fragmented, with Homebrew and MacPorts being the primary contenders. Installation from source is tricky due to the proprietary

compilation process (using XCode). I have not yet successfully installed NumPy, SciPy and pandas on my MacBook as of this writing!

However, if you can navigate the minefield that is Python installation on Mac OSX, it can provide a great environment for algorithmic research. Since the Interactive Brokers Trader Workstation is Java-based, it has no trouble running on Mac OSX.

6.1.3 Linux

"Linux" refers to a set of free UNIX distributions such as Cent OS, Debian and Ubuntu. I don't wish to go into details about the benefits/drawbacks of each distribution, rather I will concentrate on Debian-based distro. In particular I will be considering Ubuntu Desktop as the algorithmic trading environment.

The aptitude package management makes it straightforward to install the necessary underlying libraries with ease. In addition it is straightforward to create a *virtual environment* for Python that can isolate your algo trading code from other Python apps. I have never had any (major) trouble installing a Python environment on a modern Ubuntu system and as such I have chosen this as the primary environment from which to conduct my trading.

If you would like to give Ubuntu a go before committing fully, by dual-booting for example, then it is possible to use VirtualBox (https://www.virtualbox.org/) to install it. I have a detailed guide on QuantStart (http://www.quantstart.com/articles/Installing-a-Desktop-Algorithmic-Trading-Research-Environment-using-Ubuntu-Linux-and-Python), which describes the process.

6.2 Installing a Python Environment on Ubuntu Linux

In this section we will discuss how to set up a robust, efficient and interactive development environment for algorithmic trading strategy research making use of Ubuntu Desktop Linux and the Python programming language. We will utilise this environment for all subsequent algorithmic trading implementations.

To create the research environment we will install the following software tools, all of which are open-source and free to download:

- **Ubuntu Desktop Linux** - The operating system
- **Python** - The core programming environment
- **NumPy/SciPy** - For fast, efficient vectorised array/matrix calculation
- **IPython** - For visual interactive development with Python
- **matplotlib** - For graphical visualisation of data
- **pandas** - For data "wrangling" and time series analysis
- **scikit-learn** - For machine learning and artificial intelligence algorithms
- **IbPy** - To carry out trading with the Interactive Brokers API

These tools coupled with a suitable MySQL securities master database will allow us to create a rapid interactive strategy research and backtesting environment. Pandas is designed for "data wrangling" and can import and cleanse time series data very efficiently. NumPy/SciPy running underneath keeps the system extremely well optimised. IPython/matplotlib (and the qtconsole described below) allow interactive visualisation of results and rapid iteration. scikit-learn allows us to apply machine learning techniques to our strategies to further enhance performance.

6.2.1 Python

The latest versions of Ubuntu, which at the time of writing is 13.10, still make use of the Python 2.7.x version family. While there is a transition underway to 3.3.x the majority of libraries are fully compatible with the 2.7.x branch. Thus I have chosen to use this for algorithmic trading. Things are likely to evolve rapidly though so in a couple of years 3.3.x may be the predominant branch. We will now commence with the installation of the Python environment.

The first thing to do on any brand new Ubuntu Linux system is to *update* and *upgrade* the packages. The former tells Ubuntu about new packages that are available, while the latter actually performs the process of replacing older packages with newer versions. Run the following commands in a terminal session and you will be prompted for your passwords:

```
sudo apt-get -y update
sudo apt-get -y upgrade
```

Note that the -y prefix tells Ubuntu that you want to accept 'yes' to all yes/no questions. "sudo" is a Ubuntu/Debian Linux command that allows other commands to be executed with administrator privileges. Since we are installing our packages sitewide, we need 'root access' to the machine and thus must make use of 'sudo'.

Once both of those updating commands have been successfully executed we need to install the Python development packages and compilers needed to compile all of the software. Notice that we are installing `build-essential` which contains the GCC compilers and the LAPACK linear algebra library, as well as **pip** which is the Python package management system:

```
sudo apt-get install python-pip python-dev python2.7-dev \
build-essential liblapack-dev libblas-dev
```

The next stage is to install the Python numerical and data analysis libraries.

6.2.2 NumPy, SciPy and Pandas

Once the necessary packages are installed above we can go ahead and install NumPy via pip, the Python package manager. Pip will download a zip file of the package and then compile it from the source code for us. Bear in mind that it will take some time to compile, possible 10 minutes or longer depending upon your CPU:

```
sudo pip install numpy
```

Once NumPy has been installed we need to check that it works before proceeding. If you look in the terminal you'll see your username followed by your computer name. In my case it is `mhallsmoore@algobox`, which is followed by the prompt. At the prompt type `python` and then try importing NumPy. We will test that it works by calculating the mean average of a list:

```
mhallsmoore@algobox:~$ python
Python 2.7.4 (default, Sep 26 2013, 03:20:26)
[GCC 4.7.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>> from numpy import mean
>>> mean([1,2,3])
2.0
>>> exit()
```

Now that NumPy has been successfully installed we want to install the Python Scientific library known as SciPy. It has a few package dependencies of its own including the ATLAS library and the GNU Fortran compiler, which must be installed first:

```
sudo apt-get install libatlas-base-dev gfortran
```

We are ready to install SciPy now, with pip. This will take quite a long time to compile, perhaps 10-20 minutes, depending upon CPU speed:

```
sudo pip install scipy
```

SciPy has now been installed. We will test it out in a similar fashion to NumPy when calculating the standard deviation of a list of integers:

```
mhallsmoore@algobox:~$ python
Python 2.7.4 (default, Sep 26 2013, 03:20:26)
[GCC 4.7.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import scipy
>>> from scipy import std
>>> std([1,2,3])
0.81649658092772603
>>> exit()
```

The final task for this section is to install the pandas data analysis library. We don't need any additional dependencies at this stage as they're covered by NumPy and SciPy:

```
sudo pip install pandas
```

We can now test the pandas installation, as before:

```
>>> from pandas import DataFrame
>>> pd = DataFrame()
>>> pd
Empty DataFrame
Columns: []
Index: []
>>> exit()
```

Now that the base numerical and scientific libraries have been installed we will install the statistical and machine learning libraries, statsmodels and scikit-learn.

6.2.3 Statsmodels and Scikit-Learn

Installation proceeds as before, making use of pip to install the packages:

```
sudo pip install statsmodels
sudo pip install scikit-learn
```

Both libraries can be tested:

```
mhallsmoore@algobox:~$ python
Python 2.7.4 (default, Sep 26 2013, 03:20:26)
[GCC 4.7.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> iris
..
..
'petal width (cm)']}
>>>
```

Now that the two statistical libraries are installed we can install the visualisation and development tools, IPython and matplotlib.

6.2.4 PyQt, IPython and Matplotlib

The first task is to install the dependency packages for matplotlib, the Python graphing library. Since matplotlib is a Python package, we cannot use pip to install the underlying libraries for working with PNGs, JPEGs and freetype fonts, so we need Ubuntu to install them for us:

```
sudo apt-get install libpng-dev libjpeg8-dev libfreetype6-dev
```

Now we can install matplotlib:

```
sudo pip install matplotlib
```

The last task of this section is to instal IPython. This is an interactive Python interpreter that provides a significantly more streamlined workflow compared to using the standard Python console. In later chapters we will emphasise the full usefulness of IPython for algorithmic trading development:

```
sudo pip install ipython
```

While IPython is sufficiently useful on its own, it can be made even more powerful by including the *qtconsole*, which provides the ability to inline matplotlib visualisations. However, it takes a little bit more work to get this up and running.

First, we need to install the the **Qt library**:

```
sudo apt-get install libqt4-core libqt4-gui libqt4-dev
```

The qtconsole has a few additional dependency packages, namely the ZMQ and Pygments libraries:

```
sudo apt-get install libzmq-dev
sudo pip install pyzmq
sudo pip install pygments
```

It is straightforward to test IPython by typing the following command:

```
ipython qtconsole --pylab=inline
```

To test IPython a simple plot can be generated by typing the following commands. Note that I've included the IPython numbered input/outut which you do not need to type:

```
In [1]: x=np.array([1,2,3])

In [2]: plot(x)
Out[2]: [<matplotlib.lines.Line2D at 0x392a1d0>]
```

This should display an inline matplotlib graph. Closing IPython allows us to continue with the installation.

6.2.5 IbPy and Trader Workstation

Interactive Brokers is one of the main brokerages used by retail algorithmic traders due to its relatively low minimal account balance requirements (10,000 USD) and (relatively) straightforward API. In this section we will install IbPy and Trader Workstation, which we will later use to carry out automated trade execution.

I want to emphasise that we are not going to be trading any live capital with this download! We are simply going to be installing some software which will let us try out a "demo account", which provides a market simulator with out of date data in a "real time" fashion.

Disclosure: I have no affiliation with Interactive Brokers. I have used them before in a professional fund context and as such am familiar with their software.

IbPy is a Python wrapper written around the Java-based Interactive Brokers API. It makes development of algorithmic trading systems in Python somewhat less problematic. It will be used as the basis for all subsequent communication with Interactive Brokers. An alternative is to use the FIX protocol, but we won't consider that method in this book.

Since IBPy is maintained on the GitHub source code version control website, as a git repository, we will need to install git. This is handled by:

```
sudo apt-get install git-core
```

Once git has been installed it is necessary to create a subdirectory to store IBPy. It can simply be placed underneath the home directory:

```
mkdir ~/ibapi
```

The next step is to download IBPy via the 'git clone' command:

```
cd ~/ibapi
git clone https://github.com/blampe/IbPy
```

The final step is to enter the IbPy directory and install using Python setuptools:

```
cd ~/ibapi/IbPy
python setup.py.in install
```

That completes the installation of IBPy. The next step is to install Trader Workstation. At the time of writing, it was necessary to follow this link (IB), which takes you directly to the Trader Workstation download page at Interactive Brokers. Select the platform that you wish to utilise. In this instance I have chosen the UNIX download, which can be found here (IB Unix Download).

At that link it will describe the remainder of the process but I will replicate it here for completeness. The downloaded file will be called *unixmacosx_latest.jar*. Open the file:

```
jar xf unixmacosx_latest.jar
```

Then change to the IBJts directory and load TWS:

```
cd IBJts
java -cp jts.jar:total.2013.jar -Xmx512M -XX:MaxPermSize=128M jclient.LoginFrame .
```

This will present you with the Trader Workstation login screen. If you choose the username "edemo" and the password "demo user" you will be logged into the system.

This completes the installation of a full algorithmic trading environment under Python and Ubuntu. The next stage is to begin collecting and storing historical pricing data for our strategies.

Chapter 7

Financial Data Storage

In algorithmic trading the spotlight usually shines on the *alpha model* component of the full trading system. This component generates the trading signals, prior to filtration by a risk management and portfolio construction system. As such, algo traders often spend a significant portion of their research time refining the alpha model in order to optimise one or more metrics prior to deployment of the strategy into production.

However, an alpha model is only as good as the data being fed into it. This concept is nicely characterised by the old computer science adage of *"garbage in, garbage out."* It is absolutely crucial that accurate, timely data is used to feed the alpha model. Otherwise results will be at best poor or at worst completely incorrect. This will lead to significant underperformance when system is deployed live.

In this chapter we will discuss issues surrounding the acquisition and provision of timely accurate data for an algorithmic strategy backtesting system and ultimately a trading execution engine. In particular we will study how to obtain financial data and how to store it. Subsequent chapters will discuss how to clean it and how to export it. In the financial industry this type of data service is known as a **securities master database**.

7.1 Securities Master Databases

A securities master is an organisation-wide database that stores **fundamental**, **pricing** and **transactional** data for a variety of financial instruments across asset classes. It provides access to this information in a consistent manner to be used by other departments such as risk management, clearing/settlement and proprietary trading.

In large organisations a range of instruments and data will be stored. Here are some of the instruments that might be of interest to a firm:

- Equities
- Equity Options
- Indices
- Foreign Exchange
- Interest Rates
- Futures
- Commodities
- Bonds - Government and Corporate
- Derivatives - Caps, Floors, Swaps

Securities master databases often have teams of developers and data specialists ensuring *high availability* within a financial institution. While this is necessary in large companies, at the retail level or in a small fund a securities master can be far simpler. In fact, while large securities masters make use of expensive enterprise database and analysis systems, it is possibly to use commodity open-source software to provide the same level of functionality, assuming a well-optimised system.

7.2 Financial Datasets

For the retail algorithmic trader or small quantitative fund the most common data sets are end-of-day and intraday historical pricing for equities, indices, futures (mainly commodities or fixed income) and foreign exchange (forex). In order to simplify this discussion we will concentrate solely on end-of-day (EOD) data for equities, ETFs and equity indices. Later sections will discuss adding higher frequency data, additional asset classes and derivatives data, which have more advanced requirements.

EOD data for equities is easy to obtain. There are a number of services providing access for free via web-available APIs:

- **Yahoo Finance** - http://finance.yahoo.com
- **Google Finance** - https://www.google.com/finance
- **QuantQuote** - https://www.quantquote.com (S&P500 EOD data only)
- **EODData** - http://eoddata.com (requires registration)

It is straightforward to manually download historical data for individual securities but it becomes time-consuming if many stocks need to be downloaded daily. Thus an important component of our securities master will be automatically updating the data set.

Another issue is *look-back period*. How far in the past do we need to go with our data? This will be specific to the requirements of your trading strategy, but there are certain problems that span all strategies. The most common is **regime change**, which is often characterised by a new regulatory environment, periods of higher/lower volatility or longer-term trending markets. For instance a long-term short-directional trend-following/momentum strategy would likely perform very well from 2000-2003 or 2007-2009. However it would have had a tough time from 2003-2007 or 2009 to the present.

My rule of thumb is to obtain as much data as possible, especially for EOD data where storage is cheap. Just because the data exists in your security master, does not mean it must be utilised. There are caveats around performance, as larger database tables mean longer query times (see below), but the benefits of having more sample points generally outweighs any performance issues.

As with all financial data it is imperative to be aware of errors, such as incorrect high/low prices or **survivorship bias**, which I have discussed at length in previous chapters.

7.3 Storage Formats

There are three main ways to store financial data. They all possess varying degrees of access, performance and structural capabilities. We will consider each in turn.

7.3.1 Flat-File Storage

The simplest data store for financial data, and the way in which you are likely to receive the data from any data vendors, is the flat-file format. Flat-files often make use of the Comma-Separated Variable (CSV) format, which store a two-dimensional matrix of data as a series of rows, with column data separated via a delimiter (often a comma, but can be whitespace, such as a space or tab). For EOD pricing data, each row represents a trading day via the OHLC paradigm (i.e. the prices at the open, high, low and close of the trading period).

The advantage of flat-files are their simplicity and ability to be heavily compressed for archiving or download. The main disadvantages lie in their lack of query capability and poor performance for iteration across large datasets. **SQLite** and **Excel** mitigate some of these problems by providing certain querying capabilities.

7.3.2 Document Stores/NoSQL

Document stores/NoSQL databases, while certainly not a new concept, have gained significant prominence in recent years due to their use at "web-scale" firms such as Google, Facebook and Twitter. They differ substantially from RDBMS systems in that there is no concept of table schemas. Instead, there are *collections* and *documents*, which are the closest analogies to tables and records, respectively. A wide taxonomy of document stores exist, the discussion of which is well outside this chapter! However, some of the more popular stores include **MongoDB**, **Cassandra** and **CouchDB**.

Document stores, in financial applications, are mostly suited to fundamental or meta data. Fundamental data for financial assets comes in many forms, such as corporate actions, earnings statements, SEC filings etc. Thus the schema-less nature of NoSQL DBs is well-suited. However, NoSQL DBs are not well designed for time-series such as high-resolution pricing data and so we won't be considering them further in this chapter.

7.3.3 Relational Database Management Systems

A *relational database management system* (RDBMS) makes use of the *relational model* to store data. These databases are particular well-suited to financial data because different "objects" (such as exchanges, data sources, prices) can be separated into tables with relationships defined between them.

RDBMS make use of **Structured Query Language** (SQL) in order to perform complex data queries on financial data. Examples of RDBMS include **Oracle**, **MySQL**, **SQLServer** and **PostgreSQL**.

The main advantages of RDBMS are their simplicity of installation, platform-independence, ease of querying, ease of integration with major backtest software and high-performance capabilities at large scale (although some would argue the latter is not the case!). Their disadvantages are often due to the complexity of customisation and difficulties of achieving said performance without underlying knowledge of how RDBMS data is stored. In addition, they possess semi-rigid schemas and so data often has to be modified to fit into such designs. This is unlike NoSQL data stores, where there is no schema.

For all of the future historical pricing implementation code in the book we will make use of the MySQL RDBMS. It is free, open-source, cross-platform, highly robust and its behaviour at scale is well-documented, which makes it a sensible choice for quant work.

7.4 Historical Data Structure

There is a significant body of theory and academic research carried out in the realm of computer science for the optimal design for data stores. However, we won't be going into too much detail as it is easy to get lost in minutiae! Instead I will present a common pattern for the construction of an equities security master, which you can modify as you see fit for your own applications.

The first task is to define our *entities*, which are elements of the financial data that will eventually map to tables in the database. For an equities master database I foresee the following entities:

- **Exchanges** - What is the ultimate original source of the data?
- **Vendor** - Where is a particular data point obtained from?
- **Instrument/Ticker** - The ticker/symbol for the equity or index, along with corporate information of the underlying firm or fund.

- **Price** - The actual price for a particular security on a particular day.

- **Corporate Actions** - The list of all stock splits or dividend adjustments (this may lead to one or more tables), necessary for adjusting the pricing data.

- **National Holidays** - To avoid mis-classifying trading holidays as missing data errors, it can be useful to store national holidays and cross-reference.

There are significant issues with regards to storing canonical tickers. I can attest to this from first hand experience at a hedge fund dealing with this exact problem! Different vendors use different methods for resolving tickers and thus combining multiple sources for accuracy. Further, companies become bankrupt, are exposed to M&A activity (i.e. become acquired and change names/symbols) and can have multiple publicly traded share classes. Many of you will not have to worry about this because your universe of tickers will be limited to the larger index constituents (such as the S&P500 or FTSE350).

7.5 Data Accuracy Evaluation

Historical pricing data from vendors is prone to many forms of error:

- **Corporate Actions** - Incorrect handling of stock splits and dividend adjustments. One must be absolutely sure that the formulae have been implemented correctly.

- **Spikes** - Pricing points that greatly exceed certain historical volatility levels. One must be careful here as these spikes do occur - see the May Flash Crash for a scary example. Spikes can also be caused by not taking into account stock splits when they do occur. *Spike filter* scripts are used to notify traders of such situations.

- **OHLC Aggregation** - Free OHLC data, such as from Yahoo/Google is particular prone to 'bad tick aggregation' situations where smaller exchanges process small trades well above the 'main' exchange prices for the day, thus leading to over-inflated maxima/minima once aggregated. This is less an 'error' as such, but more of an issue to be wary of.

- **Missing Data** - Missing data can be caused by lack of trades in a particular time period (common in second/minute resolution data of illiquid small-caps), by trading holidays or simply an error in the exchange system. Missing data can be *padded* (i.e. filled with the previous value), *interpolated* (linearly or otherwise) or ignored, depending upon the trading system.

Many of these errors rely on manual judgement in order to decide how to proceed. It is possible to automate the notification of such errors, but it is much harder to automate their solution. For instance, one must choose the threshold for being told about spikes - how many standard deviations to use and over what look-back period? Too high a stdev will miss some spikes, but too low and many unusual news announcements will lead to false positives. All of these issues require advanced judgement from the quant trader.

It is also necessary to decide how to fix errors. Should errors be corrected as soon as they're known, and if so, should an audit trail be carried out? This will require an extra table in the DB. This brings us to the topic of back-filling, which is a particularly insidious issue for backtesting. It concerns automatic correction of bad data upstream. If your data vendor corrects a historical error, but a backtested trading strategy is in production based on research from their previous bad data then decisions need to be made regarding the strategy effectiveness. This can be somewhat mitigated by being fully aware of your strategy performance metrics (in particular the variance in your win/loss characteristics for each trade). Strategies should be chosen or designed such that a single data point cannot skew the performance of the strategy to any great extent.

7.6 Automation

The benefit of writing software scripts to carry out the download, storage and cleaning of the data is that scripts can be automated via tools provided by the operating system. In UNIX-based systems (such as Mac OSX or Linux), one can make use of the **crontab**, which is a continually running process that allows specific scripts to be executed at custom-defined times or regular periods. There is an equivalent process on MS Windows known as the Task Scheduler.

A production process, for instance, might automate the download all of the S&P500 end-of-day prices as soon as they're published via a data vendor. It will then automatically run the aforementioned missing data and spike filtration scripts, alerting the trader via email, SMS or some other form of notification. At this point any backtesting tools will automatically have access to recent data, without the trader having to lift a finger! Depending upon whether your trading system is located on a desktop or on a remote server you may choose however to have a semi-automated or fully-automated process for these tasks.

7.7 Data Availability

Once the data is automatically updated and residing in the RDBMS it is necessary to get it into the backtesting software. This process will be highly dependent upon how your database is installed and whether your trading system is local (i.e. on a desktop computer) or remote (such as with a co-located exchange server).

One of the most important considerations is to minimise excessive Input/Output (I/O) as this can be extremely expensive both in terms of time and money, assuming remote connections where bandwidth is costly. The best way to approach this problem is to only move data across a network connection that you need (via selective querying) or exporting and compressing the data.

Many RDBMS support *replication* technology, which allows a database to be cloned onto another remote system, usually with a degree of latency. Depending upon your setup and data quantity this may only be on the order of minutes or seconds. A simple approach is to replicate a remote database onto a local desktop. However, be warned that synchronisation issues are common and time consuming to fix!

7.8 MySQL for Securities Masters

Now that we have discussed the idea behind a security master database it's time to actually build one. For this we will make use of two open source technologies: the **MySQL** database and the **Python** programming language. At the end of this chapter you will have a fully fledged equities security master with which to conduct further data analysis for your quantitative trading research.

7.8.1 Installing MySQL

Installation of MySQL within Ubuntu is straightforward. Simply open up a terminal and type the following:

```
sudo apt-get install mysql-server
```

Eventually, you will be prompted for a root password. This is your primary administration password so do not forget it! Enter the password and the installation will proceed and finish.

7.8.2 Configuring MySQL

Now that MySQL is installed on your system we can create a new *database* and a *user* to interact with it. You will have been prompted for a root password on installation. To log on to MySQL from the command line use the following line and then enter your password:

```
mysql -u root -p
```

Once you have logged in to the MySQL you can create a new database called `securities_master` and then select it:

```
mysql> CREATE DATABASE securities_master;
mysql> USE securities_master;
```

Once you create a database it is necessary to add a new *user* to interact with the database. While you can use the `root` user, it is considered bad practice from a security point of view, as it grants too many permissions and can lead to a compromised system. On a local machine this is mostly irrelevant but in a remote production environment you will certainly need to create a user with reduced permissions. In this instance our user will be called `sec_user`. Remember to replace `password` with a secure password:

```
mysql> CREATE USER 'sec_user'@'localhost' IDENTIFIED BY 'password';
mysql> GRANT ALL PRIVILEGES ON securities_master.* TO 'sec_user'@'localhost';
mysql> FLUSH PRIVILEGES;
```

The above three lines create and authorise the user to use `securities_master` and apply those privileges. From now on any interaction that occurs with the database will make use of the `sec_user` user.

7.8.3 Schema Design for EOD Equities

We've now installed MySQL and have configured a user with which to interact with our database. At this stage we are ready to construct the necessary tables to hold our financial data. For a simple, straightforward equities master we will create four tables:

- **Exchange** - The exchange table lists the exchanges we wish to obtain equities pricing information from. In this instance it will almost exclusively be the New York Stock Exchange (NYSE) and the National Association of Securities Dealers Automated Quotations (NASDAQ).
- **DataVendor** - This table lists information about historical pricing data vendors. We will be using Yahoo Finance to source our end-of-day (EOD) data. By introducing this table, we make it straightforward to add more vendors if necessary, such as Google Finance.
- **Symbol** - The symbol table stores the list of ticker symbols and company information. Right now we will be avoiding issues such as differing share classes and multiple symbol names.
- **DailyPrice** - This table stores the daily pricing information for each security. It can become very large if many securities are added. Hence it is necessary to optimise it for performance.

MySQL is an extremely flexible database in that it allows you to customise how the data is stored in an underlying *storage engine*. The two primary contenders in MySQL are **MyISAM** and **InnoDB**. Although I won't go into the details of storage engines (of which there are many!), I will say that MyISAM is more useful for fast reading (such as querying across large amounts of price information), but it doesn't support transactions (necessary to fully *rollback* a multi-step operation that fails mid way through). InnoDB, while transaction safe, is slower for reads.

InnoDB also allows row-level locking when making writes, while MyISAM locks the entire table when writing to it. This can have performance issues when writing a lot of information to arbitrary points in the table (such as with UPDATE statements). This is a deep topic, so I will leave the discussion to another day!

We are going to use InnoDB as it is natively transaction safe and provides row-level locking. If we find that a table is slow to be read, we can create *indexes* as a first step and then change the underlying storage engine if performance is still an issue. All of our tables will use the UTF-8 character set, as we wish to support international exchanges.

Let's begin with the schema and `CREATE TABLE` SQL code for the `exchange` table. It stores the abbreviation and name of the exchange (i.e. NYSE - New York Stock Exchange) as well as

the geographic location. It also supports a currency and a timezone offset from UTC. We also store a created and last updated date for our own internal purposes. Finally, we set the primary index key to be an auto-incrementing integer ID (which is sufficient to handle 2^{32} records):

```
CREATE TABLE ‘exchange‘ (
  ‘id‘ int NOT NULL AUTO_INCREMENT,
  ‘abbrev‘ varchar(32) NOT NULL,
  ‘name‘ varchar(255) NOT NULL,
  ‘city‘ varchar(255) NULL,
  ‘country‘ varchar(255) NULL,
  ‘currency‘ varchar(64) NULL,
  ‘timezone_offset‘ time NULL,
  ‘created_date‘ datetime NOT NULL,
  ‘last_updated_date‘ datetime NOT NULL,
  PRIMARY KEY (‘id‘)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

Here is the schema and `CREATE TABLE` SQL code for the `data_vendor` table. It stores the name, website and support email. In time we can add more useful information for the vendor, such as an API endpoint URL:

```
CREATE TABLE ‘data_vendor‘ (
  ‘id‘ int NOT NULL AUTO_INCREMENT,
  ‘name‘ varchar(64) NOT NULL,
  ‘website_url‘ varchar(255) NULL,
  ‘support_email‘ varchar(255) NULL,
  ‘created_date‘ datetime NOT NULL,
  ‘last_updated_date‘ datetime NOT NULL,
  PRIMARY KEY (‘id‘)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

Here is the schema and `CREATE TABLE` SQL code for the `symbol` table. It contains a foreign key link to an exchange (we will only be supporting exchange-traded instruments for the time being), a ticker symbol (e.g. GOOG), an instrument type ('stock' or 'index'), the name of the stock or stock market index, an equities sector and a currency.

```
CREATE TABLE ‘symbol‘ (
  ‘id‘ int NOT NULL AUTO_INCREMENT,
  ‘exchange_id‘ int NULL,
  ‘ticker‘ varchar(32) NOT NULL,
  ‘instrument‘ varchar(64) NOT NULL,
  ‘name‘ varchar(255) NULL,
  ‘sector‘ varchar(255) NULL,
  ‘currency‘ varchar(32) NULL,
  ‘created_date‘ datetime NOT NULL,
  ‘last_updated_date‘ datetime NOT NULL,
  PRIMARY KEY (‘id‘),
  KEY ‘index_exchange_id‘ (‘exchange_id‘)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

Here is the schema and `CREATE TABLE` SQL code for the `daily_price` table. This table is where the historical pricing data is actually stored. We have prefixed the table name with `daily_` as we may wish to create minute or second resolution data in separate tables at a later date for higher frequency strategies. The table contains two foreign keys - one to the data vendor and another to a symbol. This uniquely identifies the data point and allows us to store the same price data for multiple vendors in the same table. We also store a price date (i.e. the daily period over which the OHLC data is valid) and the created and last updated dates for our own purposes.

The remaining fields store the open-high-low-close and adjusted close prices. Yahoo Finance provides dividend and stock splits for us, the price of which ends up in the `adj_close_price`

column. Notice that the datatype is `decimal(19,4)`. When dealing with financial data it is absolutely necessary to be precise. If we had used the `float` datatype we would end up with rounding errors due to the nature of how `float` data is stored internally. The final field stores the trading volume for the day. This uses the `bigint` datatype so that we don't accidentally truncate extremely high volume days.

```
CREATE TABLE `daily_price` (
  `id` int NOT NULL AUTO_INCREMENT,
  `data_vendor_id` int NOT NULL,
  `symbol_id` int NOT NULL,
  `price_date` datetime NOT NULL,
  `created_date` datetime NOT NULL,
  `last_updated_date` datetime NOT NULL,
  `open_price` decimal(19,4) NULL,
  `high_price` decimal(19,4) NULL,
  `low_price` decimal(19,4) NULL,
  `close_price` decimal(19,4) NULL,
  `adj_close_price` decimal(19,4) NULL,
  `volume` bigint NULL,
  PRIMARY KEY (`id`),
  KEY `index_data_vendor_id` (`data_vendor_id`),
  KEY `index_symbol_id` (`symbol_id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

By entering all of the above SQL commands into the MySQL command line the four necessary tables will be created.

7.8.4 Connecting to the Database

Before we can use MySQL with Python we need to install the `mysqlclient` library. `mysqlclient` is actually a fork of another library, known as Python-MySQL. Unfortunately the latter library is not supported in Python3 and so we must use `mysqlclient`. On Mac OSX/UNIX flavour machines we need to run the following commands:

```
sudo apt-get install libmysqlclient-dev
pip install mysqlclient
```

We're now ready to begin interacting with our MySQL database via Python and pandas.

7.8.5 Using an Object-Relational Mapper

For those of you with a background in database administration and development you might be asking whether it is more sensible to make use of an **Object-Relational Mapper** (ORM). An ORM allows objects within a programming language to be directly mapped to tables in databases such that the program code is fully unaware of the underlying storage engine. They are not without their problems, but they can save a great deal of time. The time-saving usually comes at the expense of performance, however.

A popular ORM for Python is **SQLAlchemy**. It allows you to specify the database schema within Python itself and thus automatically generate the `CREATE TABLE` code. Since we have specifically chosen MySQL and are concerned with performance, I've opted not to use an ORM for this chapter.

Symbol Retrieval

Let's begin by obtaining all of the ticker symbols associated with the Standard & Poor's list of 500 large-cap stocks, i.e. the S&P500. Of course, this is simply an example. If you are trading from the UK and wish to use UK domestic indices, you could equally well obtain the list of FTSE100 companies traded on the London Stock Exchange (LSE).

Wikipedia conveniently lists the constituents of the S&P500. Note that there are actually 502 components in the S&P500! We will *scrape* the website using the Python **requests** and **BeautifulSoup** libraries and then add the content directly to MySQL. Firstly make sure the libraries are installed:

```
pip install requests
pip install beautifulsoup4
```

The following code will use the requests and BeautifulSoup libraries to add the symbols directly to the MySQL database we created earlier. Remember to replace 'password' with your chosen password as created above:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# insert_symbols.py

from __future__ import print_function

import datetime
from math import ceil

import bs4
import MySQLdb as mdb
import requests


def obtain_parse_wiki_snp500():
    """
    Download and parse the Wikipedia list of S&P500
    constituents using requests and BeautifulSoup.

    Returns a list of tuples for to add to MySQL.
    """
    # Stores the current time, for the created_at record
    now = datetime.datetime.utcnow()

    # Use requests and BeautifulSoup to download the
    # list of S&P500 companies and obtain the symbol table
    response = requests.get(
        "http://en.wikipedia.org/wiki/List_of_S%26P_500_companies"
    )
    soup = bs4.BeautifulSoup(response.text)

    # This selects the first table, using CSS Selector syntax
    # and then ignores the header row ([1:])
    symbolslist = soup.select('table')[0].select('tr')[1:]

    # Obtain the symbol information for each
    # row in the S&P500 constituent table
    symbols = []
    for i, symbol in enumerate(symbolslist):
        tds = symbol.select('td')
        symbols.append(
            (
                tds[0].select('a')[0].text,  # Ticker
                'stock',
                tds[1].select('a')[0].text,  # Name
```

```
                tds[3].text,  # Sector
                'USD', now, now
            )
        )
    return symbols


def insert_snp500_symbols(symbols):
    """
    Insert the S&P500 symbols into the MySQL database.
    """
    # Connect to the MySQL instance
    db_host = 'localhost'
    db_user = 'sec_user'
    db_pass = 'password'
    db_name = 'securities_master'
    con = mdb.connect(
        host=db_host, user=db_user, passwd=db_pass, db=db_name
    )

    # Create the insert strings
    column_str = """ticker, instrument, name, sector,
                currency, created_date, last_updated_date
                """
    insert_str = ("%s, " * 7)[:-2]
    final_str = "INSERT INTO symbol (%s) VALUES (%s)" % \
        (column_str, insert_str)

    # Using the MySQL connection, carry out
    # an INSERT INTO for every symbol
    with con:
        cur = con.cursor()
        cur.executemany(final_str, symbols)


if __name__ == "__main__":
    symbols = obtain_parse_wiki_snp500()
    insert_snp500_symbols(symbols)
    print("%s symbols were successfully added." % len(symbols))
```

At this stage we'll have all 502 current symbol constituents of the S&P500 index in the database. Our next task is to actually obtain the historical pricing data from separate sources and match it up the symbols.

Price Retrieval

In order to obtain the historical data for the current S&P500 constituents, we must first query the database for the list of all the symbols.

Once the list of symbols, along with the symbol IDs, have been returned it is possible to call the Yahoo Finance API and download the historical pricing data for each symbol.

Once we have each symbol we can insert the data into the database in turn. Here's the Python code to carry this out:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# price_retrieval.py
```

```
from __future__ import print_function

import datetime
import warnings

import MySQLdb as mdb
import requests


# Obtain a database connection to the MySQL instance
db_host = 'localhost'
db_user = 'sec_user'
db_pass = 'password'
db_name = 'securities_master'
con = mdb.connect(db_host, db_user, db_pass, db_name)


def obtain_list_of_db_tickers():
    """
    Obtains a list of the ticker symbols in the database.
    """
    with con:
        cur = con.cursor()
        cur.execute("SELECT id, ticker FROM symbol")
        data = cur.fetchall()
        return [(d[0], d[1]) for d in data]


def get_daily_historic_data_yahoo(
        ticker, start_date=(2000,1,1),
        end_date=datetime.date.today().timetuple()[0:3]
    ):
    """
    Obtains data from Yahoo Finance returns and a list of tuples.

    ticker: Yahoo Finance ticker symbol, e.g. "GOOG" for Google, Inc.
    start_date: Start date in (YYYY, M, D) format
    end_date: End date in (YYYY, M, D) format
    """
    # Construct the Yahoo URL with the correct integer query parameters
    # for start and end dates. Note that some parameters are zero-based!
    ticker_tup = (
        ticker, start_date[1]-1, start_date[2],
        start_date[0], end_date[1]-1, end_date[2],
        end_date[0]
    )
    yahoo_url = "http://ichart.finance.yahoo.com/table.csv"
    yahoo_url += "?s=%s&a=%s&b=%s&c=%s&d=%s&e=%s&f=%s"
    yahoo_url = yahoo_url % ticker_tup

    # Try connecting to Yahoo Finance and obtaining the data
    # On failure, print an error message.
    try:
        yf_data = requests.get(yahoo_url).text.split("\n")[1:-1]
        prices = []
```

```
        for y in yf_data:
            p = y.strip().split(',')
            prices.append(
                (datetime.datetime.strptime(p[0], '%Y-%m-%d'),
                p[1], p[2], p[3], p[4], p[5], p[6])
            )
    except Exception as e:
        print("Could not download Yahoo data: %s" % e)
    return prices


def insert_daily_data_into_db(
        data_vendor_id, symbol_id, daily_data
    ):
    """
    Takes a list of tuples of daily data and adds it to the
    MySQL database. Appends the vendor ID and symbol ID to the data.

    daily_data: List of tuples of the OHLC data (with
    adj_close and volume)
    """
    # Create the time now
    now = datetime.datetime.utcnow()

    # Amend the data to include the vendor ID and symbol ID
    daily_data = [
        (data_vendor_id, symbol_id, d[0], now, now,
        d[1], d[2], d[3], d[4], d[5], d[6])
        for d in daily_data
    ]

    # Create the insert strings
    column_str = """data_vendor_id, symbol_id, price_date, created_date,
                 last_updated_date, open_price, high_price, low_price,
                 close_price, volume, adj_close_price"""
    insert_str = ("%s, " * 11)[:-2]
    final_str = "INSERT INTO daily_price (%s) VALUES (%s)" % \
        (column_str, insert_str)

    # Using the MySQL connection, carry out an INSERT INTO for every symbol
    with con:
        cur = con.cursor()
        cur.executemany(final_str, daily_data)


if __name__ == "__main__":
    # This ignores the warnings regarding Data Truncation
    # from the Yahoo precision to Decimal(19,4) datatypes
    warnings.filterwarnings('ignore')

    # Loop over the tickers and insert the daily historical
    # data into the database
    tickers = obtain_list_of_db_tickers()
    lentickers = len(tickers)
    for i, t in enumerate(tickers):
        print(
```

```
            "Adding data for %s: %s out of %s" %
            (t[1], i+1, lentickers)
        )
        yf_data = get_daily_historic_data_yahoo(t[1])
        insert_daily_data_into_db('1', t[0], yf_data)
    print("Successfully added Yahoo Finance pricing data to DB.")
```

Note that there are certainly ways we can optimise this procedure. If we make use of the Python **ScraPy** library, for instance, we would gain high concurrency from the downloads, as ScraPy is built on the event-driven **Twisted** framework. At the moment each download will be carried out sequentially.

7.9 Retrieving Data from the Securities Master

Now that we've downloaded the historical pricing for all of the current S&P500 constituents we want to be able to access it within Python. The **pandas** library makes this extremely straightforward. Here's a script that obtains the Open-High-Low-Close (OHLC) data for the Google stock over a certain time period from our securities master database and outputs the *tail* of the dataset:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# retrieving_data.py

from __future__ import print_function

import pandas as pd
import MySQLdb as mdb


if __name__ == "__main__":
    # Connect to the MySQL instance
    db_host = 'localhost'
    db_user = 'sec_user'
    db_pass = 'password'
    db_name = 'securities_master'
    con = mdb.connect(db_host, db_user, db_pass, db_name)

    # Select all of the historic Google adjusted close data
    sql = """SELECT dp.price_date, dp.adj_close_price
             FROM symbol AS sym
             INNER JOIN daily_price AS dp
             ON dp.symbol_id = sym.id
             WHERE sym.ticker = 'GOOG'
             ORDER BY dp.price_date ASC;"""

    # Create a pandas dataframe from the SQL query
    goog = pd.read_sql_query(sql, con=con, index_col='price_date')

    # Output the dataframe tail
    print(goog.tail())
```

The output of the script follows:

```
            adj_close_price
price_date
2015-06-09           526.69
```

```
2015-06-10          536.69
2015-06-11          534.61
2015-06-12          532.33
2015-06-15          527.20
```

This is obviously only a simple script, but it shows how powerful having a locally-stored securities master can be. It is possible to backtest certain strategies extremely rapidly with this approach, as the input/output (I/O) speed from the database will be significantly faster than that of an internet connection.

Chapter 8

Processing Financial Data

In the previous chapter we outlined how to construct an equities-based securities master database. This chapter will discuss a topic that is not often considered to any great extent in the majority of trading books, that of processing financial market data prior to usage in a strategy test.

The discussion will begin with an overview of the different types of data that will be of interest to algorithmic traders. The frequency of the data will then be considered, from quarterly data (such as SEC reports) through to tick and order book data on the millisecond scale. Sources of such data (both free and commercial) will then be outlined along with code for obtaining the data. Finally, cleansing and preparation of the data for usage in strategies will be discussed.

8.1 Market and Instrument Classification

As algorithmic traders we are often interested in a broad range of financial markets data. This can range from underlying and derivative instrument time series prices, unstructured text-based data (such as news articles) through to corporate earnings information. This book will predominantly discuss financial time series data.

8.1.1 Markets

US and international equities, foreign exchange, commodities and fixed income are the primary sources of market data that will be of interest to an algorithmic trader. In the equities market it is still extremely common to purchase the underlying asset directly, while in the latter three markets highly liquid derivative instruments (futures, options or more exotic instruments) are more commonly used for trading purposes.

This broad categorisation essentially makes it relatively straightforward to deal in the equity markets, albeit with issues surrounding data handling of corporate actions (see below). Thus a large part of the retail algorithmic trading landscape will be based around equities, such as direct corporate shares or Exchange Traded Funds (ETFs). Foreign exchange ("forex") markets are also highly popular since brokers will allow *margin trading* on *percentage in point* (PIP) movements. A *pip* is one unit of the fourth decimal point in a currency rate. For currencies denominated in US dollars this is equivalent to 1/100th of a cent.

Commodities and fixed income markets are harder to trade in the underlying directly. A retail algorithmic trader is often not interested in delivering barrels of oil to an oil depot! Instead, futures contracts on the underlying asset are used for speculative purposes. Once again, margin trading is employed allowing extensive leverage on such contracts.

8.1.2 Instruments

A wide range of underlying and derivative instruments are available to the algorithmic trader. The following table describes the common use cases of interest.

Market	Instruments
Equities/Indices	Stock, ETFs, Futures, Options
Foreign Exchange	Margin/Spot, ETFs, Futures, Options
Commodities	Futures, Options
Fixed Income	Futures, Options

For the purposes of this book we will concentrate almost exclusively upon equities and ETFs to simplify the implementation.

8.1.3 Fundamental Data

Although algorithmic traders primarily carry out analysis of financial price time series, fundamental data (of varying frequencies) is also often added to the analysis. So-called *Quantitative Value* (QV) strategies rely heavily on the accumulation and analysis of fundamental data, such as macroeconomic information, corporate earnings histories, inflation indexes, payroll reports, interest rates and SEC filings. Such data is often also in temporal format, albeit on much larger timescales over months, quarters or years. QV strategies also operate on these timeframes.

This book will not discuss QV strategies or large time-scale fundamental driven strategies to any great extent, rather will concentrate on the daily or more frequent strategies derived mostly from price action.

8.1.4 Unstructured Data

Unstructured data consists of *documents* such as news articles, blog posts, papers or reports. Analysis of such data can be complicated as it relies on *Natural Language Processing* (NLP) techniques. One such use of analysing unstructured data is in trying to determine the *sentiment* context. This can be useful in driving a trading strategy. For instance, by classifying texts as "bullish", "bearish" or "neutral" a set of trading signals could be generated. The term for this process is *sentiment analysis*.

Python provides an extremely comprehensive library for the analysis of text data known as the Natural Language Toolkit (NLTK). Indeed an O'Reilly book on NLTK can be downloaded for free via the authors' website - Natural Language Processing with Python[3].

Full-Text Data

There are numerous sources of full-text data that may be useful for generating a trading strategy. Popular financial sources such as Bloomberg and the Financial Times, as well as financial commentary blogs such as Seeking Alpha and ZeroHedge, provide significant sources of text to analyse. In addition, proprietary news feeds as provided by data vendors are also good sources of such data.

In order to obtain data on a larger scale, one must make use of "web scraping" tools, which are designed to automate the downloading of websites en-masse. Be careful here as automated web-scraping tools sometimes breach the Terms Of Service for these sites. Make sure to check before you begin downloading this sort of data. A particularly useful tool for web scraping, which makes the process efficient and structured, is the ScraPy library.

Social Media Data

In the last few years there has been significant interest in obtaining sentiment information from social media data, particularly via the Twitter micro-blogging service. Back in 2011, a hedge fund was launched around Twitter sentiment, known as Derwent Capital. Indeed, academic studies[4] have shown evidence that it is possible to generate a degree of predictive capability based on such sentiment analysis.

While sentiment analysis is out of the scope of this book if you wish to carry out research into sentiment, then there are two books[15, 14] by Matt Russell on obtaining social media data via the public APIs provided by these web services.

8.2 Frequency of Data

Frequency of data is one of the most important considerations when designing an algorithmic trading system. It will impact every design decision regarding the storage of data, backtesting a strategy and executing an algorithm.

Higher frequency strategies are likely to lead to more statistically robust analysis, simply due to the greater number of data points (and thus trades) that will be used. HFT strategies often require a significant investment of time and capital for development of the necessary software to carry them out.

Lower frequency strategies are easier to develop and deploy, since they require less automation. However, they will often generate far less trades than a higher-frequency strategy leading to a less statistically robust analysis.

8.2.1 Weekly and Monthly Data

Fundamental data is often reported on a weekly, monthly, quarterly or even yearly basis. Such data include payroll data, hedge fund performance reports, SEC filings, inflation-based indices (such as the Consumer Price Index, CPI), economic growth and corporate accounts.

Storage of such data is often suited to unstructured databases, such as MongoDB, which can handle hierarchically-nested data and thus allowing a reasonable degree of querying capability. The alternative is to store flat-file text in a RDBMS, which is less appropriate, since full-text querying is trickier.

8.2.2 Daily Data

The majority of retail algorithmic traders make use of daily ("end of day"/EOD) financial time series data, particularly in equities and foreign exchange. Such data is freely available (see below), but often of questionable quality and subject to certain biases. End-of-day data is often stored in RDBMS, since the nature of ticker/symbol mapping naturally applies to the relational model.

EOD data does not entail particularly large storage requirements. There are 252 trading days in a year for US exchanges and thus for a decade there will be 2,520 bars per security. Even with a universe of 10,000 symbols this is 25,200,000 bars, which can easily be handled within a relational database environment.

8.2.3 Intraday Bars

Intraday strategies often make use of hourly, fifteen-, five-, one-minutely or secondly OHLCV bars. Intraday feed providers such as QuantQuote and DTN IQFeed will often provide minutely or secondly bars based on their tick data.

Data at such frequencies will possess many "missing" bars simply because no trades were carried out in that time period. Pandas can be used to pad these values forward, albeit with a decrease in data accuracy. In addition pandas can also be used to create data on less granular timescales if necessary.

For a ten year period, minutely data will generate almost one million bars per security. Similarly for secondly data the number of data points over the same period will total almost sixty million per security. Thus to store one thousand of such securities will lead to sixty billion bars of data. This is a large amount of data to be kept in an RDBMS and consequently more sophisticated approaches are required.

Storage and retrieval of secondly data on this magnitude is somewhat outside the scope of this book so I won't discuss it further.

8.2.4 Tick and Order Book Data

When a trade is filled at an exchange, or other venue, a *tick* is generated. Tick feeds consist of all such transactions *per exchange*. Retail tick feeds are stored with each datum having a timestamp accurate to the millisecond level. Tick data often also includes the updated best bid/ask price. The storage of tick data is well beyond the scope of this book but needless to say

the volumes of such data are substantial. Common storage mechanisms include HDF5, kdb and simply flat-file/CSV.

Multiple *limit orders* at an exchange lead to the concept of an *order book*. This is essentially the list of all bid and ask limit orders at certain volumes for each market participant. It leads to the definition of the *bid-ask spread* (or simply the "spread"), which is the smallest difference in the bid and ask prices for the "top of book" orders. Creating a historical representation, or a market simulator, of a limit order book is usually necessary for carrying out ultra high frequency trading (UHFT) strategies. The storage of such data is complex and as such will be outside the scope of this book.

8.3 Sources of Data

There are numerous sources and vendors of financial data. They vary substantially in breadth, timeliness, quality and price.

Broadly speaking, financial market data provided on a delayed daily frequency or longer is available freely, albeit with dubious overall quality and the potential for survivorship bias. To obtain intraday data it is usually necessary to purchase a commercial data feed. The vendors of such feeds vary tremendously in their customer service capability, overall feed quality and breadth of instruments.

8.3.1 Free Sources

Free end-of-day bar data, which consists of Open-High-Low-Close-Volume (OHLCV) prices for instruments, is available for a wide range of US and international equities and futures from Yahoo Finance, Google Finance and Quandl.

Yahoo Finance

Yahoo Finance is the "go to" resource when forming an end-of-day US equities database. The breadth of data is extremely comprehensive, listing thousands of traded equities. In addition stock-splits and dividends are handled using a back-adjustment method, arising as the "Adj Close" column in the CSV output from the API (which we discuss below). Thus the data allows algorithmic traders to get started rapidly and for zero cost.

I have personally had a lot of experience in cleaning Yahoo data. I have to remark that the data can be quite erroneous. Firstly, it is subject to a problem known as *backfilling*. This problem occurs when past historical data is corrected at a future date, leading to poor quality backtests that change as your own database is re-updated. To handle this problem, a logging record is usually added to the securities master (in an appropriate logging table) whenever a historical data point is modified.

Secondly, the Yahoo feed only aggregates prices from a few sources to form the OHLCV points. This means that values around the open, high, low and close can be deceptive, as other exchanges/liquidity sources may have executed differing prices in excess of the values.

Thirdly, I have noticed that when obtaining financial data *en-masse* from Yahoo, that errors do creep into the API. For instance, multiple calls to the API with identical date/ticker parameters occasionally lead to differing result sets. This is clearly a substantial problem and must be carefully checked for.

In summary be prepared to carry out some extensive data cleansing on Yahoo Finance data, if you choose to use it to populate a large securities master, and need highly accurate data.

Quandl

Quandl is a relatively new service which purports to be *"The easiest way to find and use numerical data on the web"*. I believe they are well on the way to achieving that goal! The service provides a substantial daily data set of US and international equities, interest rates, commodities/futures, foreign exchange and other economic data. In addition, the database is continually expanded and the project is very actively maintained.

All of the data can be accessed by a very modern HTTP API (CSV, JSON, XML or HTML), with plugins for a wide variety of programming languages including R, Python, Matlab, Excel, Stata, Maple, C#, EViews, Java, C/C++, .NET, Clojure and Julia. Without an account 50 calls to the API are allowed per day, but this can be increased to 500 if registering an account. In fact, calls can be updated to 5,000 per hour if so desired by contacting the team.

I have not had a great deal of experience with Quandl "at scale" and so I can't comment on the level of errors within the dataset, but my feeling is that any errors are likely to be constantly reported and corrected. Thus they are worth considering as a primary data source for an end-of-day securities master.

Later in the chapter we will discuss how to obtain US Commodities Futures data from Quandl with Python and pandas.

8.3.2 Commercial Sources

In order to carry out intraday algorithmic trading it is usually necessary to purchase a commercial feed. Pricing can range anywhere from $30 per month to around $500 per month for "retail level" feeds. Institutional quality feeds will often be in the low-to-mid four figure range per month and as such I won't discuss them here.

EODData

I have utilised EODData in a fund context, albeit only with daily data and predominantly for foreign exchange. Despite their name they do provide a degree of intraday sources. The cost is $25 per month for their "platinum" package.

The resource is very useful for finding a full list of traded symbols on global exchanges, but remember that this will be subject to survivorship bias as I believe the list represents current listed entities.

Unfortunately (at least back in 2010) I found that the stock split feed was somewhat inaccurate (at least when compared to Morningstar information). This lead to some substantial spike issues (see below) in the data, which increased friction in the data cleansing process.

DTN IQFeed

DTN IQFeed are one of the most popular data feeds for the high-end retail algorithmic trader. They claim to have over 80,000 customers. They provide real-time tick-by-tick data unfiltered from the exchange as well as a large quantity of historic data.

The pricing starts at $50 per month, but in reality will be in the $150-$200 per month range once particular services are selected and exchange fees are factored in. I utilise DTN IQFeed for all of my intraday equities and futures strategies. In terms of historical data, IQFeed provide for equities, futures and options:

- 180 calendar days of tick (every trade)
- 7+ years of 1 minute historical bars
- 15+ years of daily historical bars

The major disadvantage is that the DTN IQFeed software (the mini-server, not the charting tools) will only run on Windows. This may not be a problem if all of your algorithmic trading is carried out in this operating system, but I personally develop all my strategies in Ubuntu Linux. However, although I have not actively tested it, I have heard it is possible to run DTN IQFeed under the WINE emulator.

Below we will discuss how to obtain data from IQFeed using Python in Windows.

QuantQuote

QuantQuote provide reasonably priced historical minute-, second- and tick-level data for US equities going back to 1998. In addition they provide institutional level real-time tick feeds, although this is of less interest to retail algorithmic traders. One of the main benefits of QuantQuote is that their data is provided free of survivorship bias, due to their TickMap symbol-matching software and inclusion of all stocks within a certain index through time.

As an example, to purchase the entire history of the S&P500 going back to 1998 in minutely-bars, inclusive of de-listed stocks, the cost at the time of writing was $895. The pricing scales with increasing frequency of data.

QuantQuote is currently the primary provider of market data to the QuantConnect web-based backtesting service. QuantQuote go to great lengths to ensure minimisation of error, so if you are looking for a US equities only feed at high resolution, then you should consider using their service.

8.4 Obtaining Data

In this section we are going to discuss how to use Quandl, pandas and DTN IQFeed to download financial market data across a range of markets and timeframes.

8.4.1 Yahoo Finance and Pandas

The pandas library makes it exceedingly simple to download EOD data from Yahoo Finance. Pandas ships with a DataReader component that ties into Yahoo Finance (among other sources). Specifying a symbol with a start and end date is sufficient to download an EOD series into a pandas DataFrame, which allows rapid vectorised operations to be carried out:

```
from __future__ import print_function

import datetime
import pandas.io.data as web

if __name__ == "__main__":
    spy = web.DataReader(
        "SPY", "yahoo",
        datetime.datetime(2007,1,1),
        datetime.datetime(2015,6,15)
    )
    print(spy.tail())
```

The output is given below:

```
                 Open        High         Low       Close     Volume  \
Date
2015-06-09  208.449997  209.100006  207.690002  208.449997   98148200
2015-06-10  209.369995  211.410004  209.300003  210.960007  129936200
2015-06-11  211.479996  212.089996  211.199997  211.649994   72672100
2015-06-12  210.639999  211.479996  209.679993  209.929993  127811900
2015-06-15  208.639999  209.449997  207.789993  209.100006  121425800

             Adj Close
Date
2015-06-09  208.449997
2015-06-10  210.960007
2015-06-11  211.649994
2015-06-12  209.929993
2015-06-15  209.100006
```

Note that in pandas 0.17.0, `pandas.io.data` *will be replaced by a separate* `pandas-datareader` *package. However, for the time being (i.e. pandas versions 0.16.x) the syntax to import the data reader is* `import pandas.io.data as web`.

In the next section we will use Quandl to create a more comprehensive, permanent download solution.

8.4.2 Quandl and Pandas

Up until recently it was rather difficult and expensive to obtain consistent futures data across exchanges in frequently updated manner. However, the release of the Quandl service has changed the situation dramatically, with financial data in some cases going back to the 1950s. In this section we will use Quandl to download a set of end-of-day futures contracts across multiple delivery dates.

Signing Up For Quandl

The first thing to do is sign up to Quandl. This will increase the daily allowance of calls to their API. Sign-up grants 500 calls per day, rather than the default 50. Visit the site at www.quandl.com:

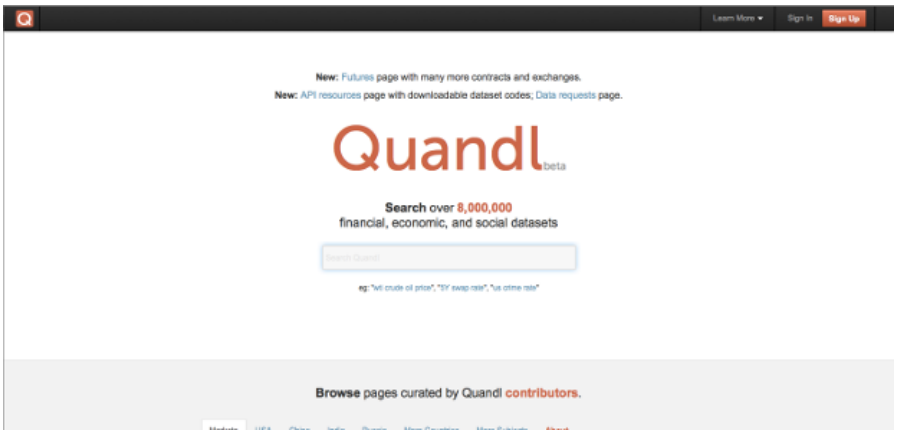


Figure 8.1: The Quandl homepage

Click on the sign-up button on the top right:

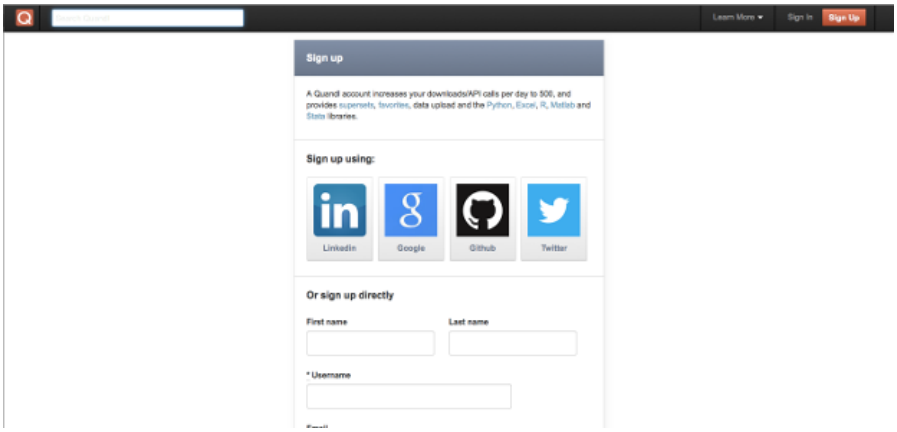


Figure 8.2: The Quandl sign-up page

Once you're signed in you'll be returned to the home page:

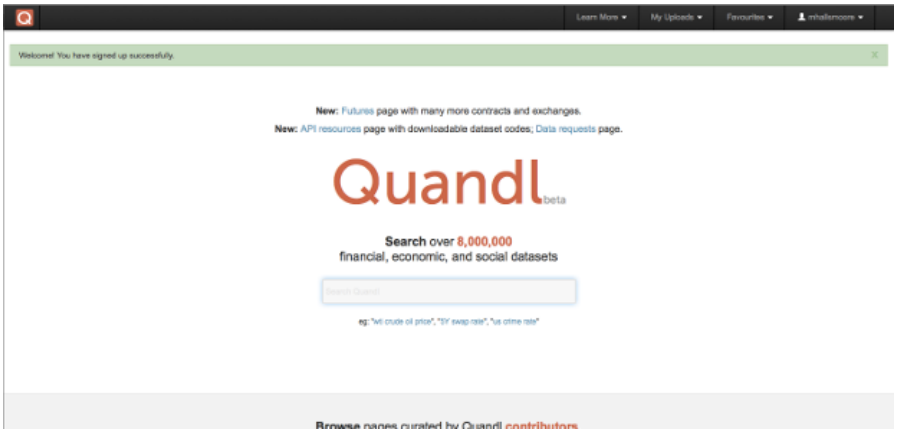


Figure 8.3: The Quandl authorised home page

Quandl Futures Data

Now click on the "New: Futures page..." link to get to the futures homepage:

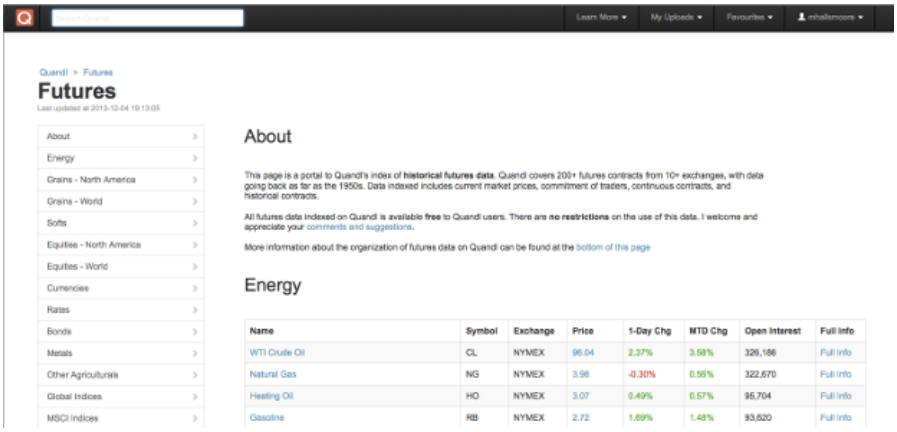


Figure 8.4: The Quandl futures contracts home page

For this tutorial we will be considering the highly liquid E-Mini S&P500 futures contract, which has the futures symbol ES. To download other contracts the remainder of this tutorial can be carried out with additional symbols replacing the reference to ES.

Click on the E-Mini S&P500 link (or your chosen futures symbol) and you'll be taken to the following screen:

Scrolling further down the screen displays the list of historical contracts going back to 1997:

Click on one of the individual contracts. As an example, I have chosen ESZ2014, which refers to the contract for December 2014 'delivery'. This will display a chart of the data:

By clicking on the "Download" button the data can be obtained in multiple formats: HTML, CSV, JSON or XML. In addition we can download the data directly into a pandas DataFrame using the Python bindings. While the latter is useful for quick "prototyping" and exploration of the data, in this section we are considering the development of a longer term data store. Click the download button, select "CSV" and then copy and paste the API call:

The API call will have the following form:

```
http://www.quandl.com/api/v1/datasets/OFDP/FUTURE_ESZ2014.csv?
&auth_token=MY_AUTH_TOKEN&trim_start=2013-09-18
&trim_end=2013-12-04&sort_order=desc
```

The authorisation token has been redacted and replaced with MY_AUTH_TOKEN. It will be necessary to copy the alphanumeric string between "auth_token=" and "&trim_start" for

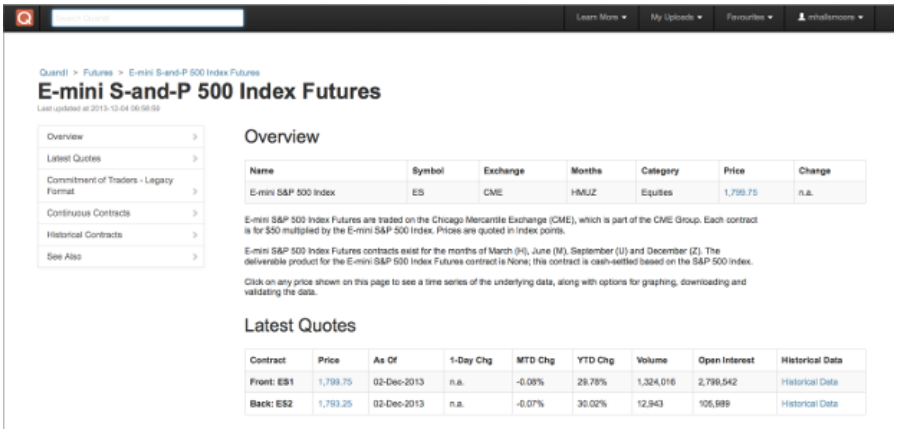


Figure 8.5: E-Mini S&P500 contract page

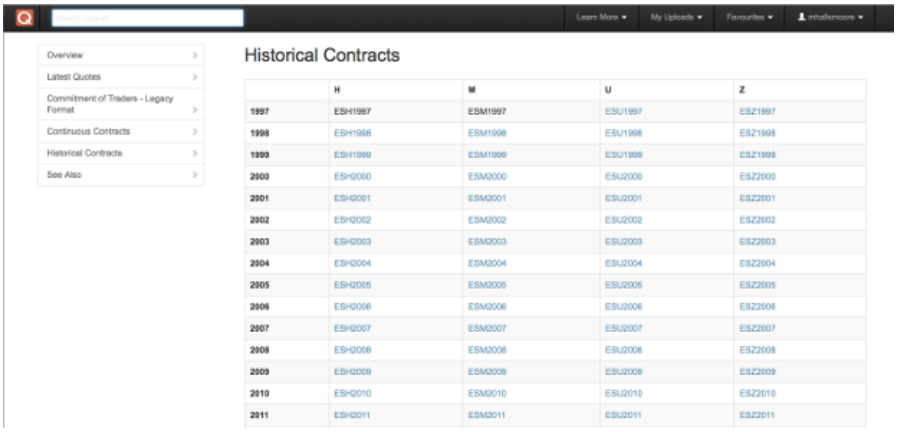


Figure 8.6: E-Mini S&P500 historical contracts

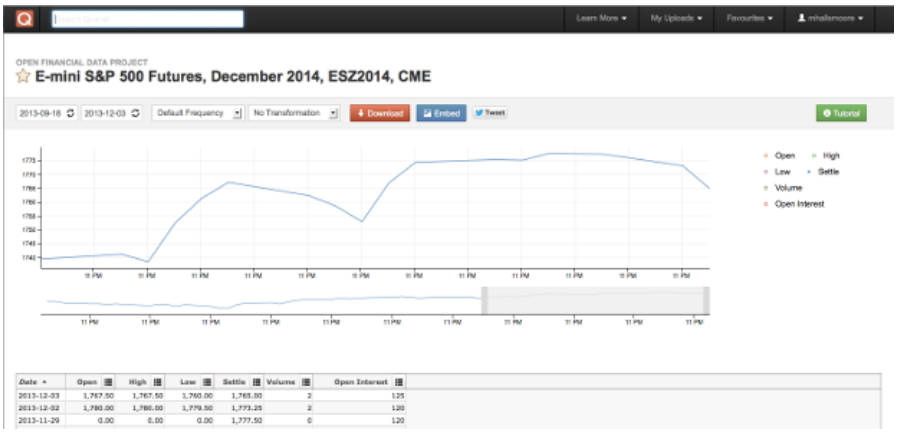


Figure 8.7: Chart of ESZ2014 (December 2014 delivery)

later usage in the Python script below. Do not share it with anyone as it is your unique authorisation token for Quandl downloads and is used to determine your download rate for the day.

This API call will form the basis of an automated script which we will write below to download a subset of the entire historical futures contract.

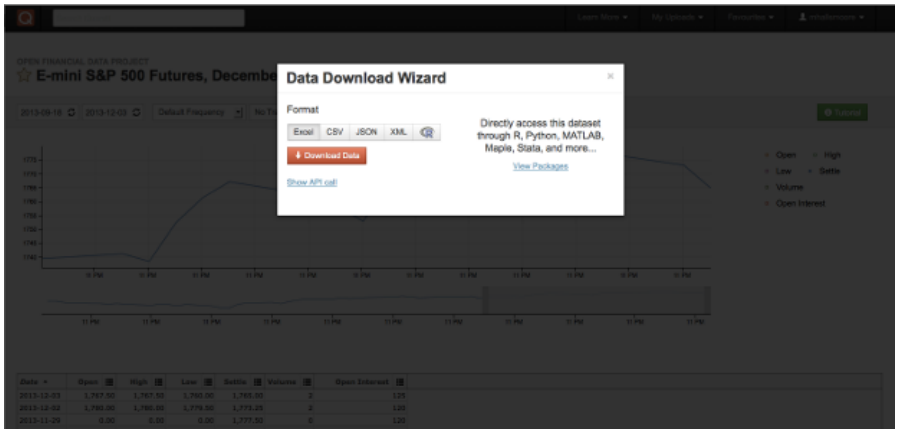


Figure 8.8: Download modal for ESZ2014 CSV file

Downloading Quandl Futures into Python

Because we are interested in using the futures data long-term as part of a wider securities master database strategy we want to store the futures data to disk. Thus we need to create a directory to hold our E-Mini contract CSV files. In Mac/Linux (within the terminal/console) this is achieved by the following command:

```
cd /PATH/TO/YOUR/quandl_data.py
mkdir -p quandl/futures/ES
```

Note: Replace `/PATH/TO/YOUR` above with the directory where your `quandl_data.py` file is located.

This creates a subdirectory of called `quandl`, which contains two further subdirectories for futures and for the ES contracts in particular. This will help us to organise our downloads in an ongoing fashion.

In order to carry out the download using Python we will need to import some libraries. In particular we will need **requests** for the download and **pandas** and **matplotlib** for plotting and data manipulation:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# quandl_data.py

from __future__ import print_function

import matplotlib.pyplot as plt
import pandas as pd
import requests
```

The first function within the code will generate the list of futures symbols we wish to download. I've added keyword parameters for the start and end years, setting them to reasonable values of 2010 and 2014. You can, of course, choose to use other timeframes:

```
def construct_futures_symbols(
        symbol, start_year=2010, end_year=2014
    ):
    """
    Constructs a list of futures contract codes
    for a particular symbol and timeframe.
    """
    futures = []
```

```
    # March, June, September and
    # December delivery codes
    months = 'HMUZ'
    for y in range(start_year, end_year+1):
        for m in months:
            futures.append("%s%s%s" % (symbol, m, y))
    return futures
```

Now we need to loop through each symbol, obtain the CSV file from Quandl for that particular contract and subsequently write it to disk so we can access it later:

```
def download_contract_from_quandl(contract, dl_dir):
    """
    Download an individual futures contract from Quandl and then
    store it to disk in the 'dl_dir' directory. An auth_token is
    required, which is obtained from the Quandl upon sign-up.
    """
    # Construct the API call from the contract and auth_token
    api_call = "http://www.quandl.com/api/v1/datasets/"
    api_call += "OFDP/FUTURE_%s.csv" % contract
    # If you wish to add an auth token for more downloads, simply
    # comment the following line and replace MY_AUTH_TOKEN with
    # your auth token in the line below
    params = "?sort_order=asc"
    #params = "?auth_token=MY_AUTH_TOKEN&sort_order=asc"
    full_url = "%s%s" % (api_call, params)

    # Download the data from Quandl
    data = requests.get(full_url).text

    # Store the data to disk
    fc = open('%s/%s.csv' % (dl_dir, contract), 'w')
    fc.write(data)
    fc.close()
```

Now we tie the above two functions together to download all of the desired contracts:

```
def download_historical_contracts(
        symbol, dl_dir, start_year=2010, end_year=2014
    ):
    """
    Downloads all futures contracts for a specified symbol
    between a start_year and an end_year.
    """
    contracts = construct_futures_symbols(
        symbol, start_year, end_year
    )
    for c in contracts:
        print("Downloading contract: %s" % c)
        download_contract_from_quandl(c, dl_dir)
```

Finally, we can add one of the futures prices to a pandas dataframe using the main function. We can then use matplotlib to plot the settle price:

```
if __name__ == "__main__":
    symbol = 'ES'

    # Make sure you've created this
    # relative directory beforehand
```

```
dl_dir = 'quandl/futures/ES'

# Create the start and end years
start_year = 2010
end_year = 2014

# Download the contracts into the directory
download_historical_contracts(
    symbol, dl_dir, start_year, end_year
)

# Open up a single contract via read_csv
# and plot the settle price
es = pd.io.parsers.read_csv(
    "%s/ESH2010.csv" % dl_dir, index_col="Date"
)
es["Settle"].plot()
plt.show()
```

The output of the plot is given in Figure 8.4.2.

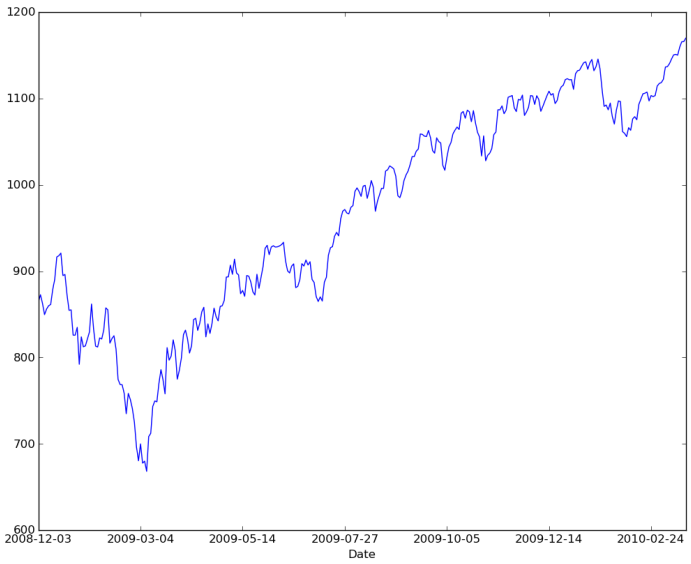


Figure 8.9: ESH2010 Settle Price

The above code can be modified to collect any combination of futures contracts from Quandl as necessary. Remember that unless a higher API request is made, the code will be limited to making 50 API requests per day.

8.4.3 DTN IQFeed

For those of you who possess a DTN IQFeed subscription, the service provides a client-server mechanism for obtaining intraday data. For this to work it is necessary to download the IQLink server and run it on Windows. Unfortunately, it is tricky to execute this server on Mac or Linux unless making use of the WINE emulator. However once the server is running it can be connected to via a socket at which point it can be queried for data.

In this section we will obtain minutely bar data for a pair of US ETFs from January 1st 2007 onwards using a Python socket interface. Since there are approximately 252 trading days within

each year for US markets, and each trading day has 6.5 hours of trading, this will equate to at least 650,000 bars of data, each with seven data points: Timestamp, Open, Low, High, Close, Volume and Open Interest.

I have chosen the SPY and IWM ETFs to download to CSV. Make such to start the IQLink program in Windows before executing this script:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# iqfeed.py

import sys
import socket

def read_historical_data_socket(sock, recv_buffer=4096):
    """
    Read the information from the socket, in a buffered
    fashion, receiving only 4096 bytes at a time.

    Parameters:
    sock - The socket object
    recv_buffer - Amount in bytes to receive per read
    """
    buffer = ""
    data = ""
    while True:
        data = sock.recv(recv_buffer)
        buffer += data

        # Check if the end message string arrives
        if "!ENDMSG!" in buffer:
            break

    # Remove the end message string
    buffer = buffer[:-12]
    return buffer

if __name__ == "__main__":
    # Define server host, port and symbols to download
    host = "127.0.0.1"  # Localhost
    port = 9100  # Historical data socket port
    syms = ["SPY", "IWM"]

    # Download each symbol to disk
    for sym in syms:
        print "Downloading symbol: %s..." % sym

        # Construct the message needed by IQFeed to retrieve data
        message = "HIT,%s,60,20070101 075000,,,093000,160000,1\n" % sym

        # Open a streaming socket to the IQFeed server locally
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.connect((host, port))

        # Send the historical data request
        # message and buffer the data
        sock.sendall(message)
```

```
        data = read_historical_data_socket(sock)
        sock.close

        # Remove all the endlines and line-ending
        # comma delimiter from each record
        data = "".join(data.split("\r"))
        data = data.replace(",\n","\n")[:-1]

        # Write the data stream to disk
        f = open("%s.csv" % sym, "w")
        f.write(data)
        f.close()
```

With additional subscription options in the DTN IQFeed account, it is possible to download individual futures contracts (and back-adjusted continuous contracts), options and indices. DTN IQFeed also provides real-time tick streaming, but this form of data falls outside the scope of the book.

8.5 Cleaning Financial Data

Subsequent to the delivery of financial data from vendors it is necessary to perform *data cleansing*. Unfortunately this can be a painstaking process, but a hugely necessary one. There are multiple issues that require resolution: Incorrect data, consideration of data aggregation and backfilling. Equities and futures contracts possess their own unique challenges that must be dealt with prior to strategy research, including back/forward adjustment, continuous contract stitching and corporate action handling.

8.5.1 Data Quality

The reputation of a data vendor will often rest on its (perceived) data quality. In simple terms, bad or missing data leads to erroneous trading signals and thus potential loss. Despite this fact, many vendors are still plagued with poor or inconsistent data quality. Thus there is always a cleansing process necessary to be carried.

The main culprits in poor data quality are conflicting/incorrect data, opaque aggregation of multiple data sources and error correction ("backfilling").

Conflicting and Incorrect Data

Bad data can happen anywhere in the stream. Bugs in the software at an exchange can lead to erroneous prices when matching trades. This filters through to the vendor and subsequently the trader. Reputable vendors will attempt to flag upstream "bad ticks" and will often leave the "correction" of these points to the trader.

8.5.2 Continuous Futures Contracts

In this section we are going to discuss the characteristics of futures contracts that present a data challenge from a backtesting point of view. In particular, the notion of the "continuous contract". We will outline the main difficulties of futures and provide an implementation in Python with pandas that can partially alleviate the problems.

Brief Overview of Futures Contracts

Futures are a form of *contract* drawn up between two parties for the purchase or sale of a quantity of an underlying asset at a specified date in the future. This date is known as the *delivery* or *expiration*. When this date is reached the buyer must deliver the physical underlying (or cash equivalent) to the seller for the price agreed at the contract formation date.

In practice futures are traded on exchanges (as opposed to *Over The Counter* - OTC trading) for standardised quantities and qualities of the underlying. The prices are *marked to market* every day. Futures are incredibly liquid and are used heavily for speculative purposes. While futures were often utilised to hedge the prices of agricultural or industrial goods, a futures contract can be formed on any tangible or intangible underlying such as stock indices, interest rates of foreign exchange values.

A detailed list of all the symbol codes used for futures contracts across various exchanges can be found on the CSI Data site: Futures Factsheet.

The main difference between a futures contract and equity ownership is the fact that a futures contract has a limited window of availability by virtue of the expiration date. At any one instant there will be a variety of futures contracts on the same underlying all with varying dates of expiry. The contract with the nearest date of expiry is known as the *near contract*. The problem we face as quantitative traders is that at any point in time we have a choice of multiple contracts with which to trade. Thus we are dealing with an overlapping set of time series rather than a continuous stream as in the case of equities or foreign exchange.

The goal of this section is to outline various approaches to constructing a continuous stream of contracts from this set of multiple series and to highlight the tradeoffs associated with each technique.

Forming a Continuous Futures Contract

The main difficulty with trying to generate a continuous contract from the underlying contracts with varying deliveries is that the contracts do not often trade at the same prices. Thus situations arise where they do not provide a smooth splice from one to the next. This is due to contango and backwardation effects. There are various approaches to tackling this problem, which we now discuss.

Unfortunately there is no single "standard" method for joining futures contracts together in the financial industry. Ultimately the method chosen will depend heavily upon the strategy employing the contracts and the method of execution. Despite the fact that no single method exists there are some common approaches:

The **Back/Forward ("Panama") Adjustment** method alleviates the "gap" across multiple contracts by shifting each contract such that the individual deliveries join in a smooth manner to the adjacent contracts. Thus the open/close across the prior contracts at expiry matches up.

The key problem with the Panama method includes the introduction of a trend bias, which will introduce a large drift to the prices. This can lead to negative data for sufficiently historical contracts. In addition there is a loss of the *relative* price differences due to an absolute shift in values. This means that returns are complicated to calculate (or just plain incorrect).

The **Proportionality Adjustment** approach is similar to the adjustment methodology of handling stock splits in equities. Rather than taking an absolute shift in the successive contracts, the ratio of the older settle (close) price to the newer open price is used to proportionally adjust the prices of historical contracts. This allows a continous stream without an interruption of the calculation of percentage returns.

The main issue with proportional adjustment is that any trading strategies reliant on an absolute price level will also have to be similarly adjusted in order to execute the correct signal. This is a problematic and error-prone process. Thus this type of continuous stream is often only useful for summary statistical analysis, as opposed to direct backtesting research.

The **Rollover/Perpetual Series** method creates a continuous contract of successive contracts by taking a linearly weighted proportion of each contract over a number of days to ensure a smoother transition between each.

For example consider five smoothing days. The price on day 1, P_1, is equal to 80% of the far contract price (F_1) and 20% of the near contract price (N_1). Similarly, on day 2 the price is $P_2 = 0.6 \times F_2 + 0.4 \times N_2$. By day 5 we have $P_5 = 0.0 \times F_5 + 1.0 \times N_5 = N_5$ and the contract then just becomes a continuation of the near price. Thus after five days the contract is smoothly transitioned from the far to the near.

The problem with the rollover method is that it requires trading on all five days, which can increase transaction costs. There are other less common approaches to the problem but we will

avoid them here.

The remainder of the section will concentrate on implementing the perpetual series method as this is most appropriate for backtesting. It is a useful way to carry out *strategy pipeline research*.

We are going to stitch together the WTI Crude Oil "near" and "far" futures contract (symbol CL) in order to generate a continuous price series. At the time of writing (January 2014), the near contract is CLF2014 (January) and the far contract is CLG2014 (February).

In order to carry out the download of futures data I've made use of the Quandl plugin. Make sure to set the correct Python virtual environment on your system and install the Quandl package by typing the following into the terminal:

```
pip install Quandl
```

Now that the Quandl package is intalled, we need to make use of NumPy and pandas in order to carry out the rollover construction. Create a new file and enter the following import statements:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# cont_futures.py

from __future__ import print_function

import datetime

import numpy as np
import pandas as pd
import Quandl
```

The main work is carried out in the `futures_rollover_weights` function. It requires a starting date (the first date of the near contract), a dictionary of contract settlement dates (`expiry_dates`), the symbols of the contracts and the number of days to roll the contract over (defaulting to five). The comments below explain the code:

```
def futures_rollover_weights(start_date, expiry_dates,
    contracts, rollover_days=5):
    """This constructs a pandas DataFrame that contains weights
    (between 0.0 and 1.0) of contract positions to hold in order to
    carry out a rollover of rollover_days prior to the expiration of
    the earliest contract. The matrix can then be 'multiplied' with
    another DataFrame containing the settle prices of each
    contract in order to produce a continuous time series
    futures contract."""

    # Construct a sequence of dates beginning
    # from the earliest contract start date to the end
    # date of the final contract
    dates = pd.date_range(start_date, expiry_dates[-1], freq='B')

    # Create the 'roll weights' DataFrame that will store the multipliers for
    # each contract (between 0.0 and 1.0)
    roll_weights = pd.DataFrame(np.zeros((len(dates), len(contracts))),
                                index=dates, columns=contracts)
    prev_date = roll_weights.index[0]

    # Loop through each contract and create the specific weightings for
    # each contract depending upon the settlement date and rollover_days
    for i, (item, ex_date) in enumerate(expiry_dates.iteritems()):
        if i < len(expiry_dates) - 1:
```

```
            roll_weights.ix[prev_date:ex_date - pd.offsets.BDay(), item] = 1
            roll_rng = pd.date_range(end=ex_date - pd.offsets.BDay(),
                                     periods=rollover_days + 1, freq='B')

            # Create a sequence of roll weights (i.e. [0.0,0.2,...,0.8,1.0]
            # and use these to adjust the weightings of each future
            decay_weights = np.linspace(0, 1, rollover_days + 1)
            roll_weights.ix[roll_rng, item] = 1 - decay_weights
            roll_weights.ix[roll_rng,
                expiry_dates.index[i+1]] = decay_weights
        else:
            roll_weights.ix[prev_date:, item] = 1
        prev_date = ex_date
    return roll_weights
```

Now that the weighting matrix has been produced, it is possible to apply this to the individual time series. The main function downloads the near and far contracts, creates a single DataFrame for both, constructs the rollover weighting matrix and then finally produces a continuous series of both prices, appropriately weighted:

```
if __name__ == "__main__":
    # Download the current Front and Back (near and far) futures contracts
    # for WTI Crude, traded on NYMEX, from Quandl.com. You will need to
    # adjust the contracts to reflect your current near/far contracts
    # depending upon the point at which you read this!
    wti_near = Quandl.get("OFDP/FUTURE_CLF2014")
    wti_far = Quandl.get("OFDP/FUTURE_CLG2014")
    wti = pd.DataFrame({'CLF2014': wti_near['Settle'],
                        'CLG2014': wti_far['Settle']}, index=wti_far.index)

    # Create the dictionary of expiry dates for each contract
    expiry_dates = pd.Series(
        {'CLF2014': datetime.datetime(2013, 12, 19),
         'CLG2014': datetime.datetime(2014, 2, 21)}).order()

    # Obtain the rollover weighting matrix/DataFrame
    weights = futures_rollover_weights(wti_near.index[0],
                                       expiry_dates, wti.columns)

    # Construct the continuous future of the WTI CL contracts
    wti_cts = (wti * weights).sum(1).dropna()

    # Output the merged series of contract settle prices
    print(wti_cts.tail(60))
```

The output is as follows:

```
2013-10-14    102.230
2013-10-15    101.240
2013-10-16    102.330
2013-10-17    100.620
2013-10-18    100.990
2013-10-21     99.760
2013-10-22     98.470
2013-10-23     97.000
2013-10-24     97.240
2013-10-25     97.950
..
```

```
..
2013-12-24     99.220
2013-12-26     99.550
2013-12-27    100.320
2013-12-30     99.290
2013-12-31     98.420
2014-01-02     95.440
2014-01-03     93.960
2014-01-06     93.430
2014-01-07     93.670
2014-01-08     92.330
Length: 60, dtype: float64
```

It can be seen that the series is now continuous across the two contracts. This can be extended to handle multiple deliveries across a variety of years, depending upon your backtesting needs.

Part IV

Modelling

Chapter 9

Statistical Learning

The goal of the Modelling section within this book is to provide a robust quantitative framework for identifying relationships in financial market data that can be exploited to generate profitable trading strategies. The approach that will be utilised is that of *Statistical Learning*. This chapter describes the philosophy of statistical learning and associated techniques that can be used to create quantitative models for financial trading.

9.1 What is Statistical Learning?

Before discussing the theoretical aspects of statistical learning it is appropriate to consider an example of a situation from quantitative finance where such techniques are applicable. Consider a quantitative fund that wishes to make long term predictions of the S&P500 stock market index. The fund has managed to collect a substantial amount of *fundamental data* associated with the companies that constitute the index. Fundamental data includes *price-earnings ratio* or *book value*, for instance. How should the fund go about using this data to make predictions of the index in order to create a trading tool? Statistical learning provides one such approach to this problem.

In a more quantitative sense we are attempting to model the behaviour of an *outcome* or *response* based on a set of *predictors* or *features* assuming a relationship between the two. In the above example the stock market index value is the response and the fundamental data associated with the constituent firms are the predictors.

This can be formalised by considering a response Y with p different features $x_1, x_2, ..., x_p$. If we utilise *vector notation* then we can define $X = (x_1, x_2, ..., x_p)$, which is a vector of length p. Then the model of our relationship is given by:

$$Y = f(X) + \epsilon \tag{9.1}$$

Where f is an unknown function of the predictors and ϵ represents an *error* or *noise term*. Importantly, ϵ is not dependent on the predictors and has a mean of zero. This term is included to represent information that is not considered within f. Thus we can return to the stock market index example to say that Y represents the value of the S&P500 whereas the x_i components represent the values of individual fundamental factors.

The goal of statistical learning is to *estimate* the form of f based on the observed data and to evaluate how accurate those estimates are.

9.1.1 Prediction and Inference

There are two general tasks that are of interest in statistical learning - *prediction* and *inference*.

Prediction is concerned with predicting a response Y based on a *newly observed* predictor, X. If the model relationship has been determined then it is simple to predict the response using an estimate for f to produce an estimate for the response:

$$\hat{Y} = \hat{f}(X) \tag{9.2}$$

The functional form of f is often unimportant in a prediction scenario assuming that the estimated responses are close to the true responses and is thus accurate in its predictions. Different estimates of f will produce various accuracies of the estimates of Y. The error associated with having a poor estimate $\hat{f}$ of f is called the *reducible error*. Note that there is always a degree of *irreducible error* because our original specification of the problem included the ϵ error term. This error term encapsulates the unmeasured factors that may affect the response Y. The approach taken is to try and minimise the reducible error with the understanding that there will always be an upper limit of accuracy based on the irreducible error.

Inference is concerned with the situation where there is a need to understand the relationship between X and Y and hence its exact form must be determined. One may wish to identify important predictors or determine the relationship between individual predictors and the response. One could also ask if the relationship is *linear* or *non-linear*. The former means the model is likely to be more interpretable but at the expense of potentially worse predictability. The latter provides models which are generally more predictive but are sometimes less interpretable. Hence a trade-off between *predictability* and *interpretability* often exists.

In this book we are less concerned with inference models since the actual form of f is not as important as its ability to make accurate predictions. Hence a large component of the Modelling section within the book will be based on predictive modelling. The next section deals with how we go about constructing an estimate $\hat{f}$ for f.

9.1.2 Parametric and Non-Parametric Models

In a statistical learning situation it is often possible to construct a set of tuples of predictors and responses of the form $\{(X_1, Y_1), (X_2, Y_2), ..., (X_N, Y_N)\}$, where X_j refers to the jth predictor vector and not the jth component of a particular predictor vector (that is denoted by x_j). A data set of this form is known as *training data* since it will be used to *train* a particular statistical learning method on how to generate $\hat{f}$. In order to actually estimate f we need to find a $\hat{f}$ that provides a reasonable approximation to a particular Y under a particular predictor X. There are two broad categories of statistical models that allow us to achieve this. They are known as *parametric* and *non-parametric* models.

Parametric Models

The defining feature of parametric methods is that they require the *specification* or *assumption* of the form of f. This is a modelling decision. The first choice is whether to consider a linear or non-linear model. Let's consider the simpler case of a linear model. Such a model reduces the problem from estimation of some unknown function of dimension p to that of estimating a coefficient vector $\beta = (\beta_0, \beta_1, ..., \beta_p)$ of length $p + 1$.

Why $p + 1$ and not p? Since linear models can be *affine*, that is they may not pass through the origin when creating a "line of best fit", a coefficient is required to specify the "intercept". In a one-dimensional linear model (regression) setting this is often represented as α. For our multi-dimensional linear model, where there are p predictors, we need an additional value β_0 to represent our intercept and hence there are $p + 1$ components in our $\hat{\beta}$ estimate of β.

Now that we have specified a (linear) functional form of f we need to *train* it. "Training" in this instance means finding an estimate for β such that:

$$Y \approx \hat{\beta}^T X = \beta_0 + \beta_1 x_1 + ... + \beta_p x_p \tag{9.3}$$

In the linear setting we can use an algorithm such as *ordinary least squares* (OLS) but other methods are available as well. It is far simpler to estimate β than fit a (potentially non-linear) f. However, by choosing a parametric linear approach our estimate $\hat{f}$ is unlikely to be replicating the true form of f. This can lead to poor estimates because the model is not *flexible* enough.

A potential remedy is to consider adding more parameters, by choosing alternate forms for $\hat{f}$. Unfortunately if the model becomes too flexible it can lead to a very dangerous situation known as *overfitting*, which we will discuss at length in subsequent chapters. In essence, the model follows the noise too closely and not the signal.

Non-Parametric Models

The alternative approach is to consider a non-parametric form of $\hat{f}$. The benefit is that it can potentially fit a wider range of possible forms for f and is thus more flexible. Unfortunately non-parametric models suffer from the need to have an extensive amount of observational data points, often far more than in a parametric settings. In addition non-parametric methods are also prone to overfitting if not treated carefully, as described above.

Non-parametric models may seem like a natural choice for quantitative trading models as there is seemingly an abundance of (historical) data on which to apply the models. However, the methods are not always optimal. While the increased flexibility is attractive for modelling the non-linearities in stock market data it is very easy to overfit the data due to the poor signal/noise ratio found in financial time series.

Thus a "middle-ground" of considering models with some degree of flexibility is preferred. We will discuss such problems in the chapter on Optimisation later in the book.

9.1.3 Supervised and Unsupervised Learning

A distinction is often made in statistical machine learning between *supervised* and *unsupervised* methods. In this book we will almost exclusively be interested in supervised techniques, but unsupervised techniques are certainly applicable to financial markets.

A supervised model requires that for each predictor vector X_j there is an associated response Y_j. The "supervision" of the procedure occurs when the model for f is *trained* or *fit* to this particular data. For example, when fitting a linear regression model, the OLS algorithm is used to train it, ultimately producing an estimate $\hat{\beta}$ to the vector of regression coefficients, β.

In an unsupervised model there is no corresponding response Y_j for any particular predictor X_j. Hence there is nothing to "supervise" the training of the model. This is clearly a much more challenging environment for an algorithm to produce results as there is no form of "fitness function" with which to assess accuracy. Despite this setback, unsupervised techniques are extremely powerful. They are particularly useful in the realm of *clustering*.

A parametrised clustering model, when provided with a parameter specifying the number of clusters to identify, can often discern unanticipated relationships within data that might not otherwise have been easily determined. Such models are generally fall within the domain of *business analytics* and *consumer marketing optimisation* but they do have uses within finance, particularly in regards to assessing clustering within volatility, for instance.

This book will predominantly concentrate on supervised learning methods since there is a vast amount of historical data on which to train such models.

9.2 Techniques

Statistical machine learning is a vast interdisciplinary field, with many disparate research areas. The remainder of this chapter will consider the techniques most relevant to quantitative finance and algorithmic trading in particular.

9.2.1 Regression

Regression refers to a broad group of supervised machine learning techniques that provide both predictive and inferential capabilities. A significant portion of quantitative finance makes use of regression techniques and thus it is essential to be familiar with the process. Regression tries to model the relationship between a dependent variable (response) and a set of independent variables (predictors). In particular, the goal of regression is to ascertain the change in a response, when

one of the independent variables changes, under the assumption that the remaining independent variables are kept fixed.

The most widely known regression technique is *Linear Regression*, which assumes a linear relationship between the predictors and the response. Such a model leads to parameter estimates (usually denoted by the vector $\hat{\beta}$) for the linear response to each predictor. These parameters are estimated via a procedure known as *ordinary least squares* (OLS). Linear regression can be used both for prediction and inference.

In the former case a new value of the predictor can be added (without a corresponding response) in order to *predict* a new response value. For instance, consider a linear regression model used to predict the value of the S&P500 in the following day, from price data over the last five days. The model can be fitted using OLS across historical data. Then, when new market data arrive for the S&P500 it can be input into the model (as a predictor) to generate a predicted response for tomorrow's daily price. This can form the basis of a simplistic trading strategy.

In the latter case (inference) the strength of the relationship between the response and each predictor can be assessed in order to determine the subset of predictors that have an effect on the response. This is more useful when the goal is to understand *why* the response varies, such as in a marketing study or clinical trial. Inference is often less useful to those carrying out algorithmic trading, as the quality of the prediction is fundamentally more important than the underlying relationship. That being said, one should not solely rely on the "black box" approach due to the prevalence of over-fitting to noise in the data.

Other techniques include *Logistic Regression*, which is designed to predict a *categorical* response (such as "UP", "DOWN", "FLAT") as opposed to a *continuous* response (such as a stock market price). This technically makes it a *classification tool* (see below), but it is usually grouped under the banner of regression. A general statistical procedure known as *Maximum Likelihood Estimation* (MLE) is used to estimate the parameter values of a logistic regression.

9.2.2 Classification

Classification encompasses supervised machine learning techniques that aim to *classify* an *observation* (similar to a predictor) into a set of pre-defined categories, based on features associated with the observation. These categories can be un-ordered, e.g. "red", "yellow", "blue" or ordered, e.g. "low", "medium", "high". In the latter case such categorical groups are known as *ordinals*. Classification algorithms - *classifiers* - are widely used in quantitative finance, especially in the realm of market direction prediction. In this book we will be studying classifiers extensively.

Classifiers can be utilised in algorithmic trading to predict whether a particular time series will have positive or negative returns in subsequent (unknown) time periods. This is similar to a regression setting except that the actual value of the time series is not being predicted, rather its direction. Once again we are able to use continuous predictors, such as prior market prices as observations. We will consider both linear and non-linear classifiers, including Logistic Regression, Linear/Quadratic Discriminant Analysis, Support Vector Machines (SVM) and Artificial Neural Networks (ANN). *Note that some of the previous methods can actually be used in a regression setting also.*

9.2.3 Time Series Models

A key component in algorithmic trading is the treatment and prediction of *financial time series*. Our goal is generally to predict future values of time series based on prior values or external factors. Thus time series modelling can be seen as a mixed-subset of regression and classification. Time series models differ from non-temporal models because the models make deliberate use of the *temporal ordering* of the series. Thus the predictors are often based on past or current values, while the responses are often future values to be predicted.

There is a large literature on differing time series models. There are two broad families of time series models that interest us in algorithmic trading. The first set are the linear *autoregressive integrated moving average* (ARIMA) family of models, which are used to model the variations in the absolute value of a time series. The other family of time series are the *autoregressive*

conditional heteroskedasticity (ARCH) models, which are used to model the variance (i.e. the volatility) of time series over time. ARCH models use previous values (volatilities) of the time series to predict future values (volatilities). This is in contrast to *stochastic volatility* models, which utilise more than one stochastic time series (i.e. multiple stochastic differential equations) to model volatility.

All of the raw historical price time series are *discrete* in that they contain finite values. In the field of quantitative finance it is common to study *continuous* time series models. In particular, the famous *Geometric Brownian Motion*, the *Heston Stochastic Volatility* model and the *Ornstein-Uhlenbeck* model all represent continuous time series with differing forms of stochastic behaviour. We will utilise these time series models in subsequent chapters to attempt to characterise the behaviour of financial time series in order to exploit their properties to create viable trading strategies.

Chapter 10

Time Series Analysis

In this chapter we are going to consider statistical tests that will help us identify price series that possess trending or mean-reverting behaviour. If we can identify such series statistically then we can capitalise on this behaviour by forming momentum or mean-reverting trading strategies.

In later chapters we will use these statistical tests to help us identify candidate time series and then create algorithmic strategies around them.

10.1 Testing for Mean Reversion

One of the key quantitative trading concepts is **mean reversion**. This process refers to a time series that displays a tendency to revert to a historical mean value. Such a time series can be exploited to generate trading strategies as we enter the market when a price series is far from the mean under the expectation that the series will return to a mean value, whereby we exit the market for a profit. Mean-reverting strategies form a large component of the *statistical arbitrage* quant hedge funds. In later chapters we will create both intraday and interday strategies that exploit mean-reverting behaviour.

The basic idea when trying to ascertain if a time series is mean-reverting is to use a statistical test to see if it differs from the behaviour of a *random walk*. A random walk is a time series where the next directional movement is completely independent of any past movements - in essence the time series has no "memory" of where it has been. A mean-reverting time series, however, is different. The change in the value of the time series in the next time period is proportional to the current value. Specifically, it is proportional to the difference between the mean historical price and the current price.

Mathematically, such a (continuous) time series is referred to as an **Ornstein-Uhlenbeck** process. If we can show, statistically, that a price series behaves like an Ornstein-Uhlenbeck series then we can begin the process of forming a trading strategy around it. Thus the goal of this chapter is to outline the statistical tests necessary to identify mean reversion and then use Python libraries (in particular *statsmodels*) in order to implement these tests. In particular, we will study the concept of **stationarity** and how to test for it.

As stated above, a *continuous* mean-reverting time series can be represented by an Ornstein-Uhlenbeck stochastic differential equation:

$$dx_t = \theta(\mu - x_t)dt + \sigma dW_t \tag{10.1}$$

Where θ is the rate of reversion to the mean, μ is the mean value of the process, σ is the variance of the process and W_t is a Wiener Process or Brownian Motion.

This equation essentially states that the change of the price series in the next continuous time period is proportional to the difference between the mean price and the current price, with the addition of Gaussian noise.

We can use this equation to motivate the definition of the Augmented Dickey-Fuller Test, which we will now describe.

10.1.1 Augmented Dickey-Fuller (ADF) Test

The ADF test makes use of the fact that if a price series possesses mean reversion, then the next price level will be proportional to the current price level. Mathematically, the ADF is based on the idea of testing for the presence of a **unit root** in an **autoregressive** time series sample.

We can consider a model for a time series, known as a *linear lag model of order p*. This model says that the change in the value of the time series is proportional to a constant, the time itself and the previous p values of the time series, along with an error term:

$$\Delta y_t = \alpha + \beta t + \gamma y_{t-1} + \delta_1 \Delta y_{t-1} + \cdots + \delta_{p-1} \Delta y_{t-p+1} + \epsilon_t \tag{10.2}$$

Where α is a constant, β represents the coefficient of a temporal trend and $\Delta y_t = y(t) - y(t-1)$. The role of the ADF hypothesis test is to ascertain, statistically, whether $\gamma = 0$, which would indicate (with $\alpha = \beta = 0$) that the process is a random walk and thus non mean reverting. Hence we are testing for the *null hypothesis* that $\gamma = 0$.

If the hypothesis that $\gamma = 0$ can be rejected then the following movement of the price series is proportional to the current price and thus it is unlikely to be a random walk. This is what we mean by a "statistical test".

So how is the ADF test carried out?

- Calculate the *test statistic*, DF_τ, which is used in the decision to reject the null hypothesis
- Use the *distribution* of the test statistic (calculated by Dickey and Fuller), along with the critical values, in order to decide whether to reject the null hypothesis

Let's begin by calculating the test statistic (DF_τ). This is given by the *sample* proportionality constant $\hat{\gamma}$ divided by the *standard error* of the sample proportionality constant:

$$DF_\tau = \frac{\hat{\gamma}}{SE(\hat{\gamma})} \tag{10.3}$$

Now that we have the test statistic, we can use the distribution of the test statistic calculated by Dickey and Fuller to determine the rejection of the null hypothesis for any chosen percentage critical value. The test statistic is a negative number and thus in order to be significant beyond the critical values, the number must be smaller (i.e. more negative) than these values.

A key practical issue for traders is that any constant long-term drift in a price is of a much smaller magnitude than any short-term fluctuations and so the drift is often assumed to be zero ($\beta = 0$) for the linear lag model described above.

Since we are considering a lag model of order p, we need to actually set p to a particular value. It is usually sufficient, for trading research, to set $p = 1$ to allow us to reject the null hypothesis. However, note that this technically introduces a parameter into a trading model based on the ADF.

To calculate the Augmented Dickey-Fuller test we can make use of the pandas and statsmodels libraries. The former provides us with a straightforward method of obtaining Open-High-Low-Close-Volume (OHLCV) data from Yahoo Finance, while the latter wraps the ADF test in a easy to call function. This prevents us from having to calculate the test statistic manually, which saves us time.

We will carry out the ADF test on a sample price series of Amazon stock, from 1st January 2000 to 1st January 2015.

Here is the Python code to carry out the test:

```
from __future__ import print_function

# Import the Time Series library
import statsmodels.tsa.stattools as ts

# Import Datetime and the Pandas DataReader
```

```
from datetime import datetime
import pandas.io.data as web

# Download the Amazon OHLCV data from 1/1/2000 to 1/1/2015
amzn = web.DataReader("AMZN", "yahoo", datetime(2000,1,1), datetime(2015,1,1))

# Output the results of the Augmented Dickey-Fuller test for Amazon
# with a lag order value of 1
ts.adfuller(amzn['Adj Close'], 1)
```

Here is the output of the Augmented Dickey-Fuller test for Amazon over the period. The first value is the calculated test-statistic, while the second value is the *p-value*. The fourth is the number of data points in the sample. The fifth value, the dictionary, contains the critical values of the test-statistic at the 1, 5 and 10 percent values respectively.

```
(0.049177575166452235,
 0.96241494632563063,
 1,
 3771,
 {'1%': -3.4320852842548395,
  '10%': -2.5671781529820348,
  '5%': -2.8623067530084247},
 19576.116041473877)
```

Since the calculated value of the test statistic is larger than any of the critical values at the 1, 5 or 10 percent levels, we cannot reject the null hypothesis of $\gamma = 0$ and thus we are unlikely to have found a mean reverting time series. This is in line with our tuition as most equities behave akin to Geometric Brownian Motion (GBM), i.e. a random walk.

This concludes how we utilise the ADF test. However, there are alternative methods for detecting mean-reversion, particularly via the concept of **stationarity**, which we will now discuss.

10.2 Testing for Stationarity

A time series (or stochastic process) is defined to be **strongly stationary** if its *joint probability distribution* is invariant under translations in time or space. In particular, and of key importance for traders, the mean and variance of the process do not change over time or space and they each do not follow a trend.

A critical feature of stationary price series is that the prices within the series diffuse from their initial value at a rate slower than that of a GBM. By measuring the rate of this diffusive behaviour we can identify the nature of the time series and thus detect whether it is mean-reverting.

We will now outline a calculation, namely the Hurst Exponent, which helps us to characterise the stationarity of a time series.

10.2.1 Hurst Exponent

The goal of the Hurst Exponent is to provide us with a scalar value that will help us to identify (within the limits of statistical estimation) whether a series is mean reverting, random walking or trending.

The idea behind the Hurst Exponent calculation is that we can use the variance of a log price series to assess the rate of diffusive behaviour. For an arbitrary time lag τ, the variance of τ is given by:

$$\mathrm{Var}(\tau) = \langle |\log(t+\tau) - \log(t)|^2 \rangle \tag{10.4}$$

Where the brackets $\langle$ and $\rangle$ refer to the average over all values of t.

The idea is to compare the rate of diffusion to that of a GBM. In the case of a GBM, at large times (i.e. when τ is large) the variance of τ is proportional to τ:

$$\langle |\log(t+\tau) - \log(t)|^2 \rangle \sim \tau \tag{10.5}$$

If we find behaviour that differs from this relation, then we have identified either a trending or a mean-reverting series. The key insight is that if any sequential price movements possess non-zero correlation (known as autocorrelation) then the above relationship is not valid. Instead it can be modified to include an exponent value "$2H$", which gives us the Hurst Exponent value H:

$$\langle |\log(t+\tau) - \log(t)|^2 \rangle \sim \tau^{2H} \tag{10.6}$$

Thus it can be seen that if $H = 0.5$ we have a GBM, since it simply becomes the previous relation. However if $H \neq 0.5$ then we have trending or mean-reverting behaviour. In particular:

- $H < 0.5$ - The time series is mean reverting
- $H = 0.5$ - The time series is a Geometric Brownian Motion
- $H > 0.5$ - The time series is trending

In addition to characterisation of the time series the Hurst Exponent also describes the extent to which a series behaves in the manner categorised. For instance, a value of H near 0 is a highly mean reverting series, while for H near 1 the series is strongly trending.

To calculate the Hurst Exponent for the Amazon price series, as utilised above in the explanation of the ADF, we can use the following Python code:

```
from __future__ import print_function

from numpy import cumsum, log, polyfit, sqrt, std, subtract
from numpy.random import randn

def hurst(ts):
    """Returns the Hurst Exponent of the time series vector ts"""
    # Create the range of lag values
    lags = range(2, 100)

    # Calculate the array of the variances of the lagged differences
    tau = [sqrt(std(subtract(ts[lag:], ts[:-lag]))) for lag in lags]

    # Use a linear fit to estimate the Hurst Exponent
    poly = polyfit(log(lags), log(tau), 1)

    # Return the Hurst exponent from the polyfit output
    return poly[0]*2.0

# Create a Gometric Brownian Motion, Mean-Reverting and Trending Series
gbm = log(cumsum(randn(100000))+1000)
mr = log(randn(100000)+1000)
tr = log(cumsum(randn(100000)+1)+1000)

# Output the Hurst Exponent for each of the above series
# and the price of Amazon (the Adjusted Close price) for
# the ADF test given above in the article
print("Hurst(GBM):   %s" % hurst(gbm))
print("Hurst(MR):    %s" % hurst(mr))
print("Hurst(TR):    %s" % hurst(tr))
```

```
# Assuming you have run the above code to obtain 'amzn'!
print("Hurst(AMZN):  %s" % hurst(amzn['Adj Close']))
```

The output from the Hurst Exponent Python code is given below:

```
Hurst(GBM):   0.502051910931
Hurst(MR):    0.000166110248967
Hurst(TR):    0.957701001252
Hurst(AMZN):  0.454337476553
```

From this output we can see that the GBM possesses a Hurst Exponent, H, that is almost exactly 0.5. The mean reverting series has H almost equal to zero, while the trending series has H close to 1.

Interestingly, Amazon has H also close to 0.5 indicating that it is similar to a GBM, at least for the sample period we're making use of!

10.3 Cointegration

It is actually very difficult to find a tradable asset that possesses mean-reverting behaviour. Equities broadly behave like GBMs and hence render the mean-reverting trade strategies relatively useless. However, there is nothing stopping us from creating a *portfolio* of price series that is stationary. Hence we can apply mean-reverting trading strategies to the portfolio.

The simplest form of mean-reverting trade strategies is the classic "pairs trade", which usually involves a dollar-neutral long-short pair of equities. The theory goes that two companies in the same sector are likely to be exposed to similar market factors, which affect their businesses. Occasionally their relative stock prices will diverge due to certain events, but will revert to the long-running mean.

Let's consider two energy sector equities Approach Resources Inc given by the ticker AREX and Whiting Petroleum Corp given by the ticker WLL. Both are exposed to similar market conditions and thus will likely have a stationary pairs relationship. We are now going to create some plots, using pandas and the Matplotlib libraries to demonstrate the cointegrating nature of AREX and WLL. The first plot (Figure 10.1) displays their respective price histories for the period Jan 1st 2012 to Jan 1st 2013.

If we create a scatter plot of their prices, we see that the relationship is broadly linear (see Figure 10.2) for this period.

The pairs trade essentially works by using a linear model for a relationship between the two stock prices:

$$y(t) = \beta x(t) + \epsilon(t) \tag{10.7}$$

Where $y(t)$ is the price of AREX stock and $x(t)$ is the price of WLL stock, both on day t.

If we plot the residuals $\epsilon(t) = y(t) - \beta x(t)$ (for a particular value of β that we will determine below) we create a new time series that, at first glance, looks relatively stationary. This is given in Figure 10.3.

We will describe the code for each of these plots below.

10.3.1 Cointegrated Augmented Dickey-Fuller Test

In order to statistically confirm whether this series is mean-reverting we could use one of the tests we described above, namely the Augmented Dickey-Fuller Test or the Hurst Exponent. However, neither of these tests will actually help us determine β, the hedging ratio needed to form the linear combination, they will only tell us whether, for a particular β, the linear combination is stationary.

This is where the Cointegrated Augmented Dickey-Fuller (CADF) test comes in. It determines the optimal hedge ratio by performing a linear regression against the two time series and then tests for stationarity under the linear combination.

Figure 10.1: Time series plots of AREX and WLL

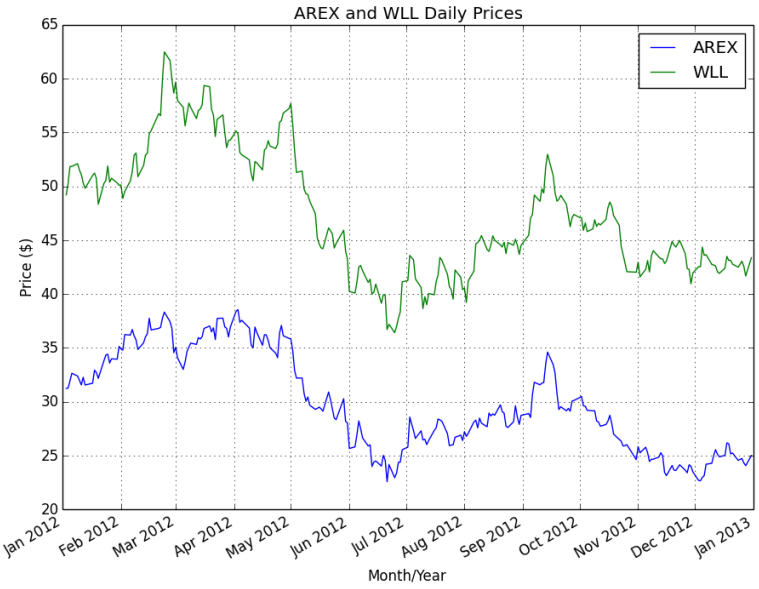
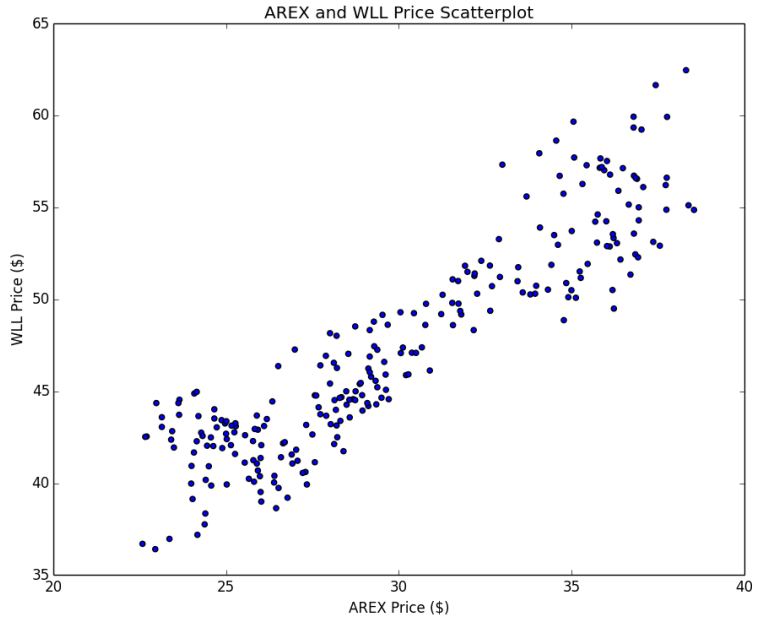


Figure 10.2: Scatter plot of AREX and WLL prices

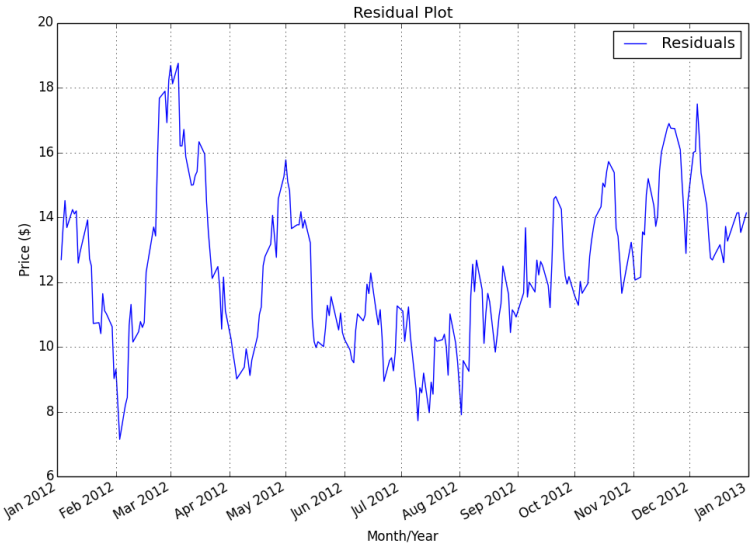


Python Implementation

We will now use Python libraries to test for a cointegrating relationship between AREX and WLL for the period of Jan 1st 2012 to Jan 1st 2013. We will use Yahoo Finance for the data source and Statsmodels to carry out the ADF test, as above.

The first task is to create a new file, cadf.py, and import the necessary libraries. The code makes use of NumPy, Matplotlib, Pandas and Statsmodels. In order to correctly label the axes

Figure 10.3: Residual plot of AREX and WLL linear combination



and download data from Yahoo Finance via pandas, we import the matplotlib.dates module and the pandas.io.data module. We also make use of the Ordinary Least Squares (OLS) function from pandas:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# cadf.py

import datetime
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import pandas as pd
import pandas.io.data as web
import pprint
import statsmodels.tsa.stattools as ts

from pandas.stats.api import ols
```

The first function, plot_price_series, takes a pandas DataFrame as input, with to columns given by the placeholder strings "ts1" and "ts2". These will be our pairs equities. The function simply plots the two price series on the same chart. This allows us to visually inspect whether any cointegration may be likely.

We use the Matplotlib dates module to obtain the months from the datetime objects. Then we create a figure and a set of axes on which to apply the labelling/plotting. Finally, we plot the figure:

```
# cadf.py

def plot_price_series(df, ts1, ts2):
    months = mdates.MonthLocator()  # every month
    fig, ax = plt.subplots()
    ax.plot(df.index, df[ts1], label=ts1)
```

```
    ax.plot(df.index, df[ts2], label=ts2)
    ax.xaxis.set_major_locator(months)
    ax.xaxis.set_major_formatter(mdates.DateFormatter('%b %Y'))
    ax.set_xlim(datetime.datetime(2012, 1, 1), datetime.datetime(2013, 1, 1))
    ax.grid(True)
    fig.autofmt_xdate()

    plt.xlabel('Month/Year')
    plt.ylabel('Price ($)')
    plt.title('%s and %s Daily Prices' % (ts1, ts2))
    plt.legend()
    plt.show()
```

The second function, plot_scatter_series, plots a scatter plot of the two prices. This allows us to visually inspect whether a linear relationship exists between the two series and thus whether it is a good candidate for the OLS procedure and subsequent ADF test:

```
# cadf.py

def plot_scatter_series(df, ts1, ts2):
    plt.xlabel('%s Price ($)' % ts1)
    plt.ylabel('%s Price ($)' % ts2)
    plt.title('%s and %s Price Scatterplot' % (ts1, ts2))
    plt.scatter(df[ts1], df[ts2])
    plt.show()
```

The third function, plot_residuals, is designed to plot the residual values from the fitted linear model of the two price series. This function requires that the pandas DataFrame has a "res" column, representing the residual prices:

```
# cadf.py

def plot_residuals(df):
    months = mdates.MonthLocator()  # every month
    fig, ax = plt.subplots()
    ax.plot(df.index, df["res"], label="Residuals")
    ax.xaxis.set_major_locator(months)
    ax.xaxis.set_major_formatter(mdates.DateFormatter('%b %Y'))
    ax.set_xlim(datetime.datetime(2012, 1, 1), datetime.datetime(2013, 1, 1))
    ax.grid(True)
    fig.autofmt_xdate()

    plt.xlabel('Month/Year')
    plt.ylabel('Price ($)')
    plt.title('Residual Plot')
    plt.legend()

    plt.plot(df["res"])
    plt.show()
```

Finally, the procedure is wrapped up in a __main__ function. The first task is to download the OHLCV data for both AREX and WLL from Yahoo Finance. Then we create a separate DataFrame, df, using the same index as the AREX frame to store both of the adjusted closing price values. We then plot the price series and the scatter plot.

After the plots are complete the residuals are calculated by calling the pandas ols function on the WLL and AREX series. This allows us to calculate the β hedge ratio. The hedge ratio is then used to create a "res" column via the formation of the linear combination of both WLL and AREX.

Finally the residuals are plotted and the ADF test is carried out on the calculated residuals. We then print the results of the ADF test:

```
# cadf.py

if __name__ == "__main__":
    start = datetime.datetime(2012, 1, 1)
    end = datetime.datetime(2013, 1, 1)

    arex = web.DataReader("AREX", "yahoo", start, end)
    wll = web.DataReader("WLL", "yahoo", start, end)

    df = pd.DataFrame(index=arex.index)
    df["AREX"] = arex["Adj Close"]
    df["WLL"] = wll["Adj Close"]

    # Plot the two time series
    plot_price_series(df, "AREX", "WLL")

    # Display a scatter plot of the two time series
    plot_scatter_series(df, "AREX", "WLL")

    # Calculate optimal hedge ratio "beta"
    res = ols(y=df['WLL'], x=df["AREX"])
    beta_hr = res.beta.x

    # Calculate the residuals of the linear combination
    df["res"] = df["WLL"] - beta_hr*df["AREX"]

    # Plot the residuals
    plot_residuals(df)

    # Calculate and output the CADF test on the residuals
    cadf = ts.adfuller(df["res"])
    pprint.pprint(cadf)
```

The output of the code (along with the Matplotlib plots) is as follows:

```
(-2.9607012342275936,
 0.038730981052330332,
 0,
 249,
 {'1%': -3.4568881317725864,
  '10%': -2.5729936189738876,
  '5%': -2.8732185133016057},
 601.96849256295991)
```

It can be seen that the calculated test statistic of -2.96 is smaller than the 5% critical value of -2.87, which means that we can reject the null hypothesis that there isn't a cointegrating relationship at the 5% level. Hence we can conclude, with a reasonable degree of certainty, that AREX and WLL possess a cointegrating relationship, at least for the time period sample considered.

We will use this pair in subsequent chapters to create an actual trading strategy using an implemented event-driven backtesting system.

10.4 Why Statistical Testing?

Fundamentally, as far as algorithmic trading is concerned, the statistical tests outlined above are only as useful as the profits they generate when applied to trading strategies. Thus, surely it makes sense to simply evaluate performance at the strategy level, as opposed to the price/time series level? Why go to the trouble of calculating all of the above metrics when we can simply use trade level analysis, risk/reward measures and drawdown evaluations?

Firstly, any implemented trading strategy based on a time series statistical measure will have a far larger sample to work with. This is simply because when calculating these statistical tests, we are making use of each *bar* of information, rather than each *trade*. There will be far less round-trip trades than bars and hence the statistical significance of any trade-level metrics will be far smaller.

Secondly, any strategy we implement will depend upon certain parameters, such as look-back periods for rolling measures or z-score measures for entering/exiting a trade in a mean-reversion setting. Hence strategy level metrics are only appropriate *for these parameters*, while the statistical tests are valid for the underlying time series sample.

In practice we want to calculate both sets of statistics. Python, via the statsmodels and pandas libraries, make this extremely straightforward. The additional effort is actually rather minimal!

Chapter 11

Forecasting

In this chapter we will create a statistically robust process for forecasting financial time series. These forecasts will form the basis for further automated trading strategies. We will expand on the topic of Statistical Learning discussed in the previous chapters and use a group of *classification algorithms* to help us predict market direction of financial time series.

Within this chapter we will be making use of **Scikit-Learn**, a statistical machine learning library for Python. Scikit-learn contains "ready-made" implementations of many machine learning techniques. Not only does this save us a great deal of time in implementing our trading algorithms, but it minimises the risk of bugs introduced by our own code. It also allows additional verification against machine learning libraries written in other packages such as R or C++. This gives us a great deal of confidence if we need to create our own custom implementation, for reasons of execution speed, say.

We will begin by discussing ways of measuring forecaster performance for the particular case of machine learning techniques used. Then we will consider the predictive factors that can be used in forecasting techniques and how to choose good factors. Then we will consider various supervised classifier algorithms. Finally, we will attempt to forecast the daily direction of the S&P500, which will later form the basis of an algorithmic trading strategy.

11.1 Measuring Forecasting Accuracy

Before we discuss choices of predictor and specific classification algorithms we must discuss their performance characteristics and how to evaluate them. The particular class of methods that we are interested in involves *binary supervised classification*. That is, we will attempt to predict whether the percentage return for a particular future day is positive or negative (i.e. whether our financial asset has risen or dropped in price).

In a production forecaster, using a regression-type technique, we would be very concerned with the *magnitude* of this prediction and the deviations of the prediction from the actual value.

To assess the performance of these classifiers we can make use of the following two measures, namely the **Hit-Rate** and **Confusion Matrix**.

11.1.1 Hit Rate

The simplest question that we could ask of our supervised classifier is *"How many times did we predict the correct direction, as a percentage of all predictions?"*. This motivates the definition of the *training hit rate* is given by the following formula[9]:

$$\frac{1}{n}\sum_{j=1}^{n} I(y_j = \hat{y_j}) \tag{11.1}$$

Where $\hat{y_j}$ is the prediction (up or down) for the jth time period (e.g. a day) using a particular classifier. $I(y_j = \hat{y_j})$ is the *indicator function* and is equal to 1 if $y_j = \hat{y_j}$ and 0 if $y_j \neq \hat{y_j}$.

Hence the hit rate provides a percentage value as to the number of times a classifier correctly predicted the up or down direction.

Scikit-Learn provides a method to calculate the hit rate for us as part of the classification/-training process.

11.1.2 Confusion Matrix

The *confusion matrix* (or *contingency table*) is the next logical step after calculating the hit rate. It is motivated by asking "How many times did we predict up correctly and how many times did we predict down correctly? Did they differ substantially?".

For instance, it might turn out that a particular algorithm is consistently more accurate at predicting "down days". This motivates a strategy that emphasises shorting of a financial asset to increase profitability.

A confusion matrix characterises this idea by determining the *false positive rate* (known statistically as a Type I error) and *false negative rate* (known statistically as a Type II error) for a supervised classifier. For the case of binary classification (up or down) we will have a 2x2 matrix:

$$\begin{pmatrix} U_T & U_F \\ D_F & D_T \end{pmatrix}$$

Where U_T represents correctly classified up periods, U_F represents incorrectly classified up periods (i.e. classified as down), D_F represents incorrectly classified down periods (i.e. classified as up) and D_T represents correctly classified down periods.

In addition to the hit rate, Scikit-Learn provides a method to calculate the confusion matrix for us as part of the classification/training process.

11.2 Factor Choice

One of the most crucial aspects of asset price forecasting is choosing the factors used as predictors. There are a staggering number of potential factors to choose and this can seem overwhelming to an individual unfamiliar with financial forecasting. However, even simple machine learning techniques will produce relatively good results when used with well-chosen factors. Note that the converse is not often the case. "Throwing an algorithm at a problem" will usually lead to poor forecasting accuracy.

Factor choice is carried out by trying to determine the fundamental drivers of asset movement. In the case of the S&P500 it is clear that the 500 constituents, in a weighted manner, will be fundamental drivers of the price, by definition! Clearly we would know the exact price of the S&P500 series if we knew the instantaneous value of its constituents, but is there any predictive power in using the prior history of returns for each constituent in predicting the series itself?

Alternatively, could we consider exchange rates with countries that carry out a lot of trade with the US as drivers of the price? We could even consider more fundamental economic and corporate factors such as interest rates, inflation, quarterly earnings.

The accuracy of the forecaster will in large part be due to the skill of the modeller in determining the right factors prior to carrying out model fitting.

11.2.1 Lagged Price Factors and Volume

The first type of factor that is often considered in forecasting a time series are prior historical values of the time series itself. Thus a set of p factors could be easily obtained by creating p lags of the time series close price. Consider a daily time series. For each particular current day k, the factors would be the historical daily values at time periods $k-1$, $k-2$, ... , $k-p$.

In addition to the price series itself we can also incorporate traded volume as an indicator, since it is provided when using OHLCV data (as is obtained from Yahoo Finance, Google Finance or Quandl for instance). Thus we can create a $p+1$-dimensional feature vector for each day of the time series, which incorporates the p time lags and the volume series. This naturally leads

to a set of pairs (X_k, y_k) representing the $p+1$-dimensional feature vector X_k at day k and the actual current closing price on day k, y_k. This is all we need to begin a supervised classification exercise.

Below we will consider such a lagged time series for the S&P500 and apply multiple machine learning techniques to see if we can forecast its direction.

11.2.2 External Factors

While lagged time series and volume information are a good starting point for time series analysis, we are far from restricted to such data. There are a vast amount of macroeconomic time series and asset prices series on which to consider forecasts. For instance we may wish to provide a long-term forecast of commodities prices based on weather patterns, or ascertain foreign exchange price direction movements via international interest rate movements.

If such a relationship between series can be ascertained and shown to be statistically significant, then we are at the point of being able to consider a robust trading model. We won't dwell on such relationships too much here, as our goal is to introduce the idea of modelling and machine learning techniques. It is easy enough to form hypotheses about economic relationships and obtain the time series data either from a repository such as Quandl, or directly from government statistics websites.

11.3 Classification Models

The field of machine learning is vast and there are many models to choose from, particularly in the realm of supervised classification. New models are being introduced on a monthly basis through the academic literature. It would be impractical to provide an exhaustive list of supervised classifiers in this chapter, rather we will consider some of the more popular techniques from the field.

11.3.1 Logistic Regression

The first technique we will consider is **logistic regression** (LR). In our case we are going to use logistic regression to measures the relationship between a *binary categorical dependent variable* (i.e. "up" or "down" periods) and multiple independent *continuous variables*, such as the lagged percentage returns of a financial asset.

The logistic regression model provides the *probability* that a particular subsequent time period will be categorised as "up" or "down". Thus the model introduces a *parameter*, namely the probability threshold for classifying whether a subsequent time period is "up" or "down". Below, we will take this threshold to be 50% (i.e. 0.5), but it can certainly be modified to produce alternative predictions.

Logistic regression is based on the logistic formula to model the probability of obtaining an "up" day ($Y = U$) based on the continuous factors.

In this case, consider the situation where we are interested in predicting the subsequent time period from the previous two lagged returns, which we will denote by (L_1, L_2). The formula below gives the probability for having an up day, given that we have observed the returns on the previous time periods, L_1 and L_2:

$$p(Y = U|L_1, L_2) = \frac{e^{\beta_0+\beta_1 L_1+\beta_2 L_2}}{1 + e^{\beta_0+\beta_1 L_1+\beta_2 L_2}} \tag{11.2}$$

The logistic function is used instead of a linear function (i.e. in linear regression) because it provides a probability between $[0, 1]$ for all values of L_1 and L_2. In a linear regression setting it is possible to obtain negative probabilities for these continuous variables so we need another function.

To fit the model (i.e. estimate the β_i coefficients) the **maximum likelihood method** is used. Fortunately for us the implementation of the fitting and prediction of the logistic regression

model is already handled by the Scikit-Learn library. The technique will be outlined below when we attempt to forecast the direction of the S&P500.

11.3.2 Discriminant Analysis

Discriminant analysis is an alternative statistical technique to logistic regression. While logistic regression is less restrictive in its assumptions than discriminant analysis, it can give greater predictive performance if the more restrictive assumptions are met.

We will now consider a linear method and a non-linear method of discriminant analysis.

Linear Discriminant Analysis

In logistic regression we model the probability of seeing an "up" time period, given the previous two lagged returns ($P(Y = U|L_1, L_2)$) as a conditional distribution of the response Y given the predictors L_i, using a logistic function.

In Linear Discriminant Analysis (LDA) the distribution of the L_i variables are modelled separately, given Y, and $P(Y = U|L_1, L_2)$ is obtained via Bayes' Theorem.

Essentially, LDA results from assuming that predictors are drawn from a multivariate Gaussian distribution. After calculating estimates for the parameters of this distribution, the parameters can be inserted into Bayes' Theorem in order to make predictions about which class an observation belongs to.

One important mathematical assumption of LDA is that all *classes* (e.g. "up" and "down") share the same *covariance matrix*.

I won't dwell on the formulae for estimating the distribution or *posterior probabilities* that are needed to make predictions, as once again scikit-learn handles this for us.

Quadratic Discriminant Analysis

Quadratic Discriminant Analysis (QDA) is closely reed to LDA. The significant difference is that each class can now possess its own covariance matrix.

QDA generally performs better when the decision boundaries are non-linear. LDA generally performs better when there are fewer training observations (i.e. when needing to reduce variance). QDA, on the other hand, performs well when the training set is large (i.e. variance is of less concern). The use of one or the other ultimately comes down to the bias-variance trade-off.

As with LR and LDA, Scikit-Learn takes care of the QDA implementation so we only need to provide it with training/test data for parameter estimation and prediction.

11.3.3 Support Vector Machines

In order to motivate Support Vector Machines (SVM) we need to consider the idea of a classifier that separates different classes via a linear separating boundary. If such a straightforward separation existed then we could create a supervised classifier solely based on deciding whether new features lie above or below this linear classifying plane. In reality, such separations rarely exist in quantitative trading situations and as such we need to consider *soft margin classifiers* or **Support Vector Classifiers (SVC)**.

SVCs work by attempting to locate a linear separation boundary in feature space that correctly classifies most, but not all, of the training observations by creating an optimal separation boundary between the two classes. Sometimes such a boundary is quite effective if the class separation is mostly linear. However, other times such separations are not possible and it is necessary to utilise other techniques.

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)**. The major advantage of SVMs is that they allow a non-linear enlargening of the feature space to include significant non-linearity, while still retaining a significant computational efficiency, using a process known as the "kernel trick".

SVMs allow non-linear decision boundaries via many different choices of "kernel". In particular, instead of using a fully linear separating boundary as in the SVC, we can use quadratic

polynomials, higher-order polynomials or even radial kernals to describe non-linear boundaries. This gives us a significant degree of flexibility, at the ever-present expense of bias in our estimates.

We will use the SVM below to try and partition feature space (i.e. the lagged price factors and volume) via a non-linear boundary that allows us to make reasonable predictions about whether the subsequent day will be an up move or a down move.

11.3.4 Decision Trees and Random Forests

Decision trees are a supervised classification technique that utilise a tree structure to partition the feature space into recursive subsets via a "decision" at each node of the tree.

For instance one could ask if yesterday's price was above or below a certain threshold, which immediately partitions the feature space into two subsets. For each of the two subsets one could then ask whether the volume was above or below a threshold, thus creating four separate subsets. This process continues until there is no more predictive power to be gained by partitioning.

A decision tree provides a naturally interpretable classification mechanism when compared to the more "black box" opaque approaches of the SVM or discriminant analysers and hence are a popular supervised classification technique.

As computational power has increased, a new method of attacking the problem of classification has emerged, that of *ensemble learning*. The basic idea is simple. Create a large quantity of classifiers from the same base model and train them all with varying parameters. Then combine the results of the prediction in an average to hopefully obtain a prediction accuracy that is greater than that brought on by any of the individual constituents.

One of the most widespread ensemble methods is that of a **Random Forest**, which takes multiple decision tree learners (usually tens of thousands or more) and combines the predictions. Such ensembles can often perform extremely well. Scikit-Learn handily comes with a RandomForestClassifier (RFC) class in its `ensemble` module.

The two main parameters of interest for the RFC are `n_estimators`, which describes how many decision trees to create, and `n_jobs`, which describes how many processing cores to spread the calculations over. We will discuss these settings in the implementation section below.

11.3.5 Principal Components Analysis

All of the above techniques outlined above belong in the *supervised classification* domain. An alternative approach to performing classification is to not supervise the training procedure and instead allow an algorithm to ascertain "features" on its own. Such methods are known as *unsupervised learning* techniques.

Common use cases for unsupervised techniques include reducing the number of dimensions of a problem to only those considered important, discovering topics among large quantities of text documents or discovering features that may provide predictive power in time series analysis.

Of interest to us in this section is the concept of *dimensionality reduction*, which aims to identify the most important components in a set of factors that provide the most predictability. In particular we are going to utilise an unsupervised technique known as **Principal Components Analysis (PCA)** to reduce the size of the feature space prior to use in our supervised classifiers.

The basic idea of a PCA is to transform a set of possibly correlated variables (such as with time series autocorrelation) into a set of linearly uncorrelated variables known as the *principal components*. Such principal components are ordered according to the amount of variance they describe, in an orthogonal manner. Thus if we have a very high-dimensional feature space (10+ features), then we could reduce the feature space via PCA to perhaps 2 or 3 principal components that provide nearly all of the variability in the data, thus leading to a more robust supervised classifier model when used on this reduced dataset.

11.3.6 Which Forecaster?

In quantitative financial situations where there is an abundance of training data one should consider using a model such as a Support Vector Machine (SVM). However, SVMs suffer from lack of interpretibility. This is not the case with Decision Trees and Random Forest ensembles.

The latter are often used to preserve interpretability, something which "black box" classifiers such as SVM do not provide.

Ultimately when the data is so extensive (e.g. tick data) it will matter very little which classifier is ultimately used. At this stage other factors arise such as computational efficiency and scalability of the algorithm. The broad rule-of-thumb is that a doubling of training data will provide a linear increase in performance, but as the data size becomes substantial, this improvement reduces to a sublinear increase in performance.

The underlying statistical and mathematical theory for supervised classifiers is quite involved, but the basic intuition on each classifier is straightforward to understand. Also - note that each of the following classifiers will have a different set of assumptions as to when they will work best, so if you find a classifier performing poorly, it may be because the data-set being used violates one of the assumptions used to generate the theory.

Naive Bayes Classifier

While we haven't considered a Naive Bayes Classifier in our examples above, I wanted to include a discussion on it for completeness. Naive Bayes (specifically Multinomial Naive Bayes - MNB) is good to use when a limited data set exists. This is because it is a high-bias classifier. The major assumption of the MNB classifier is that of conditional independence. Essentially this means that it is unable to discern interactions between individual features, unless they are specifically added as extra features.

For example, consider a document classification situation, which appears in financial settings when trying to carry out sentiment analysis. The MNB could learn that individual words such as "cat" and "dog" could respectively refer to documents pertaining to cats and dogs, but the phrase "cats and dogs" (British slang for raining heavily) would not be considered to be meteorological by the classifier! The remedy to this would be to treat "cats and dogs" as an extra feature, specifically, and then associate that to a meteorological category.

Logistic Regression

Logistic regression provides some advantages over a Naive Bayes model in that there is less concern about correlation among features and, by the nature of the model, there is a probabilistic interpretation to the results. This is best suited to an environment where it is necessary to use thresholds. For instance, we might wish to place a threshold of 80% (say) on an "up" or "down" result in order for it to be correctly selected, as opposed to picking the highest probability category. In the latter case, the prediction for "up" could be 51% and the prediction for "down" could be 49%. Setting the category to "up" is not a very strong prediction in this instance.

Decision Tree and Random Forests

Decision trees (DT) partition a space into a hierarchy of boolean choices that lead to a categorisation or grouping based on the the respective decisions. This makes them highly interpretable (assuming a "reasonable" number of decisions/nodes in the tree!). DT have many benefits, including the ability to handle interactions between features as well as being *non-parametric*.

They are also useful in cases where it is not straightforward (or impossible) to linearly separate data into classes (which is a condition required of support vector machines). The disadvantage of using individual decision trees is that they are prone to overfitting (high variance). This problem is solved using a random forest. Random forests are actually some of the "best" classifiers when used in machine learning competitions, so they should always be considered.

Support Vector Machine

Support Vector Machines (SVM), while possessing a complicated fitting procedure, are actually relatively straightforward to understand. Linear SVMs essentially try to partition a space using linear separation boundaries, into multiple distinct groups. For certain types of data this can work extremely well and leads to good predictions. However, a lot of data is not linearly-separable and so linear SVMs can perform poorly here.

The solution is to modify the kernel used by the SVM, which has the effect of allowing non-linear decision boundaries. Thus they are quite flexible models. However, the right SVM boundary needs to be chosen for the best results. SVM are especially good in text classification problems with high dimensionality. They are disadvantaged by their computational complexity, difficulty of tuning and the fact the the fitted model is difficult to interpret.

11.4 Forecasting Stock Index Movement

The S&P500 is a weighted index of the 500 largest publicly traded companies by market capitalisation in the US stock market. It is often utilised as an equities benchmark. Many derivative products exist in order to allow speculation or hedging on the index. In particular, the S&P500 E-Mini Index Futures Contract is an extremely liquid means of trading the index.

In this section we are going to use a set of classifiers to predict the direction of the closing price at day k based solely on price information known at day $k-1$. An upward directional move means that the closing price at k is higher than the price at $k-1$, while a downward move implies a closing price at k lower than at $k-1$.

If we can determine the direction of movement in a manner that significantly exceeds a 50% hit rate, with low error and a good statistical significance, then we are on the road to forming a basic systematic trading strategy based on our forecasts.

11.4.1 Python Implementations

For the implementation of these forecasters we will make use of NumPy, Pandas and Scikit-Learn, which were installed in the previous chapters.

The first step is to import the relevant modules and libraries. We're going to import the LogisticRegression, LDA, QDA, LinearSVC (a linear Support Vector Machine), SVC (a non-linear Support Vector Machine) and RandomForest classifiers for this forecast:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# forecast.py

from __future__ import print_function

import datetime
import numpy as np
import pandas as pd
import sklearn

from pandas.io.data import DataReader
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.lda import LDA
from sklearn.metrics import confusion_matrix
from sklearn.qda import QDA
from sklearn.svm import LinearSVC, SVC
```

Now that the libraries are imported, we need to create a pandas DataFrame that contains the lagged percentage returns for a prior number of days (defaulting to five). `create_lagged_series` will take a stock symbol (as recognised by Yahoo Finance) and create a lagged DataFrame across the period specified. The code is well commented so it should be straightforward to see what is going on:

```
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 range(0, lags):
        tslag["Lag%s" % str(i+1)] = ts["Adj Close"].shift(i+1)

    # Create the returns DataFrame
    tsret = pd.DataFrame(index=tslag.index)
    tsret["Volume"] = tslag["Volume"]
    tsret["Today"] = tslag["Today"].pct_change()*100.0

    # If any of the values of percentage returns equal zero, set them to
    # a small number (stops issues with QDA model in Scikit-Learn)
    for i,x in enumerate(tsret["Today"]):
        if (abs(x) < 0.0001):
            tsret["Today"][i] = 0.0001

    # Create the lagged percentage returns columns
    for i in range(0, lags):
        tsret["Lag%s" % str(i+1)] = \
        tslag["Lag%s" % str(i+1)].pct_change()*100.0

    # Create the "Direction" column (+1 or -1) indicating an up/down day
    tsret["Direction"] = np.sign(tsret["Today"])
    tsret = tsret[tsret.index >= start_date]

    return tsret
```

We tie the classification procedure together with a `__main__` function. In this instance we're going to attempt to forecast the US stock market direction in 2005, using returns data from 2001 to 2004.

Firstly we create a lagged series of the S&P500 using five lags. The series also includes trading volume. However, we are going to restrict the predictor set to use only the first two lags. Thus we are implicitly stating to the classifier that the further lags are of less predictive value. *As an aside, this effect is more concretely studied under the statistical concept of autocorrelation, although this is beyond the scope of the book.*

After creating the predictor array X and the response vector y, we can partition the arrays into a *training* and a *test* set. The former subset is used to actually train the classifier, while the latter is used to actually test the performance. We are going to split the training and testing set on the 1st January 2005, leaving a full trading years worth of data (approximately 250 days) for

the testing set.

Once we create the training/testing split we need to create an array of classification models, each of which is in a tuple with an abbreviated name attached. While we have not set any parameters for the Logistic Regression, Linear/Quadratic Discriminant Analysers or Linear Support Vector Classifier models, we have used a set of default parameters for the Radial Support Vector Machine (RSVM) and the Random Forest (RF).

Finally we iterate over the models. We train (fit) each model on the training data and then make predictions on the testing set. Finally we output the hit rate and the confusion matrix for each model:

```
if __name__ == "__main__":
    # Create a lagged series of the S&P500 US stock market index
    snpret = create_lagged_series(
        "^GSPC", datetime.datetime(2001,1,10),
        datetime.datetime(2005,12,31), lags=5
    )

    # Use the prior two days of returns as predictor
    # values, with direction as the response
    X = snpret[["Lag1","Lag2"]]
    y = snpret["Direction"]

    # The test data is split into two parts: Before and after 1st Jan 2005.
    start_test = datetime.datetime(2005,1,1)

    # Create training and test sets
    X_train = X[X.index < start_test]
    X_test = X[X.index >= start_test]
    y_train = y[y.index < start_test]
    y_test = y[y.index >= start_test]

    # Create the (parametrised) models
    print("Hit Rates/Confusion Matrices:\n")
    models = [("LR", LogisticRegression()),
              ("LDA", LDA()),
              ("QDA", QDA()),
              ("LSVC", LinearSVC()),
              ("RSVM", SVC(
                C=1000000.0, cache_size=200, class_weight=None,
                coef0=0.0, degree=3, gamma=0.0001, kernel='rbf',
                max_iter=-1, probability=False, random_state=None,
                shrinking=True, tol=0.001, verbose=False)
              ),
              ("RF", RandomForestClassifier(
                n_estimators=1000, criterion='gini',
                max_depth=None, min_samples_split=2,
                min_samples_leaf=1, max_features='auto',
                bootstrap=True, oob_score=False, n_jobs=1,
                random_state=None, verbose=0)
              )]

    # Iterate through the models
    for m in models:

        # Train each of the models on the training set
        m[1].fit(X_train, y_train)
```

```
        # Make an array of predictions on the test set
        pred = m[1].predict(X_test)

        # Output the hit-rate and the confusion matrix for each model
        print("%s:\n%0.3f" % (m[0], m[1].score(X_test, y_test)))
        print("%s\n" % confusion_matrix(pred, y_test))
```

11.4.2 Results

The output from all of the classification models is as follows. You will likely see different values on the RF (Random Forest) output as it is inherently stochastic in its construction:

```
Hit Rates/Confusion Matrices:

LR:
0.560
[[ 35  35]
 [ 76 106]]

LDA:
0.560
[[ 35  35]
 [ 76 106]]

QDA:
0.599
[[ 30  20]
 [ 81 121]]

LSVC:
0.560
[[ 35  35]
 [ 76 106]]

RSVM:
0.563
[[  9   8]
 [102 133]]

RF:
0.504
[[48 62]
 [63 79]]
```

Note that all of the hit rates lie between 50% and 60%. Thus we can see that the lagged variables are not hugely indicative of future direction. However, if we look at the quadratic discriminant analyser we can see that its overall predictive performance on the test set is just under 60%.

The confusion matrix for this model (and the others in general) also states that the true positive rate for the "down" days is much higher than the "up" days. Thus if we are to create a trading strategy based off this information we could consider restricting trades to short positions of the S&P500 as a potential means of increasing profitability.

In later chapters we will use these models as a basis of a trading strategy by incorporating them directly into the event-driven backtesting framework and using a direct instrument, such as an exchange traded fund (ETF), in order to give us access to trading the S&P500.

Part V

Performance and Risk Management

Chapter 12

Performance Measurement

Performance measurement is an absolutely crucial component of algorithmic trading. Without assessment of performance, along with solid record keeping, it is difficult, if not impossible, to determine if our strategy returns have been due to luck or due to some actual edge over the market.

In order to be successful in algorithmic trading it is necessary to be aware of all of the factors that can affect the profitability of trades, and ultimately strategies. We should be constantly trying to find improvements in all aspects of the algorithmic trading stack. In particular we should always be trying to minimise our transaction costs (fees, commission and slippage), improve our software and hardware, improve the cleanliness of our data feeds and continually seek out new strategies to add to a portfolio. Performance measurement in all these areas provides a yardstick upon which to measure alternatives.

Ultimately, algorithmic trading is about generating profits. Hence it is imperative that we measure the performance, at multiple levels of granularity, of how and why our system is producing these profits. This motivates performance assessment at the level of trades, strategies and portfolios. In particular we are looking out for:

- Whether the systematic rules codified by the strategy actually produce a consistent return and whether the strategy possesses positive performance in the backtests.
- Whether a strategy maintains this positive performance in a live implementation or whether it needs to be retired.
- The ability to compare multiple strategies/portfolios such that we can reduce the *opportunity cost* associated with allocating a limited amount of trading capital.

The particular items of quantitative analysis of performance that we will be interested in are as follows:

- **Returns** - The most visible aspect of a trading strategy concerns the percentage gain since inception, either in a backtest or a live trading environment. The two major performance measures here are Total Return and Compound Annual Growth Rate (CAGR).
- **Drawdowns** - A *drawdown* is a period of negative performance, as defined from a prior *high-water mark*, itself defined as the previous highest peak on a strategy or portfolio *equity curve*. We will define this more concretely below, but you can think of it for now as a (somewhat painful!) downward slope on your performance chart.
- **Risk** - Risk comprises many areas, and we'll spend significant time going over them in the following chapter, but generally it refers to both risk of capital loss, such as with drawdowns, and volatility of returns. The latter usually being calculated as an annualised standard deviation of returns.
- **Risk/Reward Ratio** - Institutional investors are mainly interested with *risk-adjusted returns*. Since higher volatility can often lead to higher returns at the expense of greater

drawdowns, they are always concerned with how much risk is being taken on per unit of return. Consequently a range of performance measures have been invented to quantify this aspect of strategy performance, namely the Sharpe Ratio, Sortino Ratio and CALMAR Ratio, among others. The *out of sample Sharpe* is often the first metric to be discussed by institutional investors when discussing strategy performance.

- **Trade Analysis** - The previous measures of performance are all applicable to *strategies* and *portfolios*. It is also instructive to look at the performance of individual trades and many measures exist to characterise their performance. In particular, we will quantity the number of winning/losing trades, mean profit per trade and win/loss ratio among others.

Trades are the most granular aspect of an algorithmic strategy and hence we will begin by discussing trade analysis.

12.1 Trade Analysis

The first step in analysing any strategy is to consider the performance of the actual trades. Such metrics can vary dramatically between strategies. A classic example would be the difference in performance metrics of a trend-following strategy when compared to a mean-reverting strategy.

Trend-following strategies usually consist of many losing trades, each with a likely small loss. The lesser quantity of profitable trades occur when a trend has been established and the performance from these positive trades can significantly exceed the losses of the larger quantity of losing trades. Pair-trading mean-reverting strategies display the opposing character. They generally consist of many small profitable trades. However, if a series does not mean revert in the manner expected then the long/short nature of the strategy can lead to substantial losses. This could potentially wipe out the large quantity of small gains.

It is essential to be aware of the nature of the trade profile of the strategy and your own psychological profile, as the two will need to be in alignment. Otherwise you will find that you may not be able to persevere through a period of tough drawdown.

We now review the statistics that are of interest to us as the trade level.

12.1.1 Summary Statistics

When considering our trades, we are interested in the following set of statistics. Here "period" refers to the time period covered by the trading bar containing OHLCV data. For long-term strategies it is often the case that daily bars are used. For higher frequency strategies we may be interested in hourly or minutely bars.

- **Total Profit/Loss (PnL)** - The total PnL straightforwardly states whether a particular trade has been profitable or not.
- **Average Period PnL** - The avg. period PnL states whether a bar, on average, generates a profit or loss.
- **Maximum Period Profit** - The largest bar-period profit made by this trade so far.
- **Maximum Period Loss** - The largest bar-period loss made by this trade so far. Note that this says nothing about future maximum period loss! A future loss could be much larger than this.
- **Average Period Profit** - The average over the trade lifetime of all profitable periods.
- **Average Period Loss** - The average over the trade lifetime of all unprofitable periods.
- **Winning Periods** - The count of all winning periods.
- **Losing Periods** - The count of all losing periods.
- **Percentage Win/Loss Periods** - The percentage of all winning periods to losing periods. Will differ markedly for trend-following and mean-reverting type strategies.

Thankfully, it is straightforward to generate this information from our portfolio output and so the need for manual record keeping is completely eliminated. However, this leads to the danger that we never actually stop to analyse the data!

It is imperative that trades are evaluated at least once or twice a month. Doing so is a useful early warning detection system that can help identify when strategy performance begins to degrade. It is often much better than simply considering the cumulative PnL alone.

12.2 Strategy and Portfolio Analysis

Trade-level analysis is extremely useful in longer-term strategies, particularly with strategies that employ complex trades, such as those that involve derivatives. For higher-frequency strategies, we will be less interested in any individual trade and instead will want to consider the performance measures of the strategy instead. Obviously for longer-term strategies, we are equally as interested in the overall strategy performance. We are primarily interested in the following three key areas:

- **Returns Analysis** - The returns of a strategy encapsulate the concept of profitability. In institutional settings they are generally quoted net of fees and so provide a true picture of how much money was made on money invested. Returns can be tricky to calculate, especially with cash inflows/outflows.
- **Risk/Reward Analysis** - Generally the first consideration that external investors will have in a strategy is its out of sample *Sharpe Ratio* (which we describe below). This is an industry standard metric which attempts to characterise how much return was achieved per unit of risk.
- **Drawdown Analysis** - In an institutional setting, this is probably the most important of the three aspects. The profile and extent of the drawdowns of a strategy, portfolio or fund form a key component in risk management. We'll define drawdowns below.

Despite the fact that I have emphasised their institutional performance, as a retail trader these are still highly important metrics and with suitable risk management (see next chapter) will form the basis of a continual strategy evalation procedure.

12.2.1 Returns Analysis

The most widely quoted figures when discussing strategy performance, in both institutional and retail settings, are often *total return*, *annual returns* and *monthly returns*. It is extremely common to see a hedge fund performance newsletter with a monthly return "grid". In addition, everybody will want to know what the "return" of the strategy is.

Total return is relatively straightforward to calculate, at least in a retail setting with no external investors or cash inflows/outflows. In percentage terms it is simply calculated as:

$$r_t = (P_f - P_i)/P_i \times 100 \tag{12.1}$$

Where r_t is the total return, P_f is the final portfolio dollar value and P_i is the initial portfolio value. We are mostly interested in *net* total return, that is the value of the portfolio/fund after all trading/business costs have been deducted.

Note that this formula is only applicable to long-only un-leveraged portfolios. If we wish to add in short selling or leverage we need to modify how we calculate returns because we are technically trading on a larger borrowed portfolio than that used here. This is known as a *margin portfolio*.

For instance, consider the case where a trading strategy has gone long 1,000 USD of one asset and then shorted 1,000 USD of another asset. This is a *dollar-neutral* portfolio and the total *notional traded* is 2,000 USD. If 200 USD was generated from this strategy then gross return on this notional is 10%. It becomes more complex when you factor in borrowing costs and interest rates to fund the margin. Factoring in these costs leads to the net total return, which is the value that is often quoted as "total return".

Equity Curve

The equity curve is often one of the most emphasised visualisations on a hedge fund performance report - assuming the fund is doing well! It is a plot of the portfolio value of the fund over time. In essence it is used to show how the account has grown since fund inception. Equally, in a retail setting it is used to show growth of account equity through time. See Fig 12.2.1 for a typical equity curve plot:

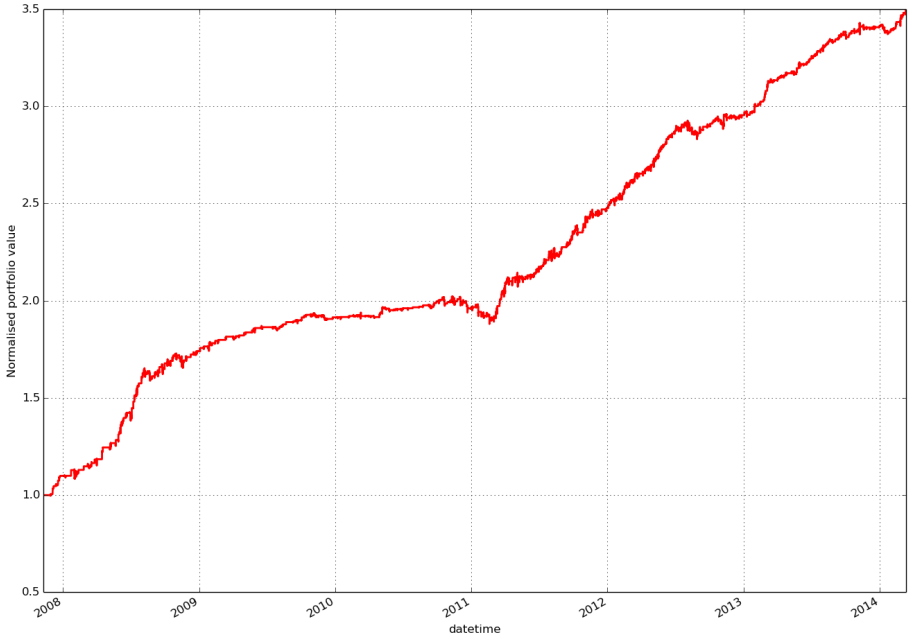


Figure 12.1: Typical intraday strategy equity curve

What is the benefit of such a plot? In essence it gives a "flavour" as to the past volatility of the strategy, as well as a visual indication of whether the strategy has suffered from prolonged periods of plateau or even drawdown. It essentially provides answers as to how the total return figure calculated at the end of the strategy trading period was arrived at.

In an equity curve we are seeking to determine how unusual historical events have shaped the strategy. For instance, a common question asks if there was excess volatility in the strategy around 2008. Another question might concern its consistency of returns.

One must be extremely careful with interpretation of equity curves as when marketed they are generally shown as "upward sloping lines". Interesting insight can be gained via truncation of such curves, which can emphasise periods of intense volatility or prolonged drawdown that may otherwise not seem as severe when considering the whole time period. Thus an equity curve needs to be considered in context with other analysis, in particular risk/reward analysis and drawdown analysis.

12.2.2 Risk/Reward Analysis

As we alluded to above the concept of risk-to-reward analysis is extremely important in an institutional setting. This **does not** mean that as a retail investor we can ignore the concept. You should pay significant attention to risk/reward metrics for your strategy as they will have a significant impact on your drawdowns, leverage and overall compound growth rate.

These concepts will be expanded on in the next chapter on Risk and Money Management. For now we will discuss the common ratios, and in particular the Sharpe Ratio, which is ubiquitous as a comparative measure in quantitative finance. Since it is held in such high regard across institutionalised quantitative trading, we will go into a reasonable amount of detail.

Sharpe Ratio

Consider the situation where we are presented with two strategies possessing identical returns. How do we know which one contains more risk? Further, what do we even mean by "more risk"? In finance, we are often concerned with volatility of returns and periods of drawdown. Thus if one of these strategies has a significantly higher volatility of returns we would likely find it less attractive, despite the fact that its historical returns might be similar if not identical. These problems of strategy comparison and risk assessment motivate the use of the **Sharpe Ratio**.

William Forsyth Sharpe is a Nobel-prize winning economist, who helped create the Capital Asset Pricing Model (CAPM) and developed the Sharpe Ratio in 1966 (later updated in 1994). The Sharpe Ratio S is defined by the following relation:

$$S = \frac{\mathbb{E}(R_a - R_b)}{\sqrt{\text{Var}(R_a - R_b)}} \tag{12.2}$$

Where R_a is the period return of the asset or strategy and R_b is the period return of a suitable *benchmark*, such as a risk-free interest rate.

The ratio compares the mean average of the *excess returns* of the asset or strategy with the standard deviation of those excess returns. Thus a lower volatility of returns will lead to a greater Sharpe ratio, assuming identical mean returns.

The "Sharpe Ratio" often quoted by those carrying out trading strategies is the **annualised Sharpe**, the calculation of which depends upon the trading period of which the returns are measured. Assuming there are N trading periods in a year, the annualised Sharpe is calculated as follows:

$$S_A = \sqrt{N}\frac{\mathbb{E}(R_a - R_b)}{\sqrt{\text{Var}(R_a - R_b)}}$$

Note that the Sharpe ratio itself MUST be calculated based on the Sharpe of that particular time period type. For a strategy based on trading period of days, $N = 252$ (as there are 252 trading days in a year, not 365), and R_a, R_b must be the daily returns. Similarly for hours $N = 252 \times 6.5 = 1638$, not $N = 252 \times 24 = 6048$, since there are only 6.5 hours in a trading day (at least for most US equities markets!).

The formula for the Sharpe ratio above alludes to the use of a *benchmark*. A benchmark is used as a "yardstick" or a "hurdle" that a particular strategy must overcome for it to worth consideration. For instance, a simple long-only strategy using US large-cap equities should hope to beat the S&P500 index on average, or match it for less volatility, otherwise what is to be gained by not simply investing in the index at far lower management/performance fees?

The choice of benchmark can sometimes be unclear. For instance, should a sector Exhange Traded Fund (ETF) be utilised as a performance benchmark for individual equities, or the S&P500 itself? Why not the Russell 3000? Equally should a hedge fund strategy be benchmarking itself against a market index or an index of other hedge funds?

There is also the complication of the "risk free rate". Should domestic government bonds be used? A basket of international bonds? Short-term or long-term bills? A mixture? Clearly there are plenty of ways to choose a benchmark. The Sharpe ratio generally utilises the risk-free rate and often, for US equities strategies, this is based on 10-year government Treasury bills.

In one particular instance, for market-neutral strategies, there is a particular complication regarding whether to make use of the risk-free rate or zero as the benchmark. The market index itself should not be utilised as the strategy is, by design, market-neutral. The correct choice for a market-neutral portfolio is not to substract the risk-free rate because it is *self-financing*. Since you gain a credit interest, R_f, from holding a margin, the actual calculation for returns

is: $(R_a + R_f) - R_f = R_a$. Hence there is no actual subtraction of the risk-free rate for dollar neutral strategies.

Despite the prevalence of the Sharpe ratio within quantitative finance, it does suffer from some limitations. The Sharpe ratio is *backward looking*. It only accounts for *historical* returns distribution and volatility, not those occurring in the *future*. When making judgements based on the Sharpe ratio there is an implicit assumption that the past will be similar to the future. This is evidently not always the case, particular under *market regime changes*.

The Sharpe ratio calculation assumes that the returns being used are normally distributed (i.e. *Gaussian*). Unfortunately, markets often suffer from kurtosis above that of a normal distribution. Essentially the distribution of returns has "fatter tails" and thus extreme events are more likely to occur than a Gaussian distribution would lead us to believe. Hence, the Sharpe ratio is poor at characterising *tail risk*.

This can be clearly seen in strategies which are highly prone to such risks. For instance, the sale of call options aka "pennies under a steam roller". A steady stream of option premia are generated by the sale of call options over time, leading to a low volatility of returns, with a strong excess returns above a benchmark. In this instance the strategy would possess a high Sharpe ratio based on historical data. However, it does not take into account that such options may be *called*, leading to significant drawdowns or even wipeout in the equity curve. Hence, as with any measure of algorithmic trading strategy performance the Sharpe ratio cannot be used in isolation.

Although this point might seem obvious to some, transaction costs MUST be included in the calculation of Sharpe ratio in order for it to be realistic. There are countless examples of trading strategies that have high Sharpes, and thus a likelihood of great profitability, only to be reduced to low Sharpe, low profitability strategies once realistic costs have been factored in. This means making use of the *net returns* when calculating in excess of the benchmark. Hence transaction costs must be factored in *upstream* of the Sharpe ratio calculation.

One obvious question that has remained unanswered thus far is "What is a good Sharpe Ratio for a strategy?". This is actually quite a difficult question to answer because each investor has a differing risk profile. The general rule of thumb is that quantitative strategies with annualised Sharpe Ratio $S < 1$ should not often be considered. However, there are exceptions to this, particularly in the trend-following futures space.

Quantitative funds tend to ignore any strategies that possess a Sharpe ratios $S < 2$. One prominent quantitative hedge fund that I am familiar with wouldn't even consider strategies that had Sharpe ratios $S < 3$ while in research. As a retail algorithmic trader, if you can achieve an out of sample (i.e. live trading!) Sharpe ratio $S > 2$ then you are doing very well.

The Sharpe ratio will often increase with trading frequency. Some high frequency strategies will have high single (and sometimes low double) digit Sharpe ratios, as they can be profitable almost every day and certainly every month. These strategies rarely suffer from catastrophic risk (in the sense of great loss) and thus minimise their volatility of returns, which leads to such high Sharpe ratios. Be aware though that high-frequency strategies such as these can simply cease to function very suddenly, which is another aspect of risk not fully reflected in the Sharpe ratio.

Let's now consider some actual Sharpe examples. We will start simply, by considering a long-only buy-and-hold of an individual equity then consider a market-neutral strategy. Both of these examples have been carried out with Pandas.

The first task is to actually obtain the data and put it into a Pandas DataFrame object. In the prior chapter on securities master implementation with Python and MySQL we created a system for achieving this. Alternatively, we can make use of this simpler code to grab Google Finance data directly and put it straight into a DataFrame. At the bottom of this script I have created a function to calculate the annualised Sharpe ratio based on a time-period returns stream:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# sharpe.py

from __future__ import print_function
```

```
import datetime
import numpy as np
import pandas as pd
import pandas.io.data as web


def annualised_sharpe(returns, N=252):
    """
    Calculate the annualised Sharpe ratio of a returns stream
    based on a number of trading periods, N. N defaults to 252,
    which then assumes a stream of daily returns.

    The function assumes that the returns are the excess of
    those compared to a benchmark.
    """
    return np.sqrt(N) * returns.mean() / returns.std()
```

Now that we have the ability to obtain data from Google Finance and straightforwardly calculate the annualised Sharpe ratio, we can test out a buy and hold strategy for two equities. We will use Google (GOOG) from Jan 1st 2000 to Jan 1st 2013.

We can create an additional helper function that allows us to quickly see buy-and-hold Sharpe across multiple equities for the same (hardcoded) period:

```
def equity_sharpe(ticker):
    """
    Calculates the annualised Sharpe ratio based on the daily
    returns of an equity ticker symbol listed in Google Finance.

    The dates have been hardcoded here for brevity.
    """
    start = datetime.datetime(2000,1,1)
    end = datetime.datetime(2013,1,1)

    # Obtain the equities daily historic data for the desired time period
    # and add to a pandas DataFrame
    pdf = web.DataReader(ticker, 'google', start, end)

    # Use the percentage change method to easily calculate daily returns
    pdf['daily_ret'] = pdf['Close'].pct_change()

    # Assume an average annual risk-free rate over the period of 5%
    pdf['excess_daily_ret'] = pdf['daily_ret'] - 0.05/252

    # Return the annualised Sharpe ratio based on the excess daily returns
    return annualised_sharpe(pdf['excess_daily_ret'])
```

For Google, the Sharpe ratio for buying and holding is `0.703`:

```
>>> equity_sharpe('GOOG')
0.70265563285799615
```

Now we can try the same calculation for a market-neutral strategy. The goal of this strategy is to fully isolate a particular equity's performance from the market in general. The simplest way to achieve this is to go short an equal amount (in dollars) of an Exchange Traded Fund (ETF) that is designed to track such a market. The most obvious choice for the US large-cap equities market is the S&P500 index, which is tracked by the SPDR ETF, with the ticker of SPY.

To calculate the annualised Sharpe ratio of such a strategy we will obtain the historical prices for SPY and calculate the percentage returns in a similar manner to the previous stocks, with

the exception that we will not use the risk-free benchmark. We will calculate the *net daily returns* which requires subtracting the difference between the long and the short returns and then dividing by 2, as we now have twice as much trading capital. Here is the Python/pandas code to carry this out:

```
def market_neutral_sharpe(ticker, benchmark):
    """
    Calculates the annualised Sharpe ratio of a market
    neutral long/short strategy inolving the long of 'ticker'
    with a corresponding short of the 'benchmark'.
    """
    start = datetime.datetime(2000, 1, 1)
    end = datetime.datetime(2013, 1, 1)

    # Get historic data for both a symbol/ticker and a benchmark ticker
    # The dates have been hardcoded, but you can modify them as you see fit!
    tick = web.DataReader(ticker, 'google', start, end)
    bench = web.DataReader(benchmark, 'google', start, end)

    # Calculate the percentage returns on each of the time series
    tick['daily_ret'] = tick['Close'].pct_change()
    bench['daily_ret'] = bench['Close'].pct_change()

    # Create a new DataFrame to store the strategy information
    # The net returns are (long - short)/2, since there is twice
    # the trading capital for this strategy
    strat = pd.DataFrame(index=tick.index)
    strat['net_ret'] = (tick['daily_ret'] - bench['daily_ret'])/2.0

    # Return the annualised Sharpe ratio for this strategy
    return annualised_sharpe(strat['net_ret'])
```

For Google, the Sharpe ratio for the long/short market-neutral strategy is `0.832`:

```
>>> market_neutral_sharpe('GOOG', 'SPY')
0.83197496084314604
```

We will now briefly consider other risk/reward ratios.

Sortino Ratio

The Sortino ratio is motivated by the fact that the Sharpe ratio captures both upward and downward volatility in its denominator. However, investors (and hedge fund managers) are generally not too bothered when we have significant upward volatility! What is actually of interest from a risk management perspective is downward volatility and periods of drawdown.

Thus the Sortino ratio is defined as the mean excess return divided by the mean downside deviation:

$$\text{Sortino} = \frac{\mathbb{E}(R_a - R_b)}{\sqrt{\text{Var}(R_a - R_b)_{\text{d}}}} \tag{12.3}$$

The Sortino is sometimes quoted in an institutional setting, but is certainly not as prevalent as the Sharpe ratio.

CALMAR Ratio

One could also argue that investors/traders are concerned solely with the maximum extent of the drawdown, rather than the average drawdown. This motivates the CALMAR (CALifornia

Managed Accounts Reports) ratio, also known as the Drawdown ratio, which provides a ratio of mean excess return to the maximum drawdown:

$$\text{CALMAR} = \frac{\mathbb{E}(R_a - R_b)}{\text{max. drawdown}} \tag{12.4}$$

Once again, the CALMAR is not as widely used as the Sharpe ratio.

12.2.3 Drawdown Analysis

In my opinion the concept of drawdown is the most important aspect of performance measurement for an algorithmic trading system. Simply put, if your account equity is wiped out then none of the other performance metrics matter! Drawdown analysis concerns the measurement of drops in account equity from previous *high water marks*. A high water mark is defined as the last account equity peak reached on the equity curve.

In an institutional setting the concept of drawdown is especially important as most hedge funds are remunerated only when the account equity is continually creating new high water marks. That is, a fund manager is not paid a performance fee while the fund remains "under water", i.e. the account equity is in a period of *drawdown*.

Most investors would be concerned at a drawdown of 10% in a fund, and would likely redeem their investment once a drawdown exceeds 30%. In a retail setting the situation is very different. Individuals are likely to be able to suffer deeper drawdowns in the hope of gaining higher returns.

Maximum Drawdown and Duration

The two key drawdown metrics are the *maximum drawdown* and the *drawdown duration*. The first describes the largest percentage drop from a previous peak to the current or previous trough in account equity. It is often quoted in an institutional setting when trying to market a fund. Retail traders should also pay significant attention to this figure. The second describes the actual duration of the drawdown. This figure is usually quoted in days, but higher frequency strategies might use a more granular time period.

In backtests these measures provide *some* idea of how a strategy *might* perform in the future. The overall account equity curve might look quite appealing after a calculated backtest. However, an upward equity curve can easily mask how difficult previous periods of drawdown might actually have been to experience.

When a strategy begins dropping below 10% of account equity, or even below 20%, it requires significant willpower to continue with the strategy, despite the fact that the strategy may have historically, at least in the backtests, been through similar periods. This is a consistent issue with algorithmic trading and systematic trading in general. It naturally motivates the need to set prior drawdown boundaries and specific rules, such as an account-wide "stop loss" that will be carried out in the event of a drawdown breaching these levels.

Drawdown Curve

While it is important to be aware of the maximum drawdown and drawdown duration, it is significantly more instructive to see a time series plot of the strategy drawdown over the trading duration.

Fig 12.2.3 quite clearly shows that this particular strategy suffered from a relatively sustained period of drawdown beginning in Q3 of 2010 and finishing in Q2 of 2011, reaching a maximum drawdown of 14.8%. While the strategy itself continued to be significantly profitable over the long term, this particular period would have been very difficult to endure. In addition, this is the maximum *historical* drawdown that has occured *to date*. The strategy may be subject to an even greater drawdown in the future. Thus it is necessary to consider drawdown curves, as with other historical looking performance measures, in the context with which they have been generated, namely via historical, and not future, data.

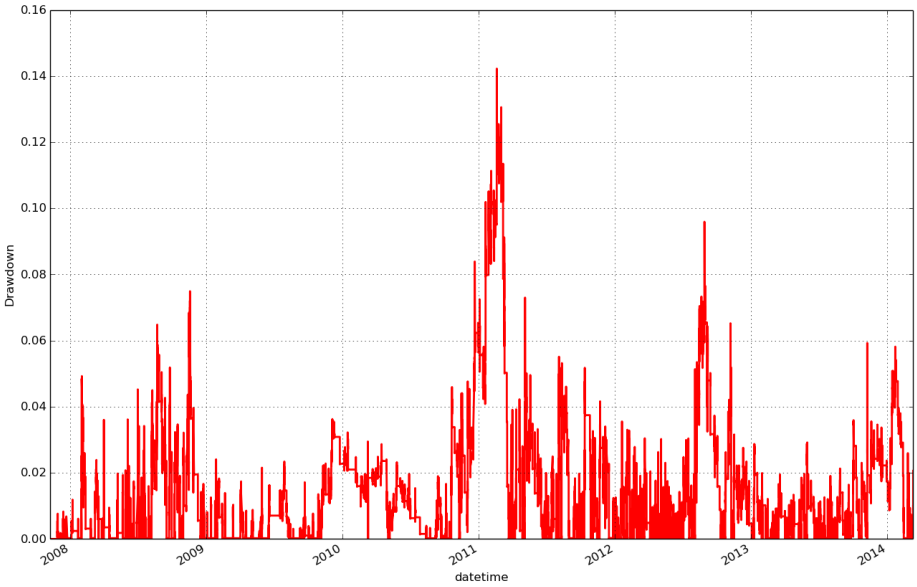


Figure 12.2: Typical intraday strategy drawdown curve

In the following chapter we will consider the concept of quantitative risk management and describe techniques that can help us to minimise drawdowns and maximise returns, all the while keeping to a reasonable degree of risk.

Chapter 13

Risk and Money Management

This chapter is concerned with managing risk as applied to quantitative trading strategies. This usually comes in two flavours, firstly identifying and mitigating internal and external factors that can affect the performance or operation of an algorithmic trading strategy and secondly, how to optimally manage the strategy portfolio in order to maximise growth rate and minimise account drawdowns.

In the first section we will consider different sources of risk (both intrinsic and extrinsic) that might affect the long-term performance of an algorithmic trading business - either retail or institutional.

In the second section we will look at money management techniques that can simultaneously protect our portfolio from ruin and also attempt to maximise the long-term growth rate of equity.

In the final section we consider institutional-level risk management techniques that can easily be applied in a retail setting to help protect trading capital.

13.1 Sources of Risk

The are numerous sources of risk that can have an impact on the correct functioning of an algorithmic trading strategy. "Risk" is usually defined in this context to mean chance of account losses. However, I am going to define it in a much broader context to mean any factor that provides a degree of uncertainty and could affect the performance of our strategies or portfolio.

The broad areas of risk that we will consider include **Strategy Risk**, **Portfolio Risk**, **Market Risk**, **Counterparty Risk** and **Operational Risk**.

13.1.1 Strategy Risk

Strategy risk, or *model risk*, encompasses the class of risks that arise from the design and implementation of a trading strategy based on a statistical model. It includes all of the previous issues we have discussed in the Successful Backtesting chapter, such as curve-fitting, survivorship bias and look-ahead bias. It also includes other topics related directly to the statistical analysis of the strategy model.

Any statistical model is based on assumptions. These assumptions are sometimes not considered in proper depth or ignored entirely. This means that the statistical model based upon these assumptions may be inappropriate and hence lead to poor predictive or inferential capability. A general example occurs in the setting of linear regression. Linear regression makes the assumption that the response data are *homoscedastic* (i.e. the responses have a constant variance in their errors). If this is not the case then linear regression provides less precision in the estimates of parameters.

Many quantitative strategies make use of *descriptive statistics* of historical price data. In particular, they will often use *moments* of the data such as the mean, variance, skew and kurtosis of strategy returns. Such models (including the Kelly Criterion outlined below) generally rely on these moments being constant in time. Under a *market regime change* these moments can be

drastically altered and hence lead to degradation of the model. Models with "rolling parameters" are usually utilised in order to mitigate this issue.

13.1.2 Portfolio Risk

A Portfolio contains one or more strategies. Thus it is indirectly subject to Strategy Risk as outlined above. In addition there are specific risks that occur at the portfolio level. These are usually only considered in an institutional setting or in a high-end retail setting where portfolio tracking is being carried out on a stable of trading strategies.

When regressing portfolio returns to a set of *factors*, such as industry sectors, asset classes or groups of financial entities it is possible to ascertain if the portfolio is heavily "loaded" into a particular factor. For instance, an equities portfolio may be extremely heavy on technology stocks and thus is extremely exposed to any issues that affect the tech sector as a whole. Hence it is often necessary - at the portfolio level - to override particular strategies in order to account for overloaded factor risk. This is often a more significant concern in an institutional setting where there is more capital to be allocated and the preservation of capital takes precedence to the long-term growth rate of the capital. However, it should certainly be considered even as a retail algorithmic trading investor.

Another issue that is largely an institutional issue (unless trading more illiquid assets) are limits on daily trading volume. For retail traders, executing strategies in the large-cap or commodities futures markets, there is no real concern regarding market impact. However, in less liquid instruments one has to be careful to not be trading a significant percentage of the daily traded volume, due to potential market impact and thus invalidation of a previously backtested trading model (which often do not take into account market impact). To avoid this it is necessary to calculate the *average daily volume* (using a mean over a loopback period, for instance) and stay within small percentage limits of this figure.

Running a portfolio of strategies brings up the issue of *strategy correlation*. Correlations can be estimated via statistical techniques such as the Pearson Product Moment Correlation Coefficient. However, correlation itself is not a static entity and is also subject to swift change, especially under market-wide liquidity constraints, often known as *financial contagion*. In general, strategies should be designed to avoid correlation with each other by virtue of differing asset classes or time horizons. Rolling correlations can be estimated over a large time-period and should be a standard part of your backtest, if considering a portfolio approach.

13.1.3 Counterparty Risk

Counterparty risk is generally considered a form of *credit risk*. It is the risk that a counterpart will not pay an obligation on a financial asset under which they are liable. There is an entire subset of quantitative finance related to the pricing of counterpart risk hedging instruments, but this is not of primary interest to us as retail algorithmic traders. We are more concerned with the risk of default from *suppliers* such as an exchange or brokerage.

While this may seem academic, I can personally assure you that these issues are quite real! In an institutional setting I have experienced first hand a brokerage bankruptcy under conditions that meant not all of the trading capital was returned. Thus I now factor such risks into a portfolio. The suggested means of mitigating this issue is to utilise multiple brokerages, although when trading under margin this can make the trading logistics somewhat tricky.

Counterparty risk is generally more of a concern in an institutional setting so I won't dwell on it too much here!

13.1.4 Operational Risk

Operation risk encompasses sources of risk from within a fund or trading operational infrastructure, including business/entrepreneurial risk, IT risk and external regulatory or legal changes. These topics aren't often discussed in any great depth, which I believe is somewhat short-sighted since they have the potential to completely halt a trading operation permanently.

Infrastructure risk is often associated with information technology systems and other related trading infrastructure. This also includes employee risk (such as fraud, sudden departure). As the scale of an infrastructure grows so does the likelihood of the "single point of failure" (SPOF). This is a critical component in the trading infrastructure that, under malfunction, can lead to a catastrophe halting of the entire operation. In the IT sense, this is usually the consequence of a badly thought out architecture. In a non-IT sense this can be the consequence of a badly designed organisation chart.

These issues are still entirely relevant for the retail trader. Often, an IT/trading infrastructure can end up being "patchy" and "hacked together". In addition, poor record keeping and other administrative failures can lead to huge potential tax burdens. Thankfully, "cloud" architecture provides the ability for redundancy in systems and automation of processes can lead to solid administrative habits. This type of behaviour, that is consideration of risks from sources other than the market and the strategy, can often make the difference between a successful long-term algorithmic trader and the individual who gives up due to catastrophic operation breakdown.

An issue that affects the hedge fund world is that of reporting and compliance. Post-2008 legislation has put a heavy burden on asset management firms, which can have a large impact on their cash-flow and operating expenditure. For an individual thinking of incorporating such a firm, in order to expand a strategy or run with external funds, it is prudent to keep on top of the legislation and regulatory environment, since it is somewhat of a "moving target".

13.2 Money Management

This section deals with one of the most fundamental concepts in trading - both discretionary and algorithmic - namely, money management. A naive investor/trader might believe that the only important investment objective is to simply make as much money as possible. However the reality of long-term trading is more complex. Since market participants have differing risk preferences and constraints there are many objectives that investors may possess.

Many retail traders consider the *only* goal to be a continual increase of account equity, with little or no consideration given to the "risk" of a strategy that achieves this growth. More sophisticated retail investors measure account drawdowns, and depending upon their risk preferences, may be able to cope with a substantial drop in account equity (say 50%). The reason they can deal with a drawdown of this magnitude is that they realise, quantitatively, that this behaviour may be optimal for the long-term growth rate of the portfolio, via the use of *leverage*.

An institutional investor is likely to consider risk in a different light. Often institutional investors have mandated maximum drawdowns (say 20%), with significant consideration given to sector allocation and average daily volume limits. They would be additional constraints on the "optimisation problem" of capital allocation to strategies. These factors might even be more important than maximising the long-term growth rate of the portfolio.

Thus we are in a situation where we can strike a balance between maximising long-term growth rate via leverage and minimising our "risk" by trying to limit the duration and extent of the drawdown. The major tool that will help us achieve this is called the Kelly Criterion.

13.2.1 Kelly Criterion

Within this section the Kelly Criterion is going to be our tool to control leverage of, and allocation towards, a set of algorithmic trading strategies that make up a multi-strategy portfolio.

We will define **leverage** as the ratio of the size of a portfolio to the actual account equity within that portfolio. To make this clear we can use the analogy of purchasing a house with a mortgage. Your down payment (or "deposit" for those of us in the UK!) constitutes your account equity, while the down payment plus the mortgage value constitutes the equivalent of the size of a portfolio. Thus a down payment of 50,000 USD on a 200,000 USD house (with a mortgage of 150,000 USD) constitutes a leverage of $(150000 + 50000)/50000 = 4$. Thus in this instance you would be 4x leveraged on the house. A margin account portfolio behaves similarly. There is a "cash" component and then more stock can be borrowed on margin, to provide the leverage.

Before we state the Kelly Criterion specifically I want to outline the assumptions that go into its derivation, which have varying degrees of accuracy:

- Each algorithmic trading strategy will be assumed to possess a returns stream that is *normally distributed* (i.e. *Gaussian*). Further, each strategy has its own *fixed* mean and standard deviation of returns. The formula assumes that these mean and std values *do not change*, i.e. that they are same in the past as in the future. This is clearly not the case with most strategies, so be aware of this assumption.

- The returns being considered here are *excess returns*, which means they are net of all financing costs such as interest paid on margin and transaction costs. If the strategy is being carried out in an institutional setting, this also means that the returns are net of management and performance fees.

- All of the trading profits are reinvested and no withdrawals of equity are carried out. This is clearly not as applicable in an institutional setting where the above mentioned management fees are taken out and investors often make withdrawals.

- All of the strategies are statistically independent (there is no correlation between strategies) and thus the covariance matrix between strategy returns is diagonal.

Now we come to the actual Kelly Criterion! Let's imagine that we have a set of N algorithmic trading strategies and we wish to determine both how to apply optimal leverage per strategy in order to maximise growth rate (but minimise drawdowns) and how to allocate capital between each strategy. If we denote the allocation between each strategy i as a vector f of length N, s.t. $f = (f_1, ..., f_N)$, then the Kelly Criterion for optimal allocation to each strategy f_i is given by:

$$f_i = \mu_i / \sigma_i^2 \tag{13.1}$$

Where μ_i are the mean excess returns and σ_i are the standard deviation of excess returns for a strategy i. This formula essentially describes the optimal leverage that should be applied to each strategy.

While the Kelly Criterion f_i gives us the optimal leverage and strategy allocation, we still need to actually calculate our expected long-term compounded growth rate of the portfolio, which we denote by g. The formula for this is given by:

$$g = r + S^2/2 \tag{13.2}$$

Where r is the risk-free interest rate, which is the rate at which you can borrow from the broker, and S is the annualised Sharpe Ratio of the strategy. The latter is calculated via the annualised mean excess returns divided by the annualised standard deviations of excess returns. See the previous chapter on Performance Measurement for details on the Sharpe Ratio.

A Realistic Example

Let's consider an example in the single strategy case ($i = 1$). Suppose we go long a mythical stock XYZ that has a mean annual return of $m = 10.7\%$ and an annual standard deviation of $\sigma = 12.4\%$. In addition suppose we are able to borrow at a risk-free interest rate of $r = 3.0\%$. This implies that the mean excess returns are $\mu = m - r = 10.7 - 3.0 = 7.7\%$. This gives us a Sharpe Ratio of $S = 0.077/0.124 = 0.62$.

With this we can calculate the optimal Kelly leverage via $f = \mu/\sigma^2 = 0.077/0.124^2 = 5.01$. Thus the Kelly leverage says that for a 100,000 USD portfolio we should borrow an additional 401,000 USD to have a total portfolio value of 501,000 USD. *In practice it is unlikely that our brokerage would let us trade with such substantial margin and so the Kelly Criterion would need to be adjusted.*

We can then use the Sharpe ratio S and the interest rate r to calculate g, the expected long-term compounded growth rate. $g = r + S^2/2 = 0.03 + 0.62^2/2 = 0.22$, i.e. 22%. Thus we should *expect* a return of 22% a year from this strategy.

Kelly Criterion in Practice

It is important to be aware that the Kelly Criterion requires a continuous rebalancing of capital allocation in order to remain valid. Clearly this is not possible in the discrete setting of actual trading and so an approximation must be made. The standard "rule of thumb" here is to update the Kelly allocation once a day. Further, the Kelly Criterion itself should be recalculated periodically, using a trailing mean and standard deviation with a lookback window. Again, for a strategy that trades roughly once a day, this lookback should be set to be on the order of 3-6 months of daily returns.

Here is an example of rebalancing a portfolio under the Kelly Criterion, which can lead to some counter-intuitive behaviour. Let's suppose we have the strategy described above. We have used the Kelly Criterion to borrow cash to size our portfolio to 501,000 USD. Let's assume we make a healthy 5% return on the following day, which boosts our account size to 526,050 USD. The Kelly Criterion tells us that we should borrow *more* to keep the same leverage factor of 5.01. In particular our account equity is 126,050 USD on a portfolio of 526,050, which means that the current leverage factor is 4.17. To increase it to 5.01, we need to borrow an additional 105,460 USD in order to increase our account size to 631,510.5 USD (this is 5.01×126050).

Now consider that the following day we lose 10% on our portfolio (ouch!). This means that the total portfolio size is now 568,359.45 USD (631510.5×0.9). Our total account *equity* is now 62,898.95 USD ($126050 - 631510.45 \times 0.1$). This means our current leverage factor is $568359.45/62898.95 = 9.03$. Hence we need to reduce our account by *selling* 253,235.71 USD of stock in order to reduce our total portfolio value to 315,123.73 USD, such that we have a leverage of 5.01 again ($315123.73/62898.95 = 5.01$).

Hence we have *bought* into a profit and *sold* into a loss. This process of selling into a loss may be extremely emotionally difficult, but it is mathematically the "correct" thing to do, assuming that the assumptions of Kelly have been met! It is the approach to follow in order to maximise long-term compounded growth rate.

You may have noticed that the absolute values of money being re-allocated between days were rather severe. This is a consequence of both the artificial nature of the example and the extensive leverage employed. 10% loss in a day is not particularly common in higher-frequency algorithmic trading, but it does serve to show how extensive leverage can be on absolute terms.

Since the estimation of means and standard deviations are always subject to uncertainty, in practice many traders tend to use a more conservative leverage regime such as the Kelly Criterion divided by two, affectionately known as "half-Kelly". The Kelly Criterion should really be considered as an upper bound of leverage to use, rather than a direct specification. If this advice is not heeded then using the direct Kelly value can lead to ruin (i.e. account equity disappearing to zero) due to the non-Gaussian nature of the strategy returns.

Should You Use The Kelly Criterion?

Every algorithmic trader is different and the same is true of risk preferences. When choosing to employ a leverage strategy (of which the Kelly Criterion is one example) you should consider the risk mandates that you need to work under. In a retail environment you are able to set your own maximum drawdown limits and thus your leverage can be increased. In an institutional setting you will need to consider risk from a very different perspective and the leverage factor will be one component of a much larger framework, usually under many other constraints.

13.3 Risk Management

13.3.1 Value-at-Risk

Estimating the risk of loss to an algorithmic trading strategy, or portfolio of strategies, is of extreme importance for long-term capital growth. Many techniques for risk management have been developed for use in institutional settings. One technique in particular, known as **Value at Risk** or **VaR**, will be the topic of this section.

We will be applying the concept of VaR to a single strategy or a set of strategies in order to help us quantify risk in our trading portfolio. The definition of VaR is as follows:

VaR provides an *estimate*, under a given degree of confidence, of the size of a loss from a portfolio over a given time period.

In this instance "portfolio" can refer to a single strategy, a group of strategies, a trader's book, a prop desk, a hedge fund or an entire investment bank. The "given degree of confidence" will be a value of, say, 95% or 99%. The "given time period" will be chosen to reflect one that would lead to a minimal *market impact* if a portfolio were to be liquidated.

For example, a VaR equal to 500,000 USD at 95% confidence level for a time period of a day would simply state that there is a 95% probability of losing no more than 500,000 USD in the following day. Mathematically this is stated as:

$$P(L \leq -5.0 \times 10^5) = 0.05 \tag{13.3}$$

Or, more generally, for loss L exceeding a value VaR with a confidence level c we have:

$$P(L \leq -VaR) = 1 - c \tag{13.4}$$

The "standard" calculation of VaR makes the following assumptions:

- **Standard Market Conditions** - VaR is not supposed to consider extreme events or "tail risk", rather it is supposed to provide the expectation of a loss under normal "day-to-day" operation.
- **Volatilities and Correlations** - VaR requires the volatilities of the assets under consideration, as well as their respective correlations. These two quantities are tricky to estimate and are subject to continual change.
- **Normality of Returns** - VaR, in its standard form, assumes the returns of the asset or portfolio are *normally distributed*. This leads to more straightforward analytical calculation, but it is quite unrealistic for most assets.

13.4 Advantages and Disadvantages

VaR is pervasive in the financial industry, hence you should be familiar with the benefits and drawbacks of the technique. Some of the advantages of VaR are as follows:

- VaR is very straightforward to calculate for individual assets, algo strategies, quant portfolios, hedge funds or even bank prop desks.
- The time period associated with the VaR can be modified for multiple trading strategies that have different time horizons.
- Different values of VaR can be associated with different forms of risk, say broken down by asset class or instrument type. This makes it easy to interpret where the majority of portfolio risk may be clustered, for instance.
- Individual strategies can be constrained as can entire portfolios based on their individual VaR.
- VaR is straightforward to interpret by (potentially) non-technical external investors and fund managers.

However, VaR is not without its disadvantages:

- VaR does not discuss the magnitude of the expected loss beyond the value of VaR, i.e. it will tell us that we are likely to see a loss *exceeding* a value, but not how much it exceeds it.

- It does not take into account extreme events, but only typical market conditions.
- Since it uses historical data (it is rearward-looking) it will not take into account future market regime shifts that can change volatilities and correlations of assets.

VaR should not be used in isolation. It should always be used with a suite of risk management techniques, such as diversification, optimal portfolio allocation and prudent use of leverage.

Methods of Calculation

As of yet we have not discussed the actual calculation of VaR, either in the general case or a concrete trading example. There are three techniques that will be of interest to us. The first is the variance-covariance method (using normality assumptions), the second is a Monte Carlo method (based on an underlying, potentially non-normal, distribution) and the third is known as historical bootstrapping, which makes use of historical returns information for assets under consideration.

In this section we will concentrate on the Variance-Covariance Method.

Variance-Covariance Method

Consider a portfolio of P dollars, with a confidence level c. We are considering daily returns, with asset (or strategy) historical standard deviation σ and mean μ. Then the *daily* VaR, under the variance-covariance method for a single asset (or strategy) is calculated as:

$$P - (P(\alpha(1-c)+1)) \tag{13.5}$$

Where α is the inverse of the cumulative distribution function of a normal distribution with mean μ and standard deviation σ.

We can use the SciPy and pandas libraries in order to calculate these values. If we set $P = 10^6$ and $c = 0.99$, we can use the SciPy ppf method to generate the values for the inverse cumulative distribution function to a normal distribution with μ and σ obtained from some real financial data, in this case the historical daily returns of CitiGroup (we could easily substitute the returns of an algorithmic strategy in here):

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# var.py

from __future__ import print_function

import datetime

import numpy as np
import pandas.io.data as web
from scipy.stats import norm


def var_cov_var(P, c, mu, sigma):
    """
    Variance-Covariance calculation of daily Value-at-Risk
    using confidence level c, with mean of returns mu
    and standard deviation of returns sigma, on a portfolio
    of value P.
    """
    alpha = norm.ppf(1-c, mu, sigma)
    return P - P*(alpha + 1)
```

```
if __name__ == "__main__":
    start = datetime.datetime(2010, 1, 1)
    end = datetime.datetime(2014, 1, 1)

    citi = web.DataReader("C", 'yahoo', start, end)
    citi["rets"] = citi["Adj Close"].pct_change()

    P = 1e6   # 1,000,000 USD
    c = 0.99  # 99% confidence interval
    mu = np.mean(citi["rets"])
    sigma = np.std(citi["rets"])

    var = var_cov_var(P, c, mu, sigma)
    print("Value-at-Risk: $%0.2f" % var)
```

The calculated value of VaR is given by:

```
Value-at-Risk: $56503.12
```

VaR is an extremely useful and pervasive technique in all areas of financial management, but it is not without its flaws. David Einhorn, the renowned hedge fund manager, has famously described VaR as "an airbag that works all the time, except when you have a car accident." Indeed, you should always use VaR as an augmentation to your risk management overlay, not as a single indicator!

Part VI

Automated Trading

Chapter 14

Event-Driven Trading Engine Implementation

This chapter provides an implementation for a fully self-contained event-driven backtest system written in Python. In particular this chapter has been written to expand on the details that are usually omitted from other algorithmic trading texts and papers. The following code will allow you to simulate high-frequency (minute to second) strategies across the forecasting, momentum and mean reversion domains in the equities, foreign exchange and futures markets.

With extensive detail comes complexity, however. The backtesting system provided here requires many components, each of which are comprehensive entities in themselves. The first step is thus to outline what event-driven software is and then describe the components of the backtester and how the entire system fits together.

14.1 Event-Driven Software

Before we delve into development of such a backtester we need to understand the concept of event-driven systems. Video games provide a natural use case for event-driven software and provide a straightforward example to explore. A video game has multiple components that interact with each other in a real-time setting at high framerates. This is handled by running the entire set of calculations within an "infinite" loop known as the event-loop or game-loop.

At each tick of the game-loop a function is called to receive the latest event, which will have been generated by some corresponding prior action within the game. Depending upon the nature of the event, which could include a key-press or a mouse click, some subsequent action is taken, which will either terminate the loop or generate some additional events. The process will then continue.

Here is some example pseudo-code:

```
while True:  # Run the loop forever
    new_event = get_new_event()   # Get the latest event

    # Based on the event type, perform an action
    if new_event.type == "LEFT_MOUSE_CLICK":
        open_menu()
    elif new_event.type == "ESCAPE_KEY_PRESS":
        quit_game()
    elif new_event.type == "UP_KEY_PRESS":
        move_player_north()
    # ... and many more events

    redraw_screen()   # Update the screen to provide animation
    tick(50)   # Wait 50 milliseconds
```

The code is continually checking for new events and then performing actions based on these events. In particular it allows the illusion of real-time response handling because the code is continually being looped and events checked for. As will become clear this is precisely what we need in order to carry out high frequency trading simulation.

14.1.1 Why An Event-Driven Backtester?

Event-driven systems provide many advantages over a vectorised approach:

- **Code Reuse** - An event-driven backtester, by design, can be used for both historical backtesting and live trading with minimal switch-out of components. This is not true of vectorised backtesters where all data must be available at once to carry out statistical analysis.
- **Lookahead Bias** - With an event-driven backtester there is no lookahead bias as market data receipt is treated as an "event" that must be acted upon. Thus it is possible to "drip feed" an event-driven backtester with market data, replicating how an order management and portfolio system would behave.
- **Realism** - Event-driven backtesters allow significant customisation over how orders are executed and transaction costs are incurred. It is straightforward to handle basic market and limit orders, as well as market-on-open (MOO) and market-on-close (MOC), since a custom exchange handler can be constructed.

Although event-driven systems come with many benefits they suffer from two major disadvantages over simpler vectorised systems. Firstly they are significantly more complex to implement and test. There are more "moving parts" leading to a greater chance of introducing bugs. To mitigate this proper software testing methodology such as test-driven development can be employed.

Secondly they are slower to execute compared to a vectorised system. Optimal vectorised operations are unable to be utilised when carrying out mathematical calculations.

14.2 Component Objects

To apply an event-driven approach to a backtesting system it is necessary to define our components (or objects) that will handle specific tasks:

- **Event** - The Event is the fundamental class unit of the event-driven system. It contains a type (such as "MARKET", "SIGNAL", "ORDER" or "FILL") that determines how it will be handled within the event-loop.
- **Event Queue** - The Event Queue is an in-memory Python Queue object that stores all of the Event sub-class objects that are generated by the rest of the software.
- **DataHandler** - The DataHandler is an abstract base class (ABC) that presents an interface for handling both historical or live market data. This provides significant flexibility as the Strategy and Portfolio modules can thus be reused between both approaches. The DataHandler generates a new MarketEvent upon every heartbeat of the system (see below).
- **Strategy** - The Strategy is also an ABC that presents an interface for taking market data and generating corresponding SignalEvents, which are ultimately utilised by the Portfolio object. A SignalEvent contains a ticker symbol, a direction (LONG or SHORT) and a timestamp.
- **Portfolio** - This is a class hierarchy which handles the order management associated with current and subsequent positions for a strategy. It also carries out risk management across the portfolio, including sector exposure and position sizing. In a more sophisticated implementation this could be delegated to a RiskManagement class. The Portfolio takes SignalEvents from the Queue and generates OrderEvents that get added to the Queue.

- **ExecutionHandler** - The ExecutionHandler simulates a connection to a brokerage. The job of the handler is to take OrderEvents from the Queue and execute them, either via a simulated approach or an actual connection to a liver brokerage. Once orders are executed the handler creates FillEvents, which describe what was actually transacted, including fees, commission and slippage (if modelled).
- **Backtest** - All of these components are wrapped in an event-loop that correctly handles all Event types, routing them to the appropriate component.

Despite the quantity of components, this is quite a basic model of a trading engine. There is significant scope for expansion, particularly in regard to how the Portfolio is used. In addition differing transaction cost models might also be abstracted into their own class hierarchy.

14.2.1 Events

The first component to be discussed is the Event class hierarchy. In this infrastructure there are four types of events which allow communication between the above components via an event queue. They are a MarketEvent, SignalEvent, OrderEvent and FillEvent.

Event

The parent class in the hierarchy is called Event. It is a base class and does not provide any functionality or specific interface. Since in many implementations the Event objects will likely develop greater complexity it is thus being "future-proofed" by creating a class hierarchy.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# event.py

from __future__ import print_function


class Event(object):
    """
    Event is base class providing an interface for all subsequent
    (inherited) events, that will trigger further events in the
    trading infrastructure.
    """
    pass
```

MarketEvent

MarketEvents are triggered when the outer while loop of the backtesting system begins a new "heartbeat". It occurs when the DataHandler object receives a new update of market data for any symbols which are currently being tracked. It is used to trigger the Strategy object generating new trading signals. The event object simply contains an identification that it is a market event, with no other structure.

```
# event.py

class MarketEvent(Event):
    """
    Handles the event of receiving a new market update with
    corresponding bars.
    """

    def __init__(self):
```

```
        """
        Initialises the MarketEvent.
        """
        self.type = 'MARKET'
```

SignalEvent

The Strategy object utilises market data to create new SignalEvents. The SignalEvent contains a strategy ID, a ticker symbol, a timestamp for when it was generated, a direction (long or short) and a "strength" indicator (this is useful for mean reversion strategies). The SignalEvents are utilised by the Portfolio object as advice for how to trade.

```
# event.py

class SignalEvent(Event):
    """
    Handles the event of sending a Signal from a Strategy object.
    This is received by a Portfolio object and acted upon.
    """

    def __init__(self, strategy_id, symbol, datetime, signal_type, strength):
        """
        Initialises the SignalEvent.

        Parameters:
        strategy_id - The unique identifier for the strategy that
            generated the signal.
        symbol - The ticker symbol, e.g. 'GOOG'.
        datetime - The timestamp at which the signal was generated.
        signal_type - 'LONG' or 'SHORT'.
        strength - An adjustment factor "suggestion" used to scale
            quantity at the portfolio level. Useful for pairs strategies.
        """

        self.type = 'SIGNAL'
        self.strategy_id = strategy_id
        self.symbol = symbol
        self.datetime = datetime
        self.signal_type = signal_type
        self.strength = strength
```

OrderEvent

When a Portfolio object receives SignalEvents it assesses them in the wider context of the portfolio, in terms of risk and position sizing. This ultimately leads to OrderEvents that will be sent to an ExecutionHandler.

The OrderEvent is slightly more complex than a SignalEvent since it contains a quantity field in addition to the aforementioned properties of SignalEvent. The quantity is determined by the Portfolio constraints. In addition the OrderEvent has a print_order() method, used to output the information to the console if necessary.

```
# event.py

class OrderEvent(Event):
    """
    Handles the event of sending an Order to an execution system.
    The order contains a symbol (e.g. GOOG), a type (market or limit),
```

```
    quantity and a direction.
    """

    def __init__(self, symbol, order_type, quantity, direction):
        """
        Initialises the order type, setting whether it is
        a Market order ('MKT') or Limit order ('LMT'), has
        a quantity (integral) and its direction ('BUY' or
        'SELL').

        Parameters:
        symbol - The instrument to trade.
        order_type - 'MKT' or 'LMT' for Market or Limit.
        quantity - Non-negative integer for quantity.
        direction - 'BUY' or 'SELL' for long or short.
        """

        self.type = 'ORDER'
        self.symbol = symbol
        self.order_type = order_type
        self.quantity = quantity
        self.direction = direction

    def print_order(self):
        """
        Outputs the values within the Order.
        """
        print(
            "Order: Symbol=%s, Type=%s, Quantity=%s, Direction=%s" %
            (self.symbol, self.order_type, self.quantity, self.direction)
        )
```

FillEvent

When an ExecutionHandler receives an OrderEvent it must transact the order. Once an order has been transacted it generates a FillEvent, which describes the cost of purchase or sale as well as the transaction costs, such as fees or slippage.

The FillEvent is the Event with the greatest complexity. It contains a timestamp for when an order was filled, the symbol of the order and the exchange it was executed on, the quantity of shares transacted, the actual price of the purchase and the commission incurred.

The commission is calculated using the Interactive Brokers commissions. For US API orders this commission is 1.30 USD minimum per order, with a flat rate of either 0.013 USD or 0.08 USD per share depending upon whether the trade size is below or above 500 units of stock.

```
# event.py

class FillEvent(Event):
    """
    Encapsulates the notion of a Filled Order, as returned
    from a brokerage. Stores the quantity of an instrument
    actually filled and at what price. In addition, stores
    the commission of the trade from the brokerage.
    """

    def __init__(self, timeindex, symbol, exchange, quantity,
                 direction, fill_cost, commission=None):
```

```
        """
        Initialises the FillEvent object. Sets the symbol, exchange,
        quantity, direction, cost of fill and an optional
        commission.

        If commission is not provided, the Fill object will
        calculate it based on the trade size and Interactive
        Brokers fees.

        Parameters:
        timeindex - The bar-resolution when the order was filled.
        symbol - The instrument which was filled.
        exchange - The exchange where the order was filled.
        quantity - The filled quantity.
        direction - The direction of fill ('BUY' or 'SELL')
        fill_cost - The holdings value in dollars.
        commission - An optional commission sent from IB.
        """

        self.type = 'FILL'
        self.timeindex = timeindex
        self.symbol = symbol
        self.exchange = exchange
        self.quantity = quantity
        self.direction = direction
        self.fill_cost = fill_cost

        # Calculate commission
        if commission is None:
            self.commission = self.calculate_ib_commission()
        else:
            self.commission = commission

    def calculate_ib_commission(self):
        """
        Calculates the fees of trading based on an Interactive
        Brokers fee structure for API, in USD.

        This does not include exchange or ECN fees.

        Based on "US API Directed Orders":
        https://www.interactivebrokers.com/en/index.php?
        f=commission&p=stocks2
        """
        full_cost = 1.3
        if self.quantity <= 500:
            full_cost = max(1.3, 0.013 * self.quantity)
        else: # Greater than 500
            full_cost = max(1.3, 0.008 * self.quantity)
        return full_cost
```

14.2.2 Data Handler

One of the goals of an event-driven trading system is to minimise duplication of code between the backtesting element and the live execution element. Ideally it would be optimal to utilise the same signal generation methodology and portfolio management components for both historical

testing and live trading. In order for this to work the Strategy object which generates the Signals, and the Portfolio object which provides Orders based on them, must utilise an identical interface to a market feed for both historic and live running.

This motivates the concept of a class hierarchy based on a DataHandler object, which gives all subclasses an interface for providing market data to the remaining components within the system. In this way any subclass data handler can be "swapped out", without affecting strategy or portfolio calculation.

Specific example subclasses could include HistoricCSVDataHandler, QuandlDataHandler, SecuritiesMasterDataHandler, InteractiveBrokersMarketFeedDataHandler etc. In this chapter we are only going to consider the creation of a historic CSV data handler, which will load intraday CSV data for equities in an Open-Low-High-Close-Volume-OpenInterest set of bars. This can then be used to "drip feed" on a bar-by-bar basis the data into the Strategy and Portfolio classes on every heartbeat of the system, thus avoiding lookahead bias.

The first task is to import the necessary libraries. Specifically it will be necessary to import pandas and the abstract base class tools. Since the DataHandler generates MarketEvents, event.py is also needed as described above.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# data.py

from __future__ import print_function

from abc import ABCMeta, abstractmethod
import datetime
import os, os.path

import numpy as np
import pandas as pd

from event import MarketEvent
```

The DataHandler is an abstract base class (ABC), which means that it is impossible to instantiate an instance directly. Only subclasses may be instantiated. The rationale for this is that the ABC provides an interface that all subsequent DataHandler subclasses must adhere to thereby ensuring compatibility with other classes that communicate with them.

We make use of the __metaclass__ property to let Python know that this is an ABC. In addition we use the @abstractmethod decorator to let Python know that the method will be overridden in subclasses (this is identical to a *pure virtual method* in C++).

There are six methods listed for the class. The first two methods, get_latest_bar and get_latest_bars, are used to retrieve a recent subset of the historical trading bars from a stored list of such bars. These methods come in handy within the Strategy and Portfolio classes, due to the need to constantly be aware of current market prices and volumes.

The following method, get_latest_bar_datetime, simply returns a Python datetime object that represents the timestamp of the bar (e.g. a date for daily bars or a minute-resolution object for minutely bars).

The following two methods, get_latest_bar_value and get_latest_bar_values, are convenience methods used to retrieve individual values from a particular bar, or list of bars. For instance it is often the case that a strategy is only interested in closing prices. In this instance we can use these methods to return a list of floating point values representing the closing prices of previous bars, rather than having to obtain it from the list of bar objects. This generally increases efficiency of strategies that utilise a "lookback window", such as those involving regressions.

The final method, update_bars, provides a "drip feed" mechanism for placing bar information on a new data structure that strictly prohibits lookahead bias. This is one of the key differences between an event-driven backtesting system and one based on vectorisation. Notice that exceptions will be raised if an attempted instantiation of the class occurs:

```
# data.py

class DataHandler(object):
    """
    DataHandler is an abstract base class providing an interface for
    all subsequent (inherited) data handlers (both live and historic).

    The goal of a (derived) DataHandler object is to output a generated
    set of bars (OHLCVI) for each symbol requested.

    This will replicate how a live strategy would function as current
    market data would be sent "down the pipe". Thus a historic and live
    system will be treated identically by the rest of the backtesting suite.
    """

    __metaclass__ = ABCMeta

    @abstractmethod
    def get_latest_bar(self, symbol):
        """
        Returns the last bar updated.
        """
        raise NotImplementedError("Should implement get_latest_bar()")

    @abstractmethod
    def get_latest_bars(self, symbol, N=1):
        """
        Returns the last N bars updated.
        """
        raise NotImplementedError("Should implement get_latest_bars()")

    @abstractmethod
    def get_latest_bar_datetime(self, symbol):
        """
        Returns a Python datetime object for the last bar.
        """
        raise NotImplementedError("Should implement
            get_latest_bar_datetime()")

    @abstractmethod
    def get_latest_bar_value(self, symbol, val_type):
        """
        Returns one of the Open, High, Low, Close, Volume or OI
        from the last bar.
        """
        raise NotImplementedError("Should implement
            get_latest_bar_value()")

    @abstractmethod
    def get_latest_bars_values(self, symbol, val_type, N=1):
        """
        Returns the last N bar values from the
        latest_symbol list, or N-k if less available.
        """
        raise NotImplementedError("Should implement
            get_latest_bars_values()")
```

```
    @abstractmethod
    def update_bars(self):
        """
        Pushes the latest bars to the bars_queue for each symbol
        in a tuple OHLCVI format: (datetime, open, high, low,
        close, volume, open interest).
        """
        raise NotImplementedError("Should implement update_bars()")
```

In order to create a backtesting system based on historical data we need to consider a mechanism for importing data via common sources. We've discussed the benefits of a Securities Master Database in previous chapters. Thus a good candidate for making a DataHandler class would be to couple it with such a database.

However, for clarity in this chapter, I want to discuss a simpler mechanism, that of importing (potentially large) comma-separated variable (CSV) files. This will allow us to focus on the mechanics of creating the DataHandler, rather than be concerned with the "boilerplate" code of connecting to a database and using SQL queries to grab data.

Thus we are going to define the HistoricCSVDataHandler subclass, which is designed to process multiple CSV files, one for each traded symbol, and convert these into a dictionary of pandas DataFrames that can be accessed by the previously mentioned bar methods.

The data handler requires a few parameters, namely an Event Queue on which to push MarketEvent information to, the absolute path of the CSV files and a list of symbols. Here is the initialisation of the class:

```
# data.py

class HistoricCSVDataHandler(DataHandler):
    """
    HistoricCSVDataHandler is designed to read CSV files for
    each requested symbol from disk and provide an interface
    to obtain the "latest" bar in a manner identical to a live
    trading interface.
    """

    def __init__(self, events, csv_dir, symbol_list):
        """
        Initialises the historic data handler by requesting
        the location of the CSV files and a list of symbols.

        It will be assumed that all files are of the form
        'symbol.csv', where symbol is a string in the list.

        Parameters:
        events - The Event Queue.
        csv_dir - Absolute directory path to the CSV files.
        symbol_list - A list of symbol strings.
        """
        self.events = events
        self.csv_dir = csv_dir
        self.symbol_list = symbol_list

        self.symbol_data = {}
        self.latest_symbol_data = {}
        self.continue_backtest = True

        self._open_convert_csv_files()
```

The handler is will look for files in the absolute directory csv_dir and try to open them with the format of "SYMBOL.csv", where SYMBOL is the ticker symbol (such as GOOG or AAPL). The format of the files matches that provided by Yahoo Finance, but is easily modified to handle additional data formats, such as those provided by Quandl or DTN IQFeed. The opening of the files is handled by the _open_convert_csv_files method below.

One of the benefits of using pandas as a datastore internally within the HistoricCSVDataHandler is that the indexes of all symbols being tracked can be merged together. This allows missing data points to be padded forward, backward or interpolated within these gaps such that tickers can be compared on a bar-to-bar basis. This is necessary for mean-reverting strategies, for instance. Notice the use of the union and reindex methods when combining the indexes for all symbols:

```
# data.py

    def _open_convert_csv_files(self):
        """
        Opens the CSV files from the data directory, converting
        them into pandas DataFrames within a symbol dictionary.

        For this handler it will be assumed that the data is
        taken from Yahoo. Thus its format will be respected.
        """
        comb_index = None
        for s in self.symbol_list:
            # Load the CSV file with no header information, indexed on date
            self.symbol_data[s] = pd.io.parsers.read_csv(
                os.path.join(self.csv_dir, '%s.csv' % s),
                header=0, index_col=0, parse_dates=True,
                names=[
                    'datetime', 'open', 'high',
                    'low', 'close', 'volume', 'adj_close'
                ]
            ).sort()

            # Combine the index to pad forward values
            if comb_index is None:
                comb_index = self.symbol_data[s].index
            else:
                comb_index.union(self.symbol_data[s].index)

            # Set the latest symbol_data to None
            self.latest_symbol_data[s] = []

        # Reindex the dataframes
        for s in self.symbol_list:
            self.symbol_data[s] = self.symbol_data[s].\
                reindex(index=comb_index, method='pad').iterrows()
```

The _get_new_bar method creates a generator to provide a new bar. This means that subsequent calls to the method will *yield* a new bar until the end of the symbol data is reached:

```
# data.py

    def _get_new_bar(self, symbol):
        """
        Returns the latest bar from the data feed.
        """
        for b in self.symbol_data[symbol]:
```

```
            yield b
```

The first abstract methods from DataHandler to be implemented are get_latest_bar and get_latest_bars. These method simply provide either a bar or list of the last N bars from the latest_symbol_data structure:

```
# data.py

    def get_latest_bar(self, symbol):
        """
        Returns the last bar from the latest_symbol list.
        """
        try:
            bars_list = self.latest_symbol_data[symbol]
        except KeyError:
            print("That symbol is not available in the historical data set.")
            raise
        else:
            return bars_list[-1]

    def get_latest_bars(self, symbol, N=1):
        """
        Returns the last N bars from the latest_symbol list,
        or N-k if less available.
        """
        try:
            bars_list = self.latest_symbol_data[symbol]
        except KeyError:
            print("That symbol is not available in the historical data set.")
            raise
        else:
            return bars_list[-N:]
```

The next method, get_latest_bar_datetime, queries the latest bar for a datetime object representing the "last market price":

```
    def get_latest_bar_datetime(self, symbol):
        """
        Returns a Python datetime object for the last bar.
        """
        try:
            bars_list = self.latest_symbol_data[symbol]
        except KeyError:
            print("That symbol is not available in the historical data set.")
            raise
        else:
            return bars_list[-1][0]
```

The next two methods being implemented are get_latest_bar_value and get_latest_bar_values. Both methods make use of the Python getattr function, which queries an object to see if a particular attribute exists on an object. Thus we can pass a string such as "open" or "close" to getattr and obtain the value direct from the bar, thus making the method more flexible. This stops us having to write methods of the type get_latest_bar_close, for instance:

```
    def get_latest_bar_value(self, symbol, val_type):
        """
        Returns one of the Open, High, Low, Close, Volume or OI
        values from the pandas Bar series object.
        """
```

```
        try:
            bars_list = self.latest_symbol_data[symbol]
        except KeyError:
            print("That symbol is not available in the historical data set.")
            raise
        else:
            return getattr(bars_list[-1][1], val_type)

    def get_latest_bars_values(self, symbol, val_type, N=1):
        """
        Returns the last N bar values from the
        latest_symbol list, or N-k if less available.
        """
        try:
            bars_list = self.get_latest_bars(symbol, N)
        except KeyError:
            print("That symbol is not available in the historical data set.")
            raise
        else:
            return np.array([getattr(b[1], val_type) for b in bars_list])
```

The final method, update_bars, is the second abstract method from DataHandler. It simply generates a MarketEvent that gets added to the queue as it appends the latest bars to the latest_symbol_data dictionary:

```
# data.py

    def update_bars(self):
        """
        Pushes the latest bar to the latest_symbol_data structure
        for all symbols in the symbol list.
        """
        for s in self.symbol_list:
            try:
                bar = next(self._get_new_bar(s))
            except StopIteration:
                self.continue_backtest = False
            else:
                if bar is not None:
                    self.latest_symbol_data[s].append(bar)
        self.events.put(MarketEvent())
```

Thus we have a DataHandler-derived object, which is used by the remaining components to keep track of market data. The Strategy, Portfolio and ExecutionHandler objects all require the current market data thus it makes sense to centralise it to avoid duplication of storage between these classes.

14.2.3 Strategy

A `Strategy` object encapsulates all calculation on market data that generate *advisory* signals to a `Portfolio` object. Thus all of the "strategy logic" resides within this class. I have opted to separate out the Strategy and Portfolio objects for this backtester, since I believe this is more amenable to the situation of multiple strategies feeding "ideas" to a larger Portfolio, which then can handle its own risk (such as sector allocation, leverage). In higher frequency trading, the strategy and portfolio concepts will be tightly coupled and extremely hardware dependent. This is well beyond the scope of this chapter, however!

At this stage in the event-driven backtester development there is no concept of an *indicator* or *filter*, such as those found in technical trading. These are also good candidates for creating

a class hierarchy but are beyond the scope of this chapter. Thus such mechanisms will be used directly in derived Strategy objects.

The strategy hierarchy is relatively simple as it consists of an abstract base class with a single pure virtual method for generating `SignalEvent` objects. In order to create the `Strategy` hierarchy it is necessary to import NumPy, pandas, the `Queue` object (which has become `queue` in Python 3), abstract base class tools and the `SignalEvent`:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# strategy.py

from __future__ import print_function

from abc import ABCMeta, abstractmethod
import datetime
try:
    import Queue as queue
except ImportError:
    import queue

import numpy as np
import pandas as pd

from event import SignalEvent
```

The `Strategy` abstract base class simply defines a pure virtual `calculate_signals` method. In derived classes this is used to handle the generation of `SignalEvent` objects based on market data updates:

```
# strategy.py

class Strategy(object):
    """
    Strategy is an abstract base class providing an interface for
    all subsequent (inherited) strategy handling objects.

    The goal of a (derived) Strategy object is to generate Signal
    objects for particular symbols based on the inputs of Bars
    (OHLCV) generated by a DataHandler object.

    This is designed to work both with historic and live data as
    the Strategy object is agnostic to where the data came from,
    since it obtains the bar tuples from a queue object.
    """

    __metaclass__ = ABCMeta

    @abstractmethod
    def calculate_signals(self):
        """
        Provides the mechanisms to calculate the list of signals.
        """
        raise NotImplementedError("Should implement calculate_signals()")
```

14.2.4 Portfolio

This section describes a `Portfolio` object that keeps track of the positions within a portfolio and generates orders of a fixed quantity of stock based on signals. More sophisticated portfolio objects could include risk management and position sizing tools (such as the Kelly Criterion). In fact, in the following chapters we will add such tools to some of our trading strategies to see how they compare to a more "naive" portfolio approach.

The portfolio order management system is possibly the most complex component of an event-driven backtester. Its role is to keep track of all current market positions as well as the market value of the positions (known as the "holdings"). This is simply an estimate of the liquidation value of the position and is derived in part from the data handling facility of the backtester.

In addition to the positions and holdings management the portfolio must also be aware of risk factors and position sizing techniques in order to optimise orders that are sent to a brokerage or other form of market access.

Unfortunately, Portfolio and Order Management Systems (OMS) can become rather complex! Thus I've made a decision here to keep the Portfolio object relatively straightforward, so that you can understand the key ideas and how they are implemented. The nature of an object-oriented design is that it allows, in a natural way, the extension to more complex situations later on.

Continuing in the vein of the `Event` class hierarchy a `Portfolio` object must be able to handle `SignalEvent` objects, generate `OrderEvent` objects and interpret `FillEvent` objects to update positions. Thus it is no surprise that the `Portfolio` objects are often the largest component of event-driven systems, in terms of lines of code (LOC).

We create a new file `portfolio.py` and import the necessary libraries. These are the same as most of the other class implementations, with the exception that Portfolio is NOT going to be an abstract base class. Instead it will be normal base class. This means that it can be instantiated and thus is useful as a "first go" Portfolio object when testing out new strategies. Other Portfolios can be derived from it and override sections to add more complexity.

For completeness, here is the performance.py file:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# performance.py

from __future__ import print_function

import numpy as np
import pandas as pd


def create_sharpe_ratio(returns, periods=252):
    """
    Create the Sharpe ratio for the strategy, based on a
    benchmark of zero (i.e. no risk-free rate information).

    Parameters:
    returns - A pandas Series representing period percentage returns.
    periods - Daily (252), Hourly (252*6.5), Minutely(252*6.5*60) etc.
    """
    return np.sqrt(periods) * (np.mean(returns)) / np.std(returns)


def create_drawdowns(pnl):
    """
    Calculate the largest peak-to-trough drawdown of the PnL curve
    as well as the duration of the drawdown. Requires that the
    pnl_returns is a pandas Series.
```

```
    Parameters:
    pnl - A pandas Series representing period percentage returns.

    Returns:
    drawdown, duration - Highest peak-to-trough drawdown and duration.
    """

    # Calculate the cumulative returns curve
    # and set up the High Water Mark
    hwm = [0]

    # Create the drawdown and duration series
    idx = pnl.index
    drawdown = pd.Series(index = idx)
    duration = pd.Series(index = idx)

    # Loop over the index range
    for t in range(1, len(idx)):
        hwm.append(max(hwm[t-1], pnl[t]))
        drawdown[t]= (hwm[t]-pnl[t])
        duration[t]= (0 if drawdown[t] == 0 else duration[t-1]+1)
    return drawdown, drawdown.max(), duration.max()
```

Here is the import listing for the Portfolio.py file. We need to import the `floor` function from the `math` library in order to generate integer-valued order sizes. We also need the `FillEvent` and `OrderEvent` objects since the `Portfolio` handles both. Notice also that we are adding two additional functions, create_sharpe_ratio and create_drawdowns, both from the performance.py file described above.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# portfolio.py

from __future__ import print_function

import datetime
from math import floor
try:
    import Queue as queue
except ImportError:
    import queue

import numpy as np
import pandas as pd

from event import FillEvent, OrderEvent
from performance import create_sharpe_ratio, create_drawdowns
```

The initialisation of the Portfolio object requires access to the bars DataHandler, the events Event Queue, a start datetime stamp and an initial capital value (defaulting to 100,000 USD).

The Portfolio is designed to handle position sizing and current holdings, but will carry out trading orders in a "dumb" manner by simply sending them directly to the brokerage with a predetermined fixed quantity size, irrespective of cash held. These are all unrealistic assumptions, but they help to outline how a portfolio order management system (OMS) functions in an event-driven fashion.

The portfolio contains the `all_positions` and `current_positions` members. The former stores a list of all previous *positions* recorded at the timestamp of a market data event. A position is simply the quantity of the asset held. Negative positions mean the asset has been shorted. The latter current_positions dictionary stores contains the current positions for the last market bar update, for each symbol.

In addition to the positions data the portfolio stores *holdings*, which describe the current market *value* of the positions held. "Current market value" in this instance means the closing price obtained from the current market bar, which is clearly an approximation, but is reasonable enough for the time being. `all_holdings` stores the historical list of all symbol holdings, while `current_holdings` stores the most up to date dictionary of all symbol holdings values:

```
# portfolio.py

class Portfolio(object):
    """
    The Portfolio class handles the positions and market
    value of all instruments at a resolution of a "bar",
    i.e. secondly, minutely, 5-min, 30-min, 60 min or EOD.

    The positions DataFrame stores a time-index of the
    quantity of positions held.

    The holdings DataFrame stores the cash and total market
    holdings value of each symbol for a particular
    time-index, as well as the percentage change in
    portfolio total across bars.
    """

    def __init__(self, bars, events, start_date, initial_capital=100000.0):
        """
        Initialises the portfolio with bars and an event queue.
        Also includes a starting datetime index and initial capital
        (USD unless otherwise stated).

        Parameters:
        bars - The DataHandler object with current market data.
        events - The Event Queue object.
        start_date - The start date (bar) of the portfolio.
        initial_capital - The starting capital in USD.
        """
        self.bars = bars
        self.events = events
        self.symbol_list = self.bars.symbol_list
        self.start_date = start_date
        self.initial_capital = initial_capital

        self.all_positions = self.construct_all_positions()
        self.current_positions = dict( (k,v) for k, v in \
            [(s, 0) for s in self.symbol_list] )

        self.all_holdings = self.construct_all_holdings()
        self.current_holdings = self.construct_current_holdings()
```

The following method, `construct_all_positions`, simply creates a dictionary for each symbol, sets the value to zero for each and then adds a datetime key, finally adding it to a list. It uses a dictionary comprehension, which is similar in spirit to a list comprehension:

```
# portfolio.py
```

```
    def construct_all_positions(self):
        """
        Constructs the positions list using the start_date
        to determine when the time index will begin.
        """
        d = dict( (k,v) for k, v in [(s, 0) for s in self.symbol_list] )
        d['datetime'] = self.start_date
        return [d]
```

The `construct_all_holdings` method is similar to the above but adds extra keys for cash, commission and total, which respectively represent the spare cash in the account after any purchases, the cumulative commission accrued and the total account equity including cash and any open positions. Short positions are treated as negative. The starting cash and total account equity are both set to the initial capital.

In this manner there are separate "accounts" for each symbol, the "cash on hand", the "commission" paid (Interactive Broker fees) and a "total" portfolio value. Clearly this does not take into account margin requirements or shorting constraints, but is sufficient to give you a flavour of how such an OMS is created:

```
# portfolio.py

    def construct_all_holdings(self):
        """
        Constructs the holdings list using the start_date
        to determine when the time index will begin.
        """
        d = dict( (k,v) for k, v in [(s, 0.0) for s in self.symbol_list] )
        d['datetime'] = self.start_date
        d['cash'] = self.initial_capital
        d['commission'] = 0.0
        d['total'] = self.initial_capital
        return [d]
```

The following method, `construct_current_holdings` is almost identical to the method above except that it doesn't wrap the dictionary in a list, because it is only creating a single entry:

```
# portfolio.py

    def construct_current_holdings(self):
        """
        This constructs the dictionary which will hold the instantaneous
        value of the portfolio across all symbols.
        """
        d = dict( (k,v) for k, v in [(s, 0.0) for s in self.symbol_list] )
        d['cash'] = self.initial_capital
        d['commission'] = 0.0
        d['total'] = self.initial_capital
        return d
```

On every *heartbeat*, that is every time new market data is requested from the `DataHandler` object, the portfolio must update the current market value of all the positions held. In a live trading scenario this information can be downloaded and parsed directly from the brokerage, but for a backtesting implementation it is necessary to calculate these values manually from the bars DataHandler.

Unfortunately there is no such as thing as the "current market value" due to bid/ask spreads and liquidity issues. Thus it is necessary to estimate it by multiplying the quantity of the asset held by a particular approximate "price". The approach I have taken here is to use the closing

price of the last bar received. For an intraday strategy this is relatively realistic. For a daily strategy this is less realistic as the opening price can differ substantially from the closing price.

The method `update_timeindex` handles the new holdings tracking. It firstly obtains the latest prices from the market data handler and creates a new dictionary of symbols to represent the current positions, by setting the "new" positions equal to the "current" positions.

The current positions are only modified when a `FillEvent` is obtained, which is handled later on in the portfolio code. The method then appends this set of current positions to the `all_positions` list.

The holdings are then updated in a similar manner, with the exception that the market value is recalculated by multiplying the current positions count with the closing price of the latest bar. Finally the new holdings are appended to `all_holdings`:

```
# portfolio.py

    def update_timeindex(self, event):
        """
        Adds a new record to the positions matrix for the current
        market data bar. This reflects the PREVIOUS bar, i.e. all
        current market data at this stage is known (OHLCV).

        Makes use of a MarketEvent from the events queue.
        """
        latest_datetime = self.bars.get_latest_bar_datetime(
            self.symbol_list[0]
        )

        # Update positions
        # ================
        dp = dict( (k,v) for k, v in [(s, 0) for s in self.symbol_list] )
        dp['datetime'] = latest_datetime

        for s in self.symbol_list:
            dp[s] = self.current_positions[s]

        # Append the current positions
        self.all_positions.append(dp)

        # Update holdings
        # ===============
        dh = dict( (k,v) for k, v in [(s, 0) for s in self.symbol_list] )
        dh['datetime'] = latest_datetime
        dh['cash'] = self.current_holdings['cash']
        dh['commission'] = self.current_holdings['commission']
        dh['total'] = self.current_holdings['cash']

        for s in self.symbol_list:
            # Approximation to the real value
            market_value = self.current_positions[s] * \
                self.bars.get_latest_bar_value(s, "adj_close")
            dh[s] = market_value
            dh['total'] += market_value

        # Append the current holdings
        self.all_holdings.append(dh)
```

The method `update_positions_from_fill` determines whether a `FillEvent` is a Buy or a Sell and then updates the `current_positions` dictionary accordingly by adding/subtracting

the correct quantity of shares:

```
# portfolio.py

    def update_positions_from_fill(self, fill):
        """
        Takes a Fill object and updates the position matrix to
        reflect the new position.

        Parameters:
        fill - The Fill object to update the positions with.
        """
        # Check whether the fill is a buy or sell
        fill_dir = 0
        if fill.direction == 'BUY':
            fill_dir = 1
        if fill.direction == 'SELL':
            fill_dir = -1

        # Update positions list with new quantities
        self.current_positions[fill.symbol] += fill_dir*fill.quantity
```

The corresponding `update_holdings_from_fill` is similar to the above method but updates the *holdings* values instead. In order to simulate the cost of a fill, the following method does not use the cost associated from the `FillEvent`. Why is this? Simply put, in a backtesting environment the fill cost is actually unknown (the *market impact* and the *depth of book* are unknown) and thus is must be estimated.

Thus the fill cost is set to the the "current market price", which is the closing price of the last bar. The holdings for a particular symbol are then set to be equal to the fill cost multiplied by the transacted quantity. For most lower frequency trading strategies in liquid markets this is a reasonable approximation, but at high frequency these issues will need to be considered in a production backtest and live trading engine.

Once the fill cost is known the current holdings, cash and total values can all be updated. The cumulative commission is also updated:

```
# portfolio.py

    def update_holdings_from_fill(self, fill):
        """
        Takes a Fill object and updates the holdings matrix to
        reflect the holdings value.

        Parameters:
        fill - The Fill object to update the holdings with.
        """
        # Check whether the fill is a buy or sell
        fill_dir = 0
        if fill.direction == 'BUY':
            fill_dir = 1
        if fill.direction == 'SELL':
            fill_dir = -1

        # Update holdings list with new quantities
        fill_cost = self.bars.get_latest_bar_value(fill.symbol, "adj_close")
        cost = fill_dir * fill_cost * fill.quantity
        self.current_holdings[fill.symbol] += cost
        self.current_holdings['commission'] += fill.commission
        self.current_holdings['cash'] -= (cost + fill.commission)
```

```
        self.current_holdings['total'] -= (cost + fill.commission)
```

The pure virtual `update_fill` method from the `Portfolio` class is implemented here. It simply executes the two preceding methods, `update_positions_from_fill` and `update_holdings_from_fill`, upon receipt of a fill event:

```
# portfolio.py

    def update_fill(self, event):
        """
        Updates the portfolio current positions and holdings
        from a FillEvent.
        """
        if event.type == 'FILL':
            self.update_positions_from_fill(event)
            self.update_holdings_from_fill(event)
```

While the `Portfolio` object must handle `FillEvent`s, it must also take care of generating `OrderEvent`s upon the receipt of one or more `SignalEvent`s.

The `generate_naive_order` method simply takes a signal to go long or short an asset, sending an order to do so for 100 shares of such an asset. Clearly 100 is an arbitrary value, and will clearly depend upon the portfolio total equity in a production simulation.

In a realistic implementation this value will be determined by a risk management or position sizing overlay. However, this is a simplistic `Portfolio` and so it "naively" sends all orders directly from the signals, without a risk system.

The method handles longing, shorting and exiting of a position, based on the current quantity and particular symbol. Corresponding `OrderEvent` objects are then generated:

```
# portfolio.py

    def generate_naive_order(self, signal):
        """
        Simply files an Order object as a constant quantity
        sizing of the signal object, without risk management or
        position sizing considerations.

        Parameters:
        signal - The tuple containing Signal information.
        """
        order = None

        symbol = signal.symbol
        direction = signal.signal_type
        strength = signal.strength

        mkt_quantity = 100
        cur_quantity = self.current_positions[symbol]
        order_type = 'MKT'

        if direction == 'LONG' and cur_quantity == 0:
            order = OrderEvent(symbol, order_type, mkt_quantity, 'BUY')
        if direction == 'SHORT' and cur_quantity == 0:
            order = OrderEvent(symbol, order_type, mkt_quantity, 'SELL')

        if direction == 'EXIT' and cur_quantity > 0:
            order = OrderEvent(symbol, order_type, abs(cur_quantity), 'SELL')
        if direction == 'EXIT' and cur_quantity < 0:
            order = OrderEvent(symbol, order_type, abs(cur_quantity), 'BUY')
```

```
        return order
```

The `update_signal` method simply calls the above method and adds the generated order to the events queue:

```
# portfolio.py

    def update_signal(self, event):
        """
        Acts on a SignalEvent to generate new orders
        based on the portfolio logic.
        """
        if event.type == 'SIGNAL':
            order_event = self.generate_naive_order(event)
            self.events.put(order_event)
```

The penultimate method in the `Portfolio` is the generation of an equity curve. This simply creates a returns stream, useful for performance calculations, and then normalises the equity curve to be percentage based. Thus the account initial size is equal to 1.0, as opposed to the absolute dollar amount:

```
# portfolio.py

    def create_equity_curve_dataframe(self):
        """
        Creates a pandas DataFrame from the all_holdings
        list of dictionaries.
        """
        curve = pd.DataFrame(self.all_holdings)
        curve.set_index('datetime', inplace=True)
        curve['returns'] = curve['total'].pct_change()
        curve['equity_curve'] = (1.0+curve['returns']).cumprod()
        self.equity_curve = curve
```

The final method in the `Portfolio` is the output of the equity curve and various performance statistics related to the strategy. The final line outputs a file, equity.csv, to the same directory as the code, which can loaded into a Matplotlib Python script (or a spreadsheet such as MS Excel or LibreOffice Calc) for subsequent analysis.

Note that the Drawdown Duration is given in terms of the absolute number of "bars" that the drawdown carried on for, as opposed to a particular timeframe.

```
    def output_summary_stats(self):
        """
        Creates a list of summary statistics for the portfolio.
        """
        total_return = self.equity_curve['equity_curve'][-1]
        returns = self.equity_curve['returns']
        pnl = self.equity_curve['equity_curve']

        sharpe_ratio = create_sharpe_ratio(returns, periods=252*60*6.5)
        drawdown, max_dd, dd_duration = create_drawdowns(pnl)
        self.equity_curve['drawdown'] = drawdown

        stats = [("Total Return", "%0.2f%%" % \
                    ((total_return - 1.0) * 100.0)),
                 ("Sharpe Ratio", "%0.2f" % sharpe_ratio),
                 ("Max Drawdown", "%0.2f%%" % (max_dd * 100.0)),
                 ("Drawdown Duration", "%d" % dd_duration)]
```

```
        self.equity_curve.to_csv('equity.csv')
        return stats
```

The `Portfolio` object is the most complex aspect of the entire event-driven backtest system. The implementation here, while intricate, is relatively elementary in its handling of positions.

14.2.5 Execution Handler

In this section we will study the execution of trade orders by creating a class hierarchy that will represent a simulated order handling mechanism and ultimately tie into a brokerage or other means of market connectivity.

The `ExecutionHandler` described here is exceedingly simple, since it fills all orders at the current market price. This is highly unrealistic, but serves as a good baseline for improvement.

As with the previous abstract base class hierarchies, we must import the necessary properties and decorators from the `abc` library. In addition we need to import the `FillEvent` and `OrderEvent`:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# execution.py

from __future__ import print_function

from abc import ABCMeta, abstractmethod
import datetime
try:
    import Queue as queue
except ImportError:
    import queue

from event import FillEvent, OrderEvent
```

The `ExecutionHandler` is similar to previous abstract base classes and simply has one pure virtual method, `execute_order`:

```
# execution.py

class ExecutionHandler(object):
    """
    The ExecutionHandler abstract class handles the interaction
    between a set of order objects generated by a Portfolio and
    the ultimate set of Fill objects that actually occur in the
    market.

    The handlers can be used to subclass simulated brokerages
    or live brokerages, with identical interfaces. This allows
    strategies to be backtested in a very similar manner to the
    live trading engine.
    """

    __metaclass__ = ABCMeta

    @abstractmethod
    def execute_order(self, event):
        """
        Takes an Order event and executes it, producing
        a Fill event that gets placed onto the Events queue.
```

```
        Parameters:
        event - Contains an Event object with order information.
        """
        raise NotImplementedError("Should implement execute_order()")
```

In order to backtest strategies we need to simulate how a trade will be transacted. The simplest possible implementation is to assume all orders are filled at the current market price for all quantities. This is clearly extremely unrealistic and a big part of improving backtest realism will come from designing more sophisticated models of slippage and market impact.

Note that the `FillEvent` is given a value of `None` for the `fill_cost` (see the penultimate line in `execute_order`) as we have already taken care of the cost of fill in the `Portfolio` object described above. In a more realistic implementation we would make use of the "current" market data value to obtain a realistic fill cost.

I have simply utilised ARCA as the exchange although for backtesting purposes this is purely a string placeholder. In a live execution environment this venue dependence would be far more important:

```
# execution.py

class SimulatedExecutionHandler(ExecutionHandler):
    """
    The simulated execution handler simply converts all order
    objects into their equivalent fill objects automatically
    without latency, slippage or fill-ratio issues.

    This allows a straightforward "first go" test of any strategy,
    before implementation with a more sophisticated execution
    handler.
    """

    def __init__(self, events):
        """
        Initialises the handler, setting the event queues
        up internally.

        Parameters:
        events - The Queue of Event objects.
        """
        self.events = events

    def execute_order(self, event):
        """
        Simply converts Order objects into Fill objects naively,
        i.e. without any latency, slippage or fill ratio problems.

        Parameters:
        event - Contains an Event object with order information.
        """
        if event.type == 'ORDER':
            fill_event = FillEvent(
                datetime.datetime.utcnow(), event.symbol,
                'ARCA', event.quantity, event.direction, None
            )
            self.events.put(fill_event)
```

14.2.6 Backtest

We are now in a position to create the Backtest class hierarchy. The Backtest object encapsulates the event-handling logic and essentially ties together all of the other classes that we have discussed above.

The Backtest object is designed to carry out a nested while-loop event-driven system in order to handle the events placed on the Event Queue object. The outer while-loop is known as the "heartbeat loop" and decides the temporal resolution of the backtesting system. In a live environment this value will be a positive number, such as 600 seconds (every ten minutes). Thus the market data and positions will only be updated on this timeframe.

For the backtester described here the "heartbeat" can be set to zero, irrespective of the strategy frequency, since the data is already available by virtue of the fact it is historical!

We can run the backtest at whatever speed we like, since the event-driven system is agnostic to *when* the data became available, so long as it has an associated timestamp. Hence I've only included it to demonstrate how a live trading engine would function. The outer loop thus ends once the DataHandler lets the Backtest object know, by using a boolean continue_backtest attribute.

The inner while-loop actually processes the signals and sends them to the correct component depending upon the event type. Thus the Event Queue is continually being populated and depopulated with events. This is what it means for a system to be *event-driven*.

The first task is to import the necessary libraries. We import pprint ("pretty-print"), because we want to display the stats in an output-friendly manner:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# backtest.py

from __future__ import print_function

import datetime
import pprint
try:
    import Queue as queue
except ImportError:
    import queue
import time
```

The initialisation of the Backtest object requires the CSV directory, the full symbol list of traded symbols, the initial capital, the heartbeat time in milliseconds, the start datetime stamp of the backtest as well as the DataHandler, ExecutionHandler, Portfolio and Strategy objects. A Queue is used to hold the events. The signals, orders and fills are counted:

```
# backtest.py

class Backtest(object):
    """
    Enscapsulates the settings and components for carrying out
    an event-driven backtest.
    """

    def __init__(
        self, csv_dir, symbol_list, initial_capital,
        heartbeat, start_date, data_handler,
        execution_handler, portfolio, strategy
    ):
        """
        Initialises the backtest.
```

```
        Parameters:
        csv_dir - The hard root to the CSV data directory.
        symbol_list - The list of symbol strings.
        intial_capital - The starting capital for the portfolio.
        heartbeat - Backtest "heartbeat" in seconds
        start_date - The start datetime of the strategy.
        data_handler - (Class) Handles the market data feed.
        execution_handler - (Class) Handles the orders/fills for trades.
        portfolio - (Class) Keeps track of portfolio current
            and prior positions.
        strategy - (Class) Generates signals based on market data.
        """
        self.csv_dir = csv_dir
        self.symbol_list = symbol_list
        self.initial_capital = initial_capital
        self.heartbeat = heartbeat
        self.start_date = start_date

        self.data_handler_cls = data_handler
        self.execution_handler_cls = execution_handler
        self.portfolio_cls = portfolio
        self.strategy_cls = strategy

        self.events = queue.Queue()

        self.signals = 0
        self.orders = 0
        self.fills = 0
        self.num_strats = 1

        self._generate_trading_instances()
```

The first method, _generate_trading_instances, attaches all of the trading objects (DataHandler, Strategy, Portfolio and ExecutionHandler) to various internal members:

```
# backtest.py

    def _generate_trading_instances(self):
        """
        Generates the trading instance objects from
        their class types.
        """
        print(
            "Creating DataHandler, Strategy, Portfolio and ExecutionHandler"
        )
        self.data_handler = self.data_handler_cls(self.events, self.csv_dir,
            self.symbol_list)
        self.strategy = self.strategy_cls(self.data_handler, self.events)
        self.portfolio = self.portfolio_cls(self.data_handler, self.events,
                                            self.start_date,
                                            self.initial_capital)
        self.execution_handler = self.execution_handler_cls(self.events)
```

The _run_backtest method is where the signal handling of the Backtest engine is carried out. As described above there are two while loops, one nested within another. The outer keeps track of the heartbeat of the system, while the inner checks if there is an event in the Queue object, and acts on it by calling the appropriate method on the necessary object.

For a MarketEvent, the Strategy object is told to recalculate new signals, while the Portfolio object is told to reindex the time. If a SignalEvent object is received the Portfolio is told to handle the new signal and convert it into a set of OrderEvents, if appropriate. If an OrderEvent is received the ExecutionHandler is sent the order to be transmitted to the broker (if in a real trading setting). Finally, if a FillEvent is received, the Portfolio will update itself to be aware of the new positions:

```
# backtest.py

    def _run_backtest(self):
        """
        Executes the backtest.
        """
        i = 0
        while True:
            i += 1
            print i
            # Update the market bars
            if self.data_handler.continue_backtest == True:
                self.data_handler.update_bars()
            else:
                break

            # Handle the events
            while True:
                try:
                    event = self.events.get(False)
                except queue.Empty:
                    break
                else:
                    if event is not None:
                        if event.type == 'MARKET':
                            self.strategy.calculate_signals(event)
                            self.portfolio.update_timeindex(event)

                        elif event.type == 'SIGNAL':
                            self.signals += 1
                            self.portfolio.update_signal(event)

                        elif event.type == 'ORDER':
                            self.orders += 1
                            self.execution_handler.execute_order(event)

                        elif event.type == 'FILL':
                            self.fills += 1
                            self.portfolio.update_fill(event)

            time.sleep(self.heartbeat)
```

Once the backtest simulation is complete the performance of the strategy can be displayed to the terminal/console. The equity curve pandas DataFrame is created and the summary statistics are displayed, as well as the count of Signals, Orders and Fills:

```
# backtest.py

    def _output_performance(self):
        """
        Outputs the strategy performance from the backtest.
```

```
        """
        self.portfolio.create_equity_curve_dataframe()

        print("Creating summary stats...")
        stats = self.portfolio.output_summary_stats()

        print("Creating equity curve...")
        print(self.portfolio.equity_curve.tail(10))
        pprint.pprint(stats)

        print("Signals: %s" % self.signals)
        print("Orders: %s" % self.orders)
        print("Fills: %s" % self.fills)
```

The last method to be implemented is `simulate_trading`. It simply calls the two previously described methods, in order:

```
# backtest.py

    def simulate_trading(self):
        """
        Simulates the backtest and outputs portfolio performance.
        """
        self._run_backtest()
        self._output_performance()
```

This concludes the event-driven backtester operational objects.

14.3 Event-Driven Execution

Above we described a basic `ExecutionHandler` class that simply created a corresponding `FillEvent` instance for every `OrderEvent`. This is precisely what we need for a "first pass" backtest, but when we wish to actually hook up the system to a brokerage, we need more sophisticated handling. In this section we define the `IBExecutionHandler`, a class that allows us to talk to the popular Interactive Brokers API and thus automate our execution.

The essential idea of the `IBExecutionHandler` class is to receive `OrderEvent` instances from the events queue and then to execute them directly against the Interactive Brokers order API using the open source IbPy library. The class will also handle the "Server Response" messages sent back via the API. At this stage, the only action taken will be to create corresponding `FillEvent` instances that will then be sent back to the events queue.

The class itself could feasibly become rather complex, with execution optimisation logic as well as sophisticated error handling. However, I have opted to keep it relatively simple here so that you can see the main ideas and extend it in the direction that suits your particular trading style.

As always, the first task is to create the Python file and import the necessary libraries. The file is called `ib_execution.py` and lives in the same directory as the other event-driven files.

We import the necessary date/time handling libraries, the IbPy objects and the specific Event objects that are handled by `IBExecutionHandler`:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# ib_execution.py

from __future__ import print_function

import datetime
import time
```

```
from ib.ext.Contract import Contract
from ib.ext.Order import Order
from ib.opt import ibConnection, message

from event import FillEvent, OrderEvent
from execution import ExecutionHandler
```

We now define the `IBExecutionHandler` class. The `__init__` constructor firstly requires knowledge of the `events` queue. It also requires specification of `order_routing`, which I've defaulted to "SMART". If you have specific exchange requirements, you can specify them here. The default `currency` has also been set to US Dollars.

Within the method we create a `fill_dict` dictionary, needed later for usage in generating `FillEvent` instances. We also create a `tws_conn` connection object to store our connection information to the Interactive Brokers API. We also have to create an initial default `order_id`, which keeps track of all subsequent orders to avoid duplicates. Finally we register the message handlers (which we'll define in more detail below):

```
# ib_execution.py

class IBExecutionHandler(ExecutionHandler):
    """
    Handles order execution via the Interactive Brokers
    API, for use against accounts when trading live
    directly.
    """

    def __init__(
        self, events, order_routing="SMART", currency="USD"
    ):
        """
        Initialises the IBExecutionHandler instance.
        """
        self.events = events
        self.order_routing = order_routing
        self.currency = currency
        self.fill_dict = {}

        self.tws_conn = self.create_tws_connection()
        self.order_id = self.create_initial_order_id()
        self.register_handlers()
```

The IB API utilises a message-based event system that allows our class to respond in particular ways to certain messages, in a similar manner to the event-driven backtester itself. I've not included any real error handling (for the purposes of brevity), beyond output to the terminal, via the `_error_handler` method.

The `_reply_handler` method, on the other hand, is used to determine if a `FillEvent` instance needs to be created. The method asks if an "openOrder" message has been received and checks whether an entry in our `fill_dict` for this particular orderId has already been set. If not then one is created.

If it sees an "orderStatus" message and that particular message states than an order has been filled, then it calls `create_fill` to create a `FillEvent`. It also outputs the message to the terminal for logging/debug purposes:

```
# ib_execution.py

    def _error_handler(self, msg):
        """
```

```
        Handles the capturing of error messages
        """
        # Currently no error handling.
        print("Server Error: %s" % msg)

    def _reply_handler(self, msg):
        """
        Handles of server replies
        """
        # Handle open order orderId processing
        if msg.typeName == "openOrder" and \
            msg.orderId == self.order_id and \
            not self.fill_dict.has_key(msg.orderId):
            self.create_fill_dict_entry(msg)
        # Handle Fills
        if msg.typeName == "orderStatus" and \
            msg.status == "Filled" and \
            self.fill_dict[msg.orderId]["filled"] == False:
            self.create_fill(msg)
        print("Server Response: %s, %s\n" % (msg.typeName, msg))
```

The following method, `create_tws_connection`, creates a connection to the IB API using the IbPy `ibConnection` object. It uses a default port of 7496 and a default clientId of 10. Once the object is created, the `connect` method is called to perform the connection:

```
# ib_execution.py

    def create_tws_connection(self):
        """
        Connect to the Trader Workstation (TWS) running on the
        usual port of 7496, with a clientId of 10.
        The clientId is chosen by us and we will need
        separate IDs for both the execution connection and
        market data connection, if the latter is used elsewhere.
        """
        tws_conn = ibConnection()
        tws_conn.connect()
        return tws_conn
```

To keep track of separate orders (for the purposes of tracking fills) the following method `create_initial_order_id` is used. I've defaulted it to "1", but a more sophisticated approach would be th query IB for the latest available ID and use that. You can always reset the current API order ID via the Trader Workstation > Global Configuration > API Settings panel:

```
# ib_execution.py

    def create_initial_order_id(self):
        """
        Creates the initial order ID used for Interactive
        Brokers to keep track of submitted orders.
        """
        # There is scope for more logic here, but we
        # will use "1" as the default for now.
        return 1
```

The following method, `register_handlers`, simply registers the error and reply handler methods defined above with the TWS connection:

```
# ib_execution.py
```

```
    def register_handlers(self):
        """
        Register the error and server reply
        message handling functions.
        """
        # Assign the error handling function defined above
        # to the TWS connection
        self.tws_conn.register(self._error_handler, 'Error')

        # Assign all of the server reply messages to the
        # reply_handler function defined above
        self.tws_conn.registerAll(self._reply_handler)
```

In order to actually transact a trade it is necessary to create an IbPy `Contract` instance and then pair it with an IbPy `Order` instance, which will be sent to the IB API. The following method, `create_contract`, generates the first component of this pair. It expects a ticker symbol, a security type (e.g. stock or future), an exchange/primary exchange and a currency. It returns the `Contract` instance:

```
# ib_execution.py

    def create_contract(self, symbol, sec_type, exch, prim_exch, curr):
        """
        Create a Contract object defining what will
        be purchased, at which exchange and in which currency.

        symbol - The ticker symbol for the contract
        sec_type - The security type for the contract ('STK' is 'stock')
        exch - The exchange to carry out the contract on
        prim_exch - The primary exchange to carry out the contract on
        curr - The currency in which to purchase the contract
        """
        contract = Contract()
        contract.m_symbol = symbol
        contract.m_secType = sec_type
        contract.m_exchange = exch
        contract.m_primaryExch = prim_exch
        contract.m_currency = curr
        return contract
```

The following method, `create_order`, generates the second component of the pair, namely the `Order` instance. It expects an order type (e.g. market or limit), a quantity of the asset to trade and an "action" (buy or sell). It returns the `Order` instance:

```
# ib_execution.py

    def create_order(self, order_type, quantity, action):
        """
        Create an Order object (Market/Limit) to go long/short.

        order_type - 'MKT', 'LMT' for Market or Limit orders
        quantity - Integral number of assets to order
        action - 'BUY' or 'SELL'
        """
        order = Order()
        order.m_orderType = order_type
        order.m_totalQuantity = quantity
```

```
        order.m_action = action
        return order
```

In order to avoid duplicating `FillEvent` instances for a particular order ID, we utilise a dictionary called the `fill_dict` to store keys that match particular order IDs. When a fill has been generated the "filled" key of an entry for a particular order ID is set to `True`. If a subsequent "Server Response" message is received from IB stating that an order has been filled (and is a duplicate message) it will not lead to a new fill. The following method `create_fill_dict_entry` carries this out:

```
# ib_execution.py

    def create_fill_dict_entry(self, msg):
        """
        Creates an entry in the Fill Dictionary that lists
        orderIds and provides security information. This is
        needed for the event-driven behaviour of the IB
        server message behaviour.
        """
        self.fill_dict[msg.orderId] = {
            "symbol": msg.contract.m_symbol,
            "exchange": msg.contract.m_exchange,
            "direction": msg.order.m_action,
            "filled": False
        }
```

The following method, `create_fill`, actually creates the `FillEvent` instance and places it onto the events queue:

```
# ib_execution.py

    def create_fill(self, msg):
        """
        Handles the creation of the FillEvent that will be
        placed onto the events queue subsequent to an order
        being filled.
        """
        fd = self.fill_dict[msg.orderId]

        # Prepare the fill data
        symbol = fd["symbol"]
        exchange = fd["exchange"]
        filled = msg.filled
        direction = fd["direction"]
        fill_cost = msg.avgFillPrice

        # Create a fill event object
        fill = FillEvent(
            datetime.datetime.utcnow(), symbol,
            exchange, filled, direction, fill_cost
        )

        # Make sure that multiple messages don't create
        # additional fills.
        self.fill_dict[msg.orderId]["filled"] = True

        # Place the fill event onto the event queue
        self.events.put(fill_event)
```

Now that all of the preceeding methods having been implemented it remains to override the `execute_order` method from the `ExecutionHandler` abstract base class. This method actually carries out the order placement with the IB API.

We first check that the event being received to this method is actually an `OrderEvent` and then prepare the `Contract` and `Order` objects with their respective parameters. Once both are created the IbPy method `placeOrder` of the connection object is called with an associated `order_id`.

It is *extremely important* to call the `time.sleep(1)` method to ensure the order actually goes through to IB. Removal of this line leads to inconsistent behaviour of the API, at least on my system!

Finally, we increment the order ID to ensure we don't duplicate orders:

```
# ib_execution.py

    def execute_order(self, event):
        """
        Creates the necessary InteractiveBrokers order object
        and submits it to IB via their API.

        The results are then queried in order to generate a
        corresponding Fill object, which is placed back on
        the event queue.

        Parameters:
        event - Contains an Event object with order information.
        """
        if event.type == 'ORDER':
            # Prepare the parameters for the asset order
            asset = event.symbol
            asset_type = "STK"
            order_type = event.order_type
            quantity = event.quantity
            direction = event.direction

            # Create the Interactive Brokers contract via the
            # passed Order event
            ib_contract = self.create_contract(
                asset, asset_type, self.order_routing,
                self.order_routing, self.currency
            )

            # Create the Interactive Brokers order via the
            # passed Order event
            ib_order = self.create_order(
                order_type, quantity, direction
            )

            # Use the connection to the send the order to IB
            self.tws_conn.placeOrder(
                self.order_id, ib_contract, ib_order
            )

            # NOTE: This following line is crucial.
            # It ensures the order goes through!
            time.sleep(1)

            # Increment the order ID for this session
```

```
            self.order_id += 1
```

This class forms the basis of an Interactive Brokers execution handler and can be used in place of the simulated execution handler, which is only suitable for backtesting. Before the IB handler can be utilised, however, it is necessary to create a live market feed handler to replace the historical data feed handler of the backtester system.

In this way we are reusing as much as possible from the backtest and live systems to ensure that code "swap out" is minimised and thus behaviour across both is similar, if not identical.

Chapter 15

Trading Strategy Implementation

In this chapter we are going to consider the full implementation of trading strategies using the aforementioned event-driven backtesting system. In particular we will generate equity curves for all trading strategies using notional portfolio amounts, thus simulating the concepts of margin/leverage, which is a far more realistic approach compared to vectorised/returns based approaches.

The first set of strategies are able to be carried out with freely available data, either from Yahoo Finance, Google Finance or Quandl. These strategies are suitable for long-term algorithmic traders who may wish to only study the trade signal generation aspect of the strategy or even the full end-to-end system. Such strategies often possess smaller Sharpe ratios, but are far easier to implement and execute.

The latter strategy is carried out using intraday equities data. This data is often not freely available and a commercial data vendor is usually necessary to provide sufficient quality and quantity of data. I myself use DTN IQFeed for intraday bars. Such strategies often possess much larger Sharpe ratios, but require more sophisticated implementation as the high frequency requires extensive automation.

We will see that our first two attempts at creating a trading strategy on interday data are not altogether successful. It can be challenging to come up with a profitable trading strategy on interday data once transaction costs have been taken into account. The latter is something that many texts on algorithmic trading tend to leave out. However, it is my belief that as many factors as possible must be added to the backtest in order to minimises surprises going forward.

In addition, this book is primarily about how to effectively create a realistic interday or intraday backtesting system (as well as a live execution platform) and less about particular individual strategies. It is far harder to create a realistic robust backtester than it is to find trading strategies on the internet! While the first two strategies presented are not particularly attractive, the latter strategy (on intraday data) performs well and gives us confidence in using higher frequency data.

15.1 Moving Average Crossover Strategy

I'm quite fond of the Moving Average Crossover technical system because it is the first non-trivial strategy that is extremely handy for testing a new backtesting implementation. On a daily timeframe, over a number of years, with long lookback periods, few signals are generated on a single stock and thus it is easy to manually verify that the system is behaving as would be expected.

In order to actually generate such a simulation based on the prior backtesting code we need to subclass the `Strategy` object as described in the previous chapter to create the `MovingAverageCrossStrategy` object, which will contain the logic of the simple moving averages and the generation of trading signals.

In addition we need to create the `__main__` function that will load the `Backtest` object and actually encapsulate the execution of the program. The following file, `mac.py`, contains both of these objects.

The first task, as always, is to correctly import the necessary components. We are importing nearly all of the objects that have been described in the previous chapter:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# mac.py

from __future__ import print_function

import datetime

import numpy as np
import pandas as pd
import statsmodels.api as sm

from strategy import Strategy
from event import SignalEvent
from backtest import Backtest
from data import HistoricCSVDataHandler
from execution import SimulatedExecutionHandler
from portfolio import Portfolio
```

Now we turn to the creation of the `MovingAverageCrossStrategy`. The strategy requires both the bars `DataHandler`, the events Event Queue and the lookback periods for the simple moving averages that are going to be employed within the strategy. I've chosen 100 and 400 as the "short" and "long" lookback periods for this strategy.

The final attribute, `bought`, is used to tell the `Strategy` when the backtest is actually "in the market". Entry signals are only generated if this is "OUT" and exit signals are only ever generated if this is "LONG" or "SHORT":

```
# mac.py

class MovingAverageCrossStrategy(Strategy):
    """
    Carries out a basic Moving Average Crossover strategy with a
    short/long simple weighted moving average. Default short/long
    windows are 100/400 periods respectively.
    """

    def __init__(
        self, bars, events, short_window=100, long_window=400
    ):
        """
        Initialises the Moving Average Cross Strategy.

        Parameters:
        bars - The DataHandler object that provides bar information
        events - The Event Queue object.
        short_window - The short moving average lookback.
        long_window - The long moving average lookback.
        """
        self.bars = bars
        self.symbol_list = self.bars.symbol_list
        self.events = events
        self.short_window = short_window
        self.long_window = long_window
```

```
        # Set to True if a symbol is in the market
        self.bought = self._calculate_initial_bought()
```

Since the strategy begins out of the market we set the initial "bought" value to be "OUT", for each symbol:

```
# mac.py

    def _calculate_initial_bought(self):
        """
        Adds keys to the bought dictionary for all symbols
        and sets them to 'OUT'.
        """
        bought = {}
        for s in self.symbol_list:
            bought[s] = 'OUT'
        return bought
```

The core of the strategy is the `calculate_signals` method. It reacts to a `MarketEvent` object and for each symbol traded obtains the latest N bar closing prices, where N is equal to the largest lookback period.

It then calculates both the short and long period *simple moving averages*. The rule of the strategy is to enter the market (go long a stock) when the short moving average value exceeds the long moving average value. Conversely, if the long moving average value exceeds the short moving average value the strategy is told to exit the market.

This logic is handled by placing a `SignalEvent` object on the events Event Queue in each of the respective situations and then updating the "bought" attribute (per symbol) to be "LONG" or "OUT", respectively. Since this is a long-only strategy, we won't be considering "SHORT" positions:

```
# mac.py

    def calculate_signals(self, event):
        """
        Generates a new set of signals based on the MAC
        SMA with the short window crossing the long window
        meaning a long entry and vice versa for a short entry.

        Parameters
        event - A MarketEvent object.
        """
        if event.type == 'MARKET':
            for s in self.symbol_list:
                bars = self.bars.get_latest_bars_values(
                    s, "adj_close", N=self.long_window
                )
                bar_date = self.bars.get_latest_bar_datetime(s)
                if bars is not None and bars != []:
                    short_sma = np.mean(bars[-self.short_window:])
                    long_sma = np.mean(bars[-self.long_window:])

                    symbol = s
                    dt = datetime.datetime.utcnow()
                    sig_dir = ""

                    if short_sma > long_sma and self.bought[s] == "OUT":
                        print("LONG: %s" % bar_date)
                        sig_dir = 'LONG'
```

```
                        signal = SignalEvent(1, symbol, dt, sig_dir, 1.0)
                        self.events.put(signal)
                        self.bought[s] = 'LONG'
                    elif short_sma < long_sma and self.bought[s] == "LONG":
                        print("SHORT: %s" % bar_date)
                        sig_dir = 'EXIT'
                        signal = SignalEvent(1, symbol, dt, sig_dir, 1.0)
                        self.events.put(signal)
                        self.bought[s] = 'OUT'
```

That concludes the `MovingAverageCrossStrategy` object implementation. The final task of the entire backtesting system is populate a `__main__` method in `mac.py` to actually execute the backtest.

Firstly, make sure to change the value of `csv_dir` to the absolute path of your CSV file directory for the financial data. You will also need to download the CSV file of the AAPL stock (from Yahoo Finance), which is given by the following link (for Jan 1st 1990 to Jan 1st 2002), since this is the stock we will be testing the strategy on:

http://ichart.finance.yahoo.com/table.csv?s=AAPL&a=00&b=1&c=1990&d=00&e=1&f=2002&g=d&ignore=.csv

Make sure to place this file in the path pointed to from the main function in `csv_dir`.

The `__main__` function simply instantiates a new backtest object and then calls the simulate_trading method on it to execute it:

```
# mac.py

if __name__ == "__main__":
    csv_dir = '/path/to/your/csv/file'  # CHANGE THIS!
    symbol_list = ['AAPL']
    initial_capital = 100000.0
    heartbeat = 0.0
    start_date = datetime.datetime(1990, 1, 1, 0, 0, 0)

    backtest = Backtest(
        csv_dir, symbol_list, initial_capital, heartbeat,
        start_date, HistoricCSVDataHandler, SimulatedExecutionHandler,
        Portfolio, MovingAverageCrossStrategy
    )
    backtest.simulate_trading()
```

To run the code, make sure you have already set up a Python environment (as described in the previous chapters) and then navigate the directory where your code is stored. You should simply be able to run:

```
python mac.py
```

You will see the following listing (truncated due to the bar count printout!):

```
..
..
3029
3030
Creating summary stats...
Creating equity curve...
            AAPL   cash  commission  total  returns  equity_curve  drawdown
datetime
2001-12-18     0  99211          13  99211        0       0.99211  0.025383
2001-12-19     0  99211          13  99211        0       0.99211  0.025383
2001-12-20     0  99211          13  99211        0       0.99211  0.025383
2001-12-21     0  99211          13  99211        0       0.99211  0.025383
```

```
2001-12-24     0  99211          13  99211        0       0.99211  0.025383
2001-12-26     0  99211          13  99211        0       0.99211  0.025383
2001-12-27     0  99211          13  99211        0       0.99211  0.025383
2001-12-28     0  99211          13  99211        0       0.99211  0.025383
2001-12-31     0  99211          13  99211        0       0.99211  0.025383
2001-12-31     0  99211          13  99211        0       0.99211  0.025383
[('Total Return', '-0.79%'),
 ('Sharpe Ratio', '-0.09'),
 ('Max Drawdown', '2.56%'),
 ('Drawdown Duration', '2312')]
Signals: 10
Orders: 10
Fills: 10
```

The performance of this strategy can be seen in Fig 15.1:

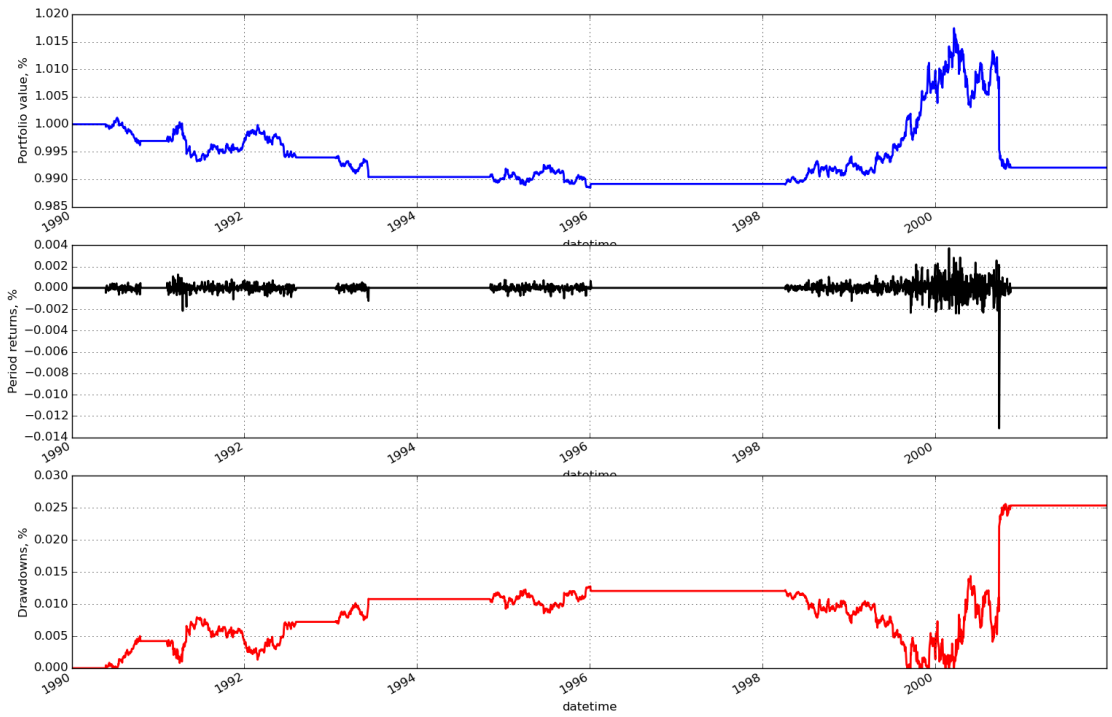


Figure 15.1: Equity Curve, Daily Returns and Drawdowns for the Moving Average Crossover strategy

Evidently the returns and Sharpe Ratio are not stellar for AAPL stock on this particular set of technical indicators! Clearly we have some work to do in the next set of strategies to find a system that can generate positive performance.

15.2 S&P500 Forecasting Trade

In this sectiuon we will consider a trading strategy built around the forecasting engine discussed in prior chapters. We will attempt to trade off the predictions made by a stock market forecaster.

We are going to attempt to forecast SPY, which is the ETF that tracks the value of the S&P500. Ultimately we want to answer the question as to whether a basic forecasting algorithm using lagged price data, with slight predictive performance, provides us with any benefit over a buy-and-hold strategy.

The rules for this strategy are as follows:

1. Fit a forecasting model to a subset of S&P500 data. This could be Logistic Regression, a Discriminant Analyser (Linear or Quadratic), a Support Vector Machine or a Random Forest. The procedure to do this was outlined in the Forecasting chapter.

2. Use two prior lags of adjusted closing returns data as a predictor for tomorrow's returns. If the returns are predicted as positive then go long. If the returns are predicted as negative then exit. We're not going to consider short selling for this particular strategy.

Implementation

For this strategy we are going to create the `snp_forecast.py` file and import the following necessary libraries:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# snp_forecast.py

from __future__ import print_function

import datetime

import pandas as pd
from sklearn.qda import QDA

from strategy import Strategy
from event import SignalEvent
from backtest import Backtest
from data import HistoricCSVDataHandler
from execution import SimulatedExecutionHandler
from portfolio import Portfolio
from create_lagged_series import create_lagged_series
```

We have imported Pandas and Scikit-Learn in order to carry out the fitting procedure for the supervised classifier model. We have also imported the necessary classes from the event-driven backtester. Finally, we have imported the `create_lagged_series` function, which we used in the Forecasting chapter.

The next step is to create the `SPYDailyForecastStrategy` as a subclass of the `Strategy` abstract base class. Since we will "hardcode" the parameters of the strategy directly into the class, for simplicity, the only parameters necessary for the `__init__` constructor are the `bars` data handler and the `events` queue.

We set the `self.model_***` start/end/test dates as datetime objects and then tell the class that we are out of the market (`self.long_market = False`). Finally, we set `self.model` to be the trained model from the `create_symbol_forecast_model` below:

```
# snp_forecast.py

class SPYDailyForecastStrategy(Strategy):
    """
```

```
    S&P500 forecast strategy. It uses a Quadratic Discriminant
    Analyser to predict the returns for a subsequent time
    period and then generated long/exit signals based on the
    prediction.
    """
    def __init__(self, bars, events):
        self.bars = bars
        self.symbol_list = self.bars.symbol_list
        self.events = events
        self.datetime_now = datetime.datetime.utcnow()

        self.model_start_date = datetime.datetime(2001,1,10)
        self.model_end_date = datetime.datetime(2005,12,31)
        self.model_start_test_date = datetime.datetime(2005,1,1)

        self.long_market = False
        self.short_market = False
        self.bar_index = 0

        self.model = self.create_symbol_forecast_model()
```

Here we define the `create_symbol_forecast_model`. It essentially calls the `create_lagged_series` function, which produces a Pandas DataFrame with five daily returns lags for each current predictor. We then consider only the two most recent of these lags. This is because we are making the modelling decision that the predictive power of earlier lags is likely to be minimal.

At this stage we create the training and test data, the latter of which can be used to test our model if we wish. I have opted to not output testing data, since we have already trained the model before in the Forecasting chapter. Finally we fit the training data to the Quadratic Discriminant Analyser and then return the model.

Note that we could easily replace the model with a Random Forest, Support Vector Machine or Logistic Regression, for instance. All we need to do is import the correct library from Scikit-Learn and simply replace the `model = QDA()` line:

```
# snp_forecast.py

    def create_symbol_forecast_model(self):
        # Create a lagged series of the S&P500 US stock market index
        snpret = create_lagged_series(
            self.symbol_list[0], self.model_start_date,
            self.model_end_date, lags=5
        )

        # Use the prior two days of returns as predictor
        # values, with direction as the response
        X = snpret[["Lag1","Lag2"]]
        y = snpret["Direction"]

        # Create training and test sets
        start_test = self.model_start_test_date
        X_train = X[X.index < start_test]
        X_test = X[X.index >= start_test]
        y_train = y[y.index < start_test]
        y_test = y[y.index >= start_test]

        model = QDA()
        model.fit(X_train, y_train)
        return model
```

At this stage we are ready to override the `calculate_signals` method of the `Strategy` base class. We firstly calculate some convenience parameters that enter our `SignalEvent` object and then only generate a set of signals if we have received a `MarketEvent` object (a basic sanity check).

We wait for five bars to have elapsed (i.e. five days in this strategy!) and then obtain the lagged returns values. We then wrap these values in a Pandas Series so that the `predict` method of the model will function correctly. We then calculate a prediction, which manifests itself as a +1 or -1.

If the prediction is a +1 and we are not already long the market, we create a `SignalEvent` to go long and let the class know we are now in the market. If the prediction is -1 and we are long the market, then we simply exit the market:

```
# snp_forecast.py

    def calculate_signals(self, event):
        """
        Calculate the SignalEvents based on market data.
        """
        sym = self.symbol_list[0]
        dt = self.datetime_now

        if event.type == 'MARKET':
            self.bar_index += 1
            if self.bar_index > 5:
                lags = self.bars.get_latest_bars_values(
                    self.symbol_list[0], "returns", N=3
                )
                pred_series = pd.Series(
                    {
                        'Lag1': lags[1]*100.0,
                        'Lag2': lags[2]*100.0
                    }
                )
                pred = self.model.predict(pred_series)
                if pred > 0 and not self.long_market:
                    self.long_market = True
                    signal = SignalEvent(1, sym, dt, 'LONG', 1.0)
                    self.events.put(signal)

                if pred < 0 and self.long_market:
                    self.long_market = False
                    signal = SignalEvent(1, sym, dt, 'EXIT', 1.0)
                    self.events.put(signal)
```

In order to run the strategy you will need to download a CSV file from Yahoo Finance for SPY and place it in a suitable directory (note that you will need to change your path below!). We then wrap the backtest up via the `Backtest` class and carry out the test by calling `simulate_trading`:

```
# snp_forecast.py

if __name__ == "__main__":
    csv_dir = '/path/to/your/csv/file'  # CHANGE THIS!
    symbol_list = ['SPY']
    initial_capital = 100000.0
    heartbeat = 0.0
    start_date = datetime.datetime(2006,1,3)

    backtest = Backtest(
```

```
        csv_dir, symbol_list, initial_capital, heartbeat,
        start_date, HistoricCSVDataHandler, SimulatedExecutionHandler,
        Portfolio, SPYDailyForecastStrategy
    )
    backtest.simulate_trading()
```

The output of the strategy is as follows and is net of transaction costs:

```
..
..
2209
2210
Creating summary stats...
Creating equity curve...
              SPY      cash  commission     total   returns  equity_curve  \
datetime
2014-09-29  19754   90563.3       349.7  110317.3 -0.000326      1.103173
2014-09-30  19702   90563.3       349.7  110265.3 -0.000471      1.102653
2014-10-01  19435   90563.3       349.7  109998.3 -0.002421      1.099983
2014-10-02  19438   90563.3       349.7  110001.3  0.000027      1.100013
2014-10-03  19652   90563.3       349.7  110215.3  0.001945      1.102153
2014-10-06  19629   90563.3       349.7  110192.3 -0.000209      1.101923
2014-10-07  19326   90563.3       349.7  109889.3 -0.002750      1.098893
2014-10-08  19664   90563.3       349.7  110227.3  0.003076      1.102273
2014-10-09  19274   90563.3       349.7  109837.3 -0.003538      1.098373
2014-10-09      0  109836.0       351.0  109836.0 -0.000012      1.098360

            drawdown
datetime
2014-09-29  0.003340
2014-09-30  0.003860
2014-10-01  0.006530
2014-10-02  0.006500
2014-10-03  0.004360
2014-10-06  0.004590
2014-10-07  0.007620
2014-10-08  0.004240
2014-10-09  0.008140
2014-10-09  0.008153
[('Total Return', '9.84%'),
 ('Sharpe Ratio', '0.54'),
 ('Max Drawdown', '5.99%'),
 ('Drawdown Duration', '811')]
Signals: 270
Orders: 270
Fills: 270
```

The following visualisation in Fig 15.2 shows the Equity Curve, the Daily Returns and the Drawdown of the strategy as a function of time:

Note immediately that the performance is not great! We have a Sharpe Ratio < 1 but a reasonable drawdown of just under 6%. It turns out that if we had simply bought and held SPY in this time period we would have performed similarly, if slightly worse.

Hence we have not actually gained very much from our predictive strategy once transaction costs are included. I specifically wanted to include this example because it uses an "end to end" realistic implementation of such a strategy that takes into account conservative, realistic transaction costs. As can be seen it is not easy to make a predictive forecaster on daily data that produces good performance!

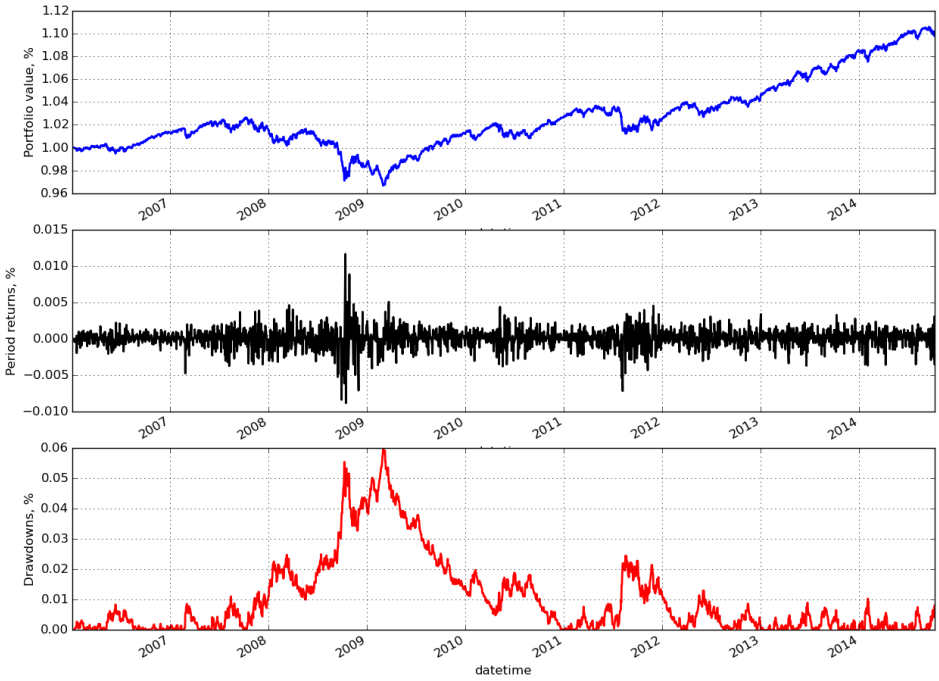


Figure 15.2: Equity Curve, Daily Returns and Drawdowns for the SPY forecast strategy

Our final strategy will make use of other time series and a higher frequency. We will see that performance can be improved dramatically after modifying certain aspects of the system.

15.3 Mean-Reverting Equity Pairs Trade

In order to seek higher Sharpe ratios for our trading, we need to consider higher-frequency intraday strategies.

The first major issue is that obtaining data is significantly less straightforward because high quality intraday data is usually not free. As stated above I use DTN IQFeed for intraday minutely bars and thus you will need your own DTN account to obtain the data required for this strategy.

The second issue is that backtesting simulations take substantially longer, especially with the event-driven model that we have constructed here. Once we begin considering a backtest of a diversified portfolio of minutely data spanning years, and then performing any parameter optimisation, we rapidly realise that simulations can take hours or even days to calculate on a modern desktop PC. This will need to be factored in to your research process.

The third issue is that live execution will now need to be fully automated since we are edging into higher-frequency trading. This means that such execution environments and code must be highly reliable and bug-free, otherwise the potential for significant losses can occur.

This strategy expands on the previous interday strategy above to make use of intraday data. In particular we are going to use minutely OHLCV bars, as opposed to daily OHLCV.

The rules for the strategy are straightforward:

1. Identify a pair of equities that possess a residuals time series which has been statistically identified as mean-reverting. In this case, I have found two energy sector US equities with tickers AREX and WLL.

2. Create the residuals time series of the pair by performing a rolling linear regression, for a particular lookback window, via the ordinary least squares (OLS) algorithm. This lookback period is a parameter to be optimised.

3. Create a rolling z-score of the residuals time series of the same lookback period and use this to determine entry/exit thresholds for trading signals.

4. If the upper threshold is exceeded when not in the market then enter the market (long or short depending on direction of threshold excess). If the lower threshold is exceeded when in the market, exit the market. Once again, the upper and lower thresholds are parameters to be optimised.

Indeed we could have used the Cointegrated Augmented Dickey-Fuller (CADF) test to identify an even more accurate hedging parameter. This would make an interesting extension of the strategy.

Implementation

The first step, as always, is to import the necessary libraries. We require pandas for the `rolling_apply` method, which is used to apply the z-score calculation with a lookback window on a rolling basis. We import statsmodels because it provides a means of calculating the ordinary least squares (OLS) algorithm for the linear regression, necessary to obtain the hedging ratio for the construction of the residuals.

We also require a slightly modified `DataHandler` and `Portfolio` in order to carry out minutely bars trading on DTN IQFeed data. In order to create these files you can simply copy all of the code in `portfolio.py` and `data.py` into the new files `hft_portfolio.py` and `hft_data.py` respectively and then modify the necessary sections, which I will outline below.

Here is the import listing for `intraday_mr.py`:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# intraday_mr.py

from __future__ import print_function

import datetime

import numpy as np
import pandas as pd
import statsmodels.api as sm

from strategy import Strategy
from event import SignalEvent
from backtest import Backtest
from hft_data import HistoricCSVDataHandlerHFT
from hft_portfolio import PortfolioHFT
from execution import SimulatedExecutionHandler
```

In the following snippet we create the `IntradayOLSMRStrategy` class derived from the `Strategy` abstract base class. The constructor `__init__` method requires access to the `bars` historical data provider, the `events` queue, a `zscore_low` threshold and a `zscore_high` threshold, used to determine when the residual series between the two pairs is mean-reverting.

In addition, we specify the OLS lookback window (set to 100 here), which is a parameter that is subject to potential optimisation. At the start of the simulation we are neither long or short the market, so we set both `self.long_market` and `self.short_market` equal to `False`:

```
# intraday_mr.py
```

```
class IntradayOLSMRStrategy(Strategy):
    """
    Uses ordinary least squares (OLS) to perform a rolling linear
    regression to determine the hedge ratio between a pair of equities.
    The z-score of the residuals time series is then calculated in a
    rolling fashion and if it exceeds an interval of thresholds
    (defaulting to [0.5, 3.0]) then a long/short signal pair are generated
    (for the high threshold) or an exit signal pair are generated (for the
    low threshold).
    """

    def __init__(
        self, bars, events, ols_window=100,
        zscore_low=0.5, zscore_high=3.0
    ):
        """
        Initialises the stat arb strategy.

        Parameters:
        bars - The DataHandler object that provides bar information
        events - The Event Queue object.
        """
        self.bars = bars
        self.symbol_list = self.bars.symbol_list
        self.events = events
        self.ols_window = ols_window
        self.zscore_low = zscore_low
        self.zscore_high = zscore_high

        self.pair = ('AREX', 'WLL')
        self.datetime = datetime.datetime.utcnow()

        self.long_market = False
        self.short_market = False
```

The following method, `calculate_xy_signals`, takes the current zscore (from the rolling calculation performed below) and determines whether new trading signals need to be generated. These signals are then returned.

There are four potential states that we may be interested in. They are:

1. Long the market and below the negative zscore higher threshold
2. Long the market and between the absolute value of the zscore lower threshold
3. Short the market and above the positive zscore higher threshold
4. Short the market and between the absolute value of the zscore lower threshold

In either case it is necessary to generate two signals, one for the first component of the pair (AREX) and one for the second component of the pair (WLL). If none of these conditions are reached, then a pair of `None` values are returned:

```
# intraday_mr.py

    def calculate_xy_signals(self, zscore_last):
        """
        Calculates the actual x, y signal pairings
        to be sent to the signal generator.
```

```
        Parameters
        zscore_last - The current zscore to test against
        """
        y_signal = None
        x_signal = None
        p0 = self.pair[0]
        p1 = self.pair[1]
        dt = self.datetime
        hr = abs(self.hedge_ratio)

        # If we're long the market and below the
        # negative of the high zscore threshold
        if zscore_last <= -self.zscore_high and not self.long_market:
            self.long_market = True
            y_signal = SignalEvent(1, p0, dt, 'LONG', 1.0)
            x_signal = SignalEvent(1, p1, dt, 'SHORT', hr)

        # If we're long the market and between the
        # absolute value of the low zscore threshold
        if abs(zscore_last) <= self.zscore_low and self.long_market:
            self.long_market = False
            y_signal = SignalEvent(1, p0, dt, 'EXIT', 1.0)
            x_signal = SignalEvent(1, p1, dt, 'EXIT', 1.0)

        # If we're short the market and above
        # the high zscore threshold
        if zscore_last >= self.zscore_high and not self.short_market:
            self.short_market = True
            y_signal = SignalEvent(1, p0, dt, 'SHORT', 1.0)
            x_signal = SignalEvent(1, p1, dt, 'LONG', hr)

        # If we're short the market and between the
        # absolute value of the low zscore threshold
        if abs(zscore_last) <= self.zscore_low and self.short_market:
            self.short_market = False
            y_signal = SignalEvent(1, p0, dt, 'EXIT', 1.0)
            x_signal = SignalEvent(1, p1, dt, 'EXIT', 1.0)

        return y_signal, x_signal
```

The following method, `calculate_signals_for_pairs` obtains the latest set of bars for each component of the pair (in this case 100 bars) and uses them to construct an ordinary least squares based linear regression. This allows identification of the hedge ratio, necessary for the construction of the residuals time series.

Once the hedge ratio is constructed, a `spread` series of residuals is constructed. The next step is to calculate the latest zscore from the residual series by subtracting its mean and dividing by its standard deviation over the lookback period.

Finally, the `y_signal` and `x_signal` are calculated on the basis of this zscore. If the signals are not both `None` then the `SignalEvent` instances are sent back to the `events` queue:

```
# intraday_mr.py

    def calculate_signals_for_pairs(self):
        """
        Generates a new set of signals based on the mean reversion
        strategy.
```

```
        Calculates the hedge ratio between the pair of tickers.
        We use OLS for this, althought we should ideall use CADF.
        """
        # Obtain the latest window of values for each
        # component of the pair of tickers
        y = self.bars.get_latest_bars_values(
            self.pair[0], "close", N=self.ols_window
        )
        x = self.bars.get_latest_bars_values(
            self.pair[1], "close", N=self.ols_window
        )

        if y is not None and x is not None:
            # Check that all window periods are available
            if len(y) >= self.ols_window and len(x) >= self.ols_window:
                # Calculate the current hedge ratio using  OLS
                self.hedge_ratio = sm.OLS(y, x).fit().params[0]

                # Calculate the current z-score of the residuals
                spread = y - self.hedge_ratio * x
                zscore_last = ((spread - spread.mean())/spread.std())[-1]

                # Calculate signals and add to events queue
                y_signal, x_signal = self.calculate_xy_signals(zscore_last)
                if y_signal is not None and x_signal is not None:
                    self.events.put(y_signal)
                    self.events.put(x_signal)
```

The final method, `calculate_signals` is overidden from the base class and is used to check whether a received event from the queue is actually a `MarketEvent`, in which case the calculation of the new signals is carried out:

```
# intraday_mr.py

    def calculate_signals(self, event):
        """
        Calculate the SignalEvents based on market data.
        """
        if event.type == 'MARKET':
            self.calculate_signals_for_pairs()
```

The `__main__` section ties together the components to produce a backtest for the strategy. We tell the simulation where the ticker minutely data is stored. I'm using DTN IQFeed format. I truncated both files so that they began and ended on the same respective minute. For this particular pair of AREX and WLL, the common start date is 8th November 2007 at 10:41:00AM.

Finally, we build the backtest object and begin simulating the trading:

```
# intraday_mr.py

if __name__ == "__main__":
    csv_dir = '/path/to/your/csv/file'  # CHANGE THIS!
    symbol_list = ['AREX', 'WLL']
    initial_capital = 100000.0
    heartbeat = 0.0
    start_date = datetime.datetime(2007, 11, 8, 10, 41, 0)

    backtest = Backtest(
        csv_dir, symbol_list, initial_capital, heartbeat,
```

```
        start_date, HistoricCSVDataHandlerHFT, SimulatedExecutionHandler,
        PortfolioHFT, IntradayOLSMRStrategy
    )
    backtest.simulate_trading()
```

However, before we can execute this file we need to make some modifications to the data handler and portfolio objects.

In particular, it is necessary to create new files `hft_data.py` and `hft_portfolio.py` which are copies of `data.py` and `portfolio.py` respectively.

In `hft_data.py` we need to rename `HistoricCSVDataHandler` to `HistoricCSVDataHandlerHFT` and replace the `names` list in the `_open_convert_csv_files` method.

The old line is:

```
                names=[
                    'datetime', 'open', 'high',
                    'low', 'close', 'volume', 'adj_close'
                ]
```

This must be replaced with:

```
                names=[
                    'datetime', 'open', 'low',
                    'high', 'close', 'volume', 'oi'
                ]
```

This is to ensure that the new format for DTN IQFeed works with the backtester.

The other change is to rename `Portfolio` to `PortfolioHFT` in `hft_portfolio.py`. We must then modify a few lines in order to account for the minutely frequency of the DTN data.

In particular, within the `update_timeindex` method, we must change the following code:

```
        for s in self.symbol_list:
            # Approximation to the real value
            market_value = self.current_positions[s] * \
                self.bars.get_latest_bar_value(s, "adj_close")
            dh[s] = market_value
            dh['total'] += market_value
```

To:

```
        for s in self.symbol_list:
            # Approximation to the real value
            market_value = self.current_positions[s] * \
                self.bars.get_latest_bar_value(s, "close")
            dh[s] = market_value
            dh['total'] += market_value
```

This ensures we obtain the `close` price, rather than the `adj_close` price. The latter is for Yahoo Finance, whereas the former is for DTN IQFeed.

We must also make a similar adjustment in `update_holdings_from_fill`. We need to change the following code:

```
        # Update holdings list with new quantities
        fill_cost = self.bars.get_latest_bar_value(
            fill.symbol, "adj_close"
        )
```

To:

```
        # Update holdings list with new quantities
        fill_cost = self.bars.get_latest_bar_value(
            fill.symbol, "close"
        )
```

The final change is occurs in the `output_summary_stats` method at the bottom of the file. We need to modify how the Sharpe Ratio is calculated to take into account minutely trading. The following line:

```
        sharpe_ratio = create_sharpe_ratio(returns)
```

Must be changed to:

```
        sharpe_ratio = create_sharpe_ratio(returns, periods=252*6.5*60)
```

This completes the necessary changes. Upon execution of `intraday_mr.py` we get the following (truncated) output from the backtest simulation:

```
..
..
375072
375073
Creating summary stats...
Creating equity curve...
                     AREX   WLL      cash  commission     total   returns  \
datetime
2014-03-11 15:53:00  2098 -6802  120604.3      9721.4  115900.3 -0.000052
2014-03-11 15:54:00  2101 -6799  120604.3      9721.4  115906.3  0.000052
2014-03-11 15:55:00  2100 -6802  120604.3      9721.4  115902.3 -0.000035
2014-03-11 15:56:00  2097 -6810  120604.3      9721.4  115891.3 -0.000095
2014-03-11 15:57:00  2098 -6801  120604.3      9721.4  115901.3  0.000086
2014-03-11 15:58:00  2098 -6800  120604.3      9721.4  115902.3  0.000009
2014-03-11 15:59:00  2099 -6800  120604.3      9721.4  115903.3  0.000009
2014-03-11 16:00:00  2100 -6801  120604.3      9721.4  115903.3  0.000000
2014-03-11 16:01:00  2100 -6801  120604.3      9721.4  115903.3  0.000000
2014-03-11 16:01:00  2100 -6801  120604.3      9721.4  115903.3  0.000000

                     equity_curve  drawdown
datetime
2014-03-11 15:53:00      1.159003  0.003933
2014-03-11 15:54:00      1.159063  0.003873
2014-03-11 15:55:00      1.159023  0.003913
2014-03-11 15:56:00      1.158913  0.004023
2014-03-11 15:57:00      1.159013  0.003923
2014-03-11 15:58:00      1.159023  0.003913
2014-03-11 15:59:00      1.159033  0.003903
2014-03-11 16:00:00      1.159033  0.003903
2014-03-11 16:01:00      1.159033  0.003903
2014-03-11 16:01:00      1.159033  0.003903
[('Total Return', '15.90%'),
 ('Sharpe Ratio', '1.89'),
 ('Max Drawdown', '3.03%'),
 ('Drawdown Duration', '120718')]
Signals: 7594
Orders: 7478
Fills: 7478
```

You can see that the strategy performs adequately well during this period. It has a total return of just under 16%. The Sharpe ratio is reasonable (when compared to a typical daily strategy), but given the high-frequency nature of the strategy we should be expecting more. The major attraction of this strategy is that the maximum drawdown is low (approximately 3%). This suggests we could apply more leverage to gain more return.

The performance of this strategy can be seen in Fig 15.3:

Note that these figures are based on trading a total of 100 shares. You can adjust the leverage by simply adjusting the `generate_naive_order` method of the `Portfolio` class. Look for the

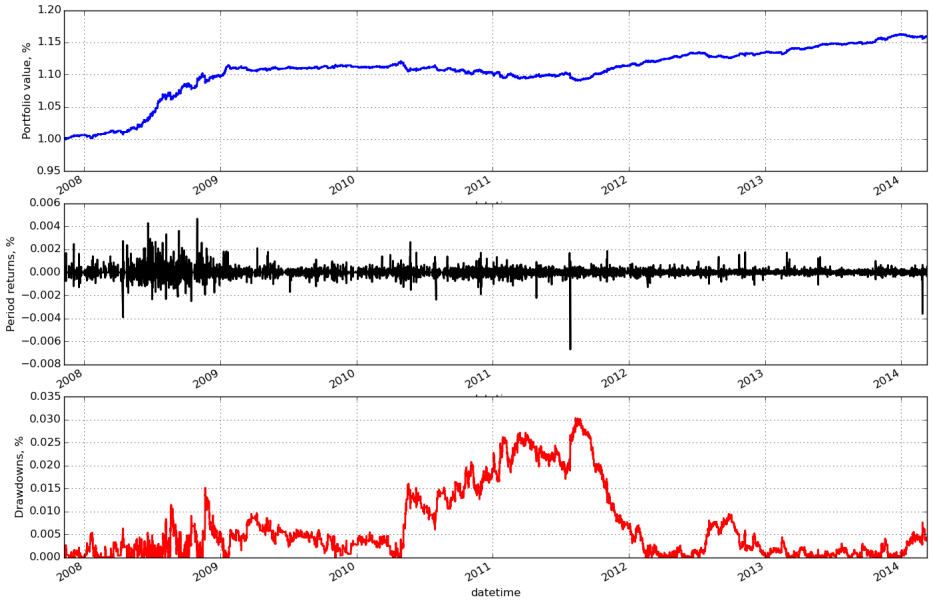


Figure 15.3: Equity Curve, Daily Returns and Drawdowns for intraday mean-reversion strategy

attribute known as `mkt_quantity`. It will be set to 100. Changing this to 2000, for instance, provides these results:

```
..
..
[('Total Return', '392.85%'),
 ('Sharpe Ratio', '2.29'),
 ('Max Drawdown', '45.69%'),
 ('Drawdown Duration', '102150')]
..
..
```

Clearly the Sharpe Ratio and Total Return are much more attractive, but we have to endure a 45% maximum drawdown over this period as well!

15.4 Plotting Performance

The three Figures displayed above are all created using the `plot_performance.py` script. For completeness I've included the code so that you can use it as a base to create your own performance charts.

It is necessary to run this in the same directory as the output file from the backtest, namely where `equity.csv` resides. The listing is as follows:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# plot_performance.py

import os.path
```

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd


if __name__ == "__main__":
    data = pd.io.parsers.read_csv(
        "equity.csv", header=0,
        parse_dates=True, index_col=0
    ).sort()

    # Plot three charts: Equity curve,
    # period returns, drawdowns
    fig = plt.figure()
    # Set the outer colour to white
    fig.patch.set_facecolor('white')

    # Plot the equity curve
    ax1 = fig.add_subplot(311, ylabel='Portfolio value, %')
    data['equity_curve'].plot(ax=ax1, color="blue", lw=2.)
    plt.grid(True)

    # Plot the returns
    ax2 = fig.add_subplot(312, ylabel='Period returns, %')
    data['returns'].plot(ax=ax2, color="black", lw=2.)
    plt.grid(True)

    # Plot the returns
    ax3 = fig.add_subplot(313, ylabel='Drawdowns, %')
    data['drawdown'].plot(ax=ax3, color="red", lw=2.)
    plt.grid(True)

    # Plot the figure
    plt.show()
```

Chapter 16

Strategy Optimisation

In prior chapters we have considered how to create both an underlying predictive model (such as with the Suppor Vector Machine and Random Forest Classifier) as well as a trading strategy based upon it. Along the way we have seen that there are many *parameters* to such models. In the case of an SVM we have the "tuning" parameters γ and C. In a Moving Average Crossover trading strategy we have the parameters for the two lookback windows of the moving average filters.

In this chapter we are going to describe optimisation methods to improve the performance of our trading strategies by tuning the parameters in a systematic fashion. For this we will use mechanisms from the statistical field of *Model Selection*, such as cross-validation and grid search. The literature on model selection and parameter optimisation is vast and most of the methods are somewhat beyond the scope of this book. I want to introduce the subject here so that you can explore more sophisticated techniques at your own pace.

16.1 Parameter Optimisation

At this stage nearly all of the trading strategies and underlying statistical models have required one or more parameters in order to be utilised. In momentum strategies using technical indicators, such as with moving averages (simple or exponential), there is a need to specify a lookback window. The same is true of many mean-reverting strategies, which require a (rolling) lookback window in order to calculate a regression between two time series. Particular statistical machine learning models such as a logistic regression, SVM or Random Forest also require parameters in order to be calculated.

The biggest danger when considering parameter optimisation is that of *overfitting* a model or trading strategy. This problem occurs when a model is trained on an *in sample* retained slice of training data and is optimised to perform well (by the appropriate performance measure), but performance degrades substantially when applied to *out of sample* data. For instance, a trading strategy could perform extremely well in the backtest (the in sample data) but when deployed for live trading can be completely unprofitable.

An additional concern of parameter optimisation is that it can become very computationally expensive. With modern computational systems this is less of an issue than it once was, due to parallelisation and fast CPUs. However, multiple parameter optimisation can increase computational complexity by orders of magnitudes. One must be aware of this as part of the research and development process.

16.1.1 Which Parameters to Optimise?

A statistical-based algorithmic trading model will often have many parameters and different measures of performance. An underlying statistical learning algorithm will have its own set of parameters. In the case of a multiple linear or logistic regression these would be the β_i coefficients. In the case of a random forest one such parameter would be the number of underlying decision trees to use in the ensemble. Once applied to a trading model other parameters might be entry

and exit thresholds, such as a z-score of a particular time series. The z-score itself might have an implicit rolling lookback window. As can be seen the number of parameters can be quite extensive.

In addition to parameters there are numerous means of evaluating the performance of a statistical model and the trading strategy based upon it. We have defined concepts such as the hit rate and the confusion matrix. In addition there are more statistical measures such as the *Mean Squared Error* (MSE). These are performance measures that would be optimised at the statistical model level, via parameters relevant to their domain.

The actual trading strategy is evaluated on different criteria, such as compound annual growth rate (CAGR) and maximum drawdown. We would need to vary entry and exit criteria, as well as other thresholds that are not directly related to the statistical model. Hence this motivates the question as to which set of parameters to optimise and when.

In the following sections we are going to optimise both the statistical model parameters, at the early research and development stage, as well as the parameters associated with a trading strategy using an underlying optimised statistical model, on each of their respective performance measures.

16.1.2 Optimisation is Expensive

With multiple real-valued parameters, optimisation can rapidly become extremely expensive, as each new parameter adds an additional spatial dimension. If we consider the example of a *grid search* (to be discussed in full below), and have a single parameter α, then we might wish to vary α within the set $\{0.1, 0.2, 0.3, 0.4, 0.5\}$. This requires 5 simulations.

If we now consider an additional parameter β, which may vary in the range $\{0.2, 0.4, 0.6, 0.8, 1.0\}$, then we will have to consider $5^2 = 25$ simulations. Another parameter, γ, with 5 variations brings this to $5^3 = 125$ simulations. If each paramater had 10 separate values to be tested, this would be equal to $10^3 = 1000$ simulations. As can be seen the parameter search space can rapidly make such simulations extremely expensive.

It is clear that a trade-off exists between conducting an exhaustive parameter search and maintaining a reasonable total simulation time. While parallelism, including many-core CPUs and graphics processing units (GPUs), have mitigated the issue somewhat, we still need to be careful when introducing parameters. This notion of reducing parameters is also an issue of model effectiveness, as we shall see below.

16.1.3 Overfitting

Overfitting is the process of optimising a parameter, or set of parameters, against a particular data set such that an appropriate performance measure (or error measure) is found to be maximised (or minimised), but when applied to an unseen data set, such a performance measure degrades substantially. The concept is closely related to the idea of the *bias-variance dilemma*.

The bias-variance dilemma concerns the situation where a statistical model has a trade-off between being a low-bias model or a low-variance model, or a compromise between the two. *Bias* refers to the difference between the model's estimation of a parameter and the true "population" value of the parameter, or erroneous assumptions in the statistical model. *Variance* refers to the error from the sensitivity of the model to small fluctuations in the training set (in sample data).

In all statistical models one is simultaneously trying to minimise both the bias error and the variance error in order to improve model accuracy. Such a situation can lead to overfitting in models, as the training error can be substantially reduced by introducing models with more flexibility (variation). However, such models can perform extremely poorly on new (out of sample) data since they were essentially "fit" to the in sample data.

A common example of a high-bias, low-variance model is that of linear regression applied to a non-linear data set. Additions of new points do not affect the regression slope dramatically (assuming they are not too far from the remaining data), but since the problem is inherently non-linear, there is a systematic bias in the results by using a linear model.

A common example of a low-bias, high-variance model is that of a polynomial spline fit applied to a non-linear data set. The parameter of the model (the degree of the polynomial)

could be adjusted to fit such a model very precisely (i.e. low-bias on the training data), but additions of new points would almost certainly lead to the model having to modify its degree of polynomial to fit the new data. This would make it a very high-variance model on the in sample data. Such a model would likely have very poor predictability or inferential capability on out of sample data.

Overfitting can also manifest itself on the trading strategy and not just the statistical model. For instance, we could optimise the Sharpe ratio by varying entry and exit threshold parameters. While this may improve profitability in the backtest (or minimise risk substantially), it would likely not be behaviour that is replicated when the strategy was deployed live, as we might have been fitting such optimisations to noise in the historical data.

We will discuss techniques below to minimise overfitting, as much as possible. However one has to be aware that it is an ever-present danger in both algorithmic trading and statistical analysis in general.

16.2 Model Selection

In this section we are going to consider how to optimise the statistical model that will underly a trading strategy. In the field of statistics and machine learning this is known as *Model Selection*. While I won't present an exhaustive discussion on the various model selection techniques, I will describe some of the basic mechanisms such as *Cross Validation* and *Grid Search* that work well for trading strategies.

16.2.1 Cross Validation

Cross Validation is a technique used to assess how a statistical model will generalise to new data that it has not been exposed to before. Such a technique is usually used on predictive models, such as the aforementioned supervised classifiers used to predict the sign of the following daily returns of an asset price series. Fundamentally, the goal of cross validation is to minimise error on out of sample data without leading to an overfit model.

In this section we will describe the *training/test split* and *k-fold cross validation*, as well as use techniques within Scikit-Learn to automatically carry out these procedures on statistical models we have already developed.

Train/Test Split

The simplest example of cross validation is known as a *training/test split*, or a *2-fold cross validation*. Once a prior historical data set is assembled (such as a daily time series of asset prices), it is split into two components. The ratio of the split is usually varied between 0.5 and 0.8. In the latter case this means 80% of the data is used for training and 20% is used for testing. All of the statistics of interest, such as the hit rate, confusion matrix or mean-squared error are calculated on the test set, which has not been used within the training process.

To carry out this process in Python with Scikit-Learn we can use the `sklearn.cross_validation.train_test_split` method. We will continue with our model as discussed in the chapter on Forecasting. In particular, we are going to modify `forecast.py` and create a new file called `train_test_split.py`. We will need to add the new import to the list of imports:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# train_test_split.py

from __future__ import print_function

import datetime

import sklearn
```

```
from sklearn.cross_validation import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.lda import LDA
from sklearn.metrics import confusion_matrix
from sklearn.qda import QDA
from sklearn.svm import LinearSVC, SVC

from create_lagged_series import create_lagged_series
```

In `forecast.py` we originally split the data based on a particular date within the time series:

```
# forecast.py

    ..

    # The test data is split into two parts: Before and after 1st Jan 2005.
    start_test = datetime.datetime(2005,1,1)

    # Create training and test sets
    X_train = X[X.index < start_test]
    X_test = X[X.index >= start_test]
    y_train = y[y.index < start_test]
    y_test = y[y.index >= start_test]
    ..
```

This can be replaced with the method `train_test_split` from Scikit-Learn in the `train_test_split.py` file. For completeness, the full `__main__` method is provided below:

```
# train_test_split.py

if __name__ == "__main__":
    # Create a lagged series of the S&P500 US stock market index
    snpret = create_lagged_series(
        "^GSPC", datetime.datetime(2001,1,10),
        datetime.datetime(2005,12,31), lags=5
    )

    # Use the prior two days of returns as predictor
    # values, with direction as the response
    X = snpret[["Lag1","Lag2"]]
    y = snpret["Direction"]

    # Train/test split
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.8, random_state=42
    )

    # Create the (parametrised) models
    print("Hit Rates/Confusion Matrices:\n")
    models = [("LR", LogisticRegression()),
              ("LDA", LDA()),
              ("QDA", QDA()),
              ("LSVC", LinearSVC()),
              ("RSVM", SVC(
                C=1000000.0, cache_size=200, class_weight=None,
                coef0=0.0, degree=3, gamma=0.0001, kernel='rbf',
                max_iter=-1, probability=False, random_state=None,
```

```
                shrinking=True, tol=0.001, verbose=False)
            ),
            ("RF", RandomForestClassifier(
              n_estimators=1000, criterion='gini',
              max_depth=None, min_samples_split=2,
              min_samples_leaf=1, max_features='auto',
              bootstrap=True, oob_score=False, n_jobs=1,
              random_state=None, verbose=0)
            )]

    # Iterate through the models
    for m in models:

        # Train each of the models on the training set
        m[1].fit(X_train, y_train)

        # Make an array of predictions on the test set
        pred = m[1].predict(X_test)

        # Output the hit-rate and the confusion matrix for each model
        print("%s:\n%0.3f" % (m[0], m[1].score(X_test, y_test)))
        print("%s\n" % confusion_matrix(pred, y_test))
```

Notice that we have picked the ratio of the training set to be 80% of the data, leaving the testing data with only 20%. In addition we have specified a `random_state` to randomise the sampling within the selection of data. This means that the data is not sequentially divided chronologically, but rather is sampled randomly.

The results of the cross-validation on the model are as follows (yous will likely appear slightly different due to the nature of the fitting procedure):

```
Hit Rates/Confusion Matrices:

LR:
0.511
[[ 70  70]
 [419 441]]

LDA:
0.513
[[ 69  67]
 [420 444]]

QDA:
0.503
[[ 83  91]
 [406 420]]

LSVC:
0.513
[[ 69  67]
 [420 444]]

RSVM:
0.506
[[  8  13]
 [481 498]]
```

```
RF:
0.490
[[200 221]
 [289 290]]
```

It can be seen that the hit rates are substantially lower than those found in the aforementioned forecasting chapter. Consequently we can likely conclude that the particular choice of training/test split lead to an over-optimistic view of the predictive capability of the classifier.

The next step is to increase the number of times a cross-validation is performed in order to minimise any potential overfitting. For this we will use k-fold cross validation.

K-Fold Cross Validation

Rather than partitioning the set into a single training and test set, we can use k-fold cross validation to *randomly* partition the the set into k equally sized subsamples. For each iteration (of which there are k), one of the k subsamples is retained as a test set, while the remaining $k-1$ subsamples together form a training set. A statistical model is then trained on each of the k folds and its performance evaluated on its specific k-th test set.

The purpose of this is to combine the results of each model into an *emsemble* by means of averaging the results of the prediction (or otherwise) to produce a single prediction. The main benefit of using k-fold cross validation is that the every predictor within the original data set is used both for training and testing only once.

This motivates a question as to how to choose k, which is now another parameter! Generally, $k = 10$ is used but one can also perform another analysis to choose an optimal value of k.

We will now make use of the `cross_validation` module of Scikit-Learn to obtain the `KFold` k-fold cross validation object. We create a new file called `k_fold_cross_val.py`, which is a copy of `train_test_split.py` and modify the imports by adding the following line:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# k_fold_cross_val.py

from __future__ import print_function

import datetime

import pandas as pd
import sklearn
from sklearn import cross_validation
from sklearn.metrics import confusion_matrix
from sklearn.svm import SVC

from create_lagged_series import create_lagged_series
```

We then need to make changes the `__main__` function by removing the `train_test_split` method and replacing it with an instance of `KFold`. It takes five parameters.

The first isthe length of the dataset, which in this case 1250 days. The second value is K representing the number of folds, which in this case is 10. The third value is `indices`, which I have set to `False`. This means that the actual index values are used for the arrays returned by the iterator. The fourth and fifth are used to randomise the order of the samples.

As before in `forecast.py` and `train_test_split.py` we obtain the lagged series of the S&P500. We then create a set of vectors of predictors (X) and responses (y). We then utilise the `KFold` object and iterate over it. During each iteration we create the training and testing sets for each of the X and y vectors. These are then fed into a radial support vector machine with identical parameters to the aforementioned files and the model is fit.

Finally the hit rate and confusion matrix for each instance of the SVM is output.

```
# k_fold_cross_val.py

if __name__ == "__main__":
    # Create a lagged series of the S&P500 US stock market index
    snpret = create_lagged_series(
        "^GSPC", datetime.datetime(2001,1,10),
        datetime.datetime(2005,12,31), lags=5
    )

    # Use the prior two days of returns as predictor
    # values, with direction as the response
    X = snpret[["Lag1","Lag2"]]
    y = snpret["Direction"]

    # Create a k-fold cross validation object
    kf = cross_validation.KFold(
        len(snpret), n_folds=10, indices=False,
        shuffle=True, random_state=42
    )

    # Use the kf object to create index arrays that
    # state which elements have been retained for training
    # and which elements have beenr retained for testing
    # for each k-element iteration
    for train_index, test_index in kf:
        X_train = X.ix[X.index[train_index]]
        X_test = X.ix[X.index[test_index]]
        y_train = y.ix[y.index[train_index]]
        y_test = y.ix[y.index[test_index]]

        # In this instance only use the
        # Radial Support Vector Machine (SVM)
        print("Hit Rate/Confusion Matrix:")
        model = SVC(
            C=1000000.0, cache_size=200, class_weight=None,
            coef0=0.0, degree=3, gamma=0.0001, kernel='rbf',
            max_iter=-1, probability=False, random_state=None,
            shrinking=True, tol=0.001, verbose=False
        )

        # Train the model on the retained training data
        model.fit(X_train, y_train)

        # Make an array of predictions on the test set
        pred = model.predict(X_test)

        # Output the hit-rate and the confusion matrix for each model
        print("%0.3f" % model.score(X_test, y_test))
        print("%s\n" % confusion_matrix(pred, y_test))
```

The output of the code is as follows:

```
Hit Rate/Confusion Matrix:
0.528
[[11 10]
 [49 55]]
```

```
Hit Rate/Confusion Matrix:
0.400
[[ 2  5]
 [70 48]]

Hit Rate/Confusion Matrix:
0.528
[[ 8  8]
 [51 58]]

Hit Rate/Confusion Matrix:
0.536
[[ 6  3]
 [55 61]]

Hit Rate/Confusion Matrix:
0.512
[[ 7  5]
 [56 57]]

Hit Rate/Confusion Matrix:
0.480
[[11 11]
 [54 49]]

Hit Rate/Confusion Matrix:
0.608
[[12 13]
 [36 64]]

Hit Rate/Confusion Matrix:
0.440
[[ 8 17]
 [53 47]]

Hit Rate/Confusion Matrix:
0.560
[[10  9]
 [46 60]]

Hit Rate/Confusion Matrix:
0.528
[[ 9 11]
 [48 57]]
```

It is the clear that the hit rate and confusion matrices vary dramatically across the various folds. This is indicative that the model is prone to overfitting, on this particular dataset. A remedy for this is to use significantly more data, either at a higher frequency or over a longer duration.

In order to utilise this model in a trading strrategy it would be necessary to combine each of these individually trained classifiers (i.e. each of the K objects) into an ensemble average and then use that combined model for classification within the strategy.

Note that technically it is not appropriate to use simple cross-validation techniques on temporally ordered data (i.e. time-series). There are more sophisticated mechanisms for coping with autocorrelation in this fashion, but I wanted to highlight the approach so we have used time series data for simplicity.

16.2.2 Grid Search

We have so far seen that k-fold cross validation helps us to avoid overfitting in the data by performing validation on every element of the sample. We now turn our attention to optimising the *hyper-parameters* of a particular statistical model. Such parameters are those not directly learnt by the model estimation procedure. For instance, C and γ for a support vector machine. In essence they are the parameters that we need to specify when calling the initialisation of each statistical model. For this procedure we will use a process known as a *grid search*.

The basic idea is to take a range of parameters and assess the performance of the statistical model on each parameter element within the range. To achieve this in Scikit-Learn we can create a `ParameterGrid`. Such an object will produce a list of Python dictionaries that each contain a parameter combination to be fed into a statistical model.

An example code snippet that produces a parameter grid, for parameters related to a support vector machine, is given below:

```
>>> from sklearn.grid_search import ParameterGrid
>>> param_grid = {'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001]}
>>> list(ParameterGrid(param_grid))

[{'C': 1, 'gamma': 0.001},
 {'C': 1, 'gamma': 0.0001},
 {'C': 10, 'gamma': 0.001},
 {'C': 10, 'gamma': 0.0001},
 {'C': 100, 'gamma': 0.001},
 {'C': 100, 'gamma': 0.0001},
 {'C': 1000, 'gamma': 0.001},
 {'C': 1000, 'gamma': 0.0001}]
```

Now that we have a suitable means of generating a `ParameterGrid` we need to feed this into a statistical model iteratively to search for an optimal performance score. In this case we are going to seek to maximise the hit rate of the classifier.

The `GridSearchCV` mechanism from Scikit-Learn allows us to perform the actual grid search. In fact, it allows us to perform not only a standard grid search but also a cross validation scheme at the same time.

We are now going to create a new file, `grid_search.py`, that once again uses `create_lagged_series.py` and a support vector machine to perform a cross-validated hyperparameter grid search. To this end we must import the correct libraries:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# grid_search.py

from __future__ import print_function

import datetime

import sklearn
from sklearn import cross_validation
from sklearn.cross_validation import train_test_split
from sklearn.grid_search import GridSearchCV
from sklearn.metrics import classification_report
from sklearn.svm import SVC

from create_lagged_series import create_lagged_series
```

As before with `k_fold_cross_val.py` we create a lagged series and then use the previous two days of returns as predictors. We initially create a training/test split such that 50% of the

data can be used for training and cross validation while the remaining data can be "held out" for evaluation.

Subsequently we create the `tuned_parameters` list, which contains a single dictionary denoting the parameters we wish to test over. This will create a *cartesian product* of all parameter lists, i.e. a list of pairs of every possible parameter combination.

Once the parameter list is created we pass it to the `GridSearchCV` class, along with the type of classifier that we're interested in (namely a radial support vector machine), with a k-fold cross-validation k-value of 10.

Finally, we train the model and output the best estimator and its associated hit rate scores. In this way we have not only optimised the model parameters via cross validation but we have also optimised the hyperparameters of the model via a parametrised grid search, all in one class! Such succinctness of the code allows significant experimentation without being bogged down by excessive "data wrangling".

```
if __name__ == "__main__":
    # Create a lagged series of the S&P500 US stock market index
    snpret = create_lagged_series(
        "^GSPC", datetime.datetime(2001,1,10),
        datetime.datetime(2005,12,31), lags=5
    )

    # Use the prior two days of returns as predictor
    # values, with direction as the response
    X = snpret[["Lag1","Lag2"]]
    y = snpret["Direction"]

    # Train/test split
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.5, random_state=42
    )

    # Set the parameters by cross-validation
    tuned_parameters = [
        {'kernel': ['rbf'], 'gamma': [1e-3, 1e-4], 'C': [1, 10, 100, 1000]}
    ]

    # Perform the grid search on the tuned parameters
    model = GridSearchCV(SVC(C=1), tuned_parameters, cv=10)
    model.fit(X_train, y_train)

    print("Optimised parameters found on training set:")
    print(model.best_estimator_, "\n")

    print("Grid scores calculated on training set:")
    for params, mean_score, scores in model.grid_scores_:
        print("%0.3f for %r" % (mean_score, params))
```

The output of the grid search cross validation procedure is as follows:

```
Optimised parameters found on training set:
SVC(C=1, cache_size=200, class_weight=None, coef0=0.0, degree=3, gamma=0.001,
  kernel='rbf', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False)

Grid scores calculated on training set:
0.541 for {'kernel': 'rbf', 'C': 1, 'gamma': 0.001}
0.541 for {'kernel': 'rbf', 'C': 1, 'gamma': 0.0001}
0.541 for {'kernel': 'rbf', 'C': 10, 'gamma': 0.001}
```

```
0.541 for {'kernel': 'rbf', 'C': 10, 'gamma': 0.0001}
0.541 for {'kernel': 'rbf', 'C': 100, 'gamma': 0.001}
0.541 for {'kernel': 'rbf', 'C': 100, 'gamma': 0.0001}
0.538 for {'kernel': 'rbf', 'C': 1000, 'gamma': 0.001}
0.541 for {'kernel': 'rbf', 'C': 1000, 'gamma': 0.0001}
```

As we can see $\gamma = 0.001$ and $C = 1$ provides the best hit rate, on the validation set, for this particular radial kernel support vector machine. This model could now form the basis of a forecasting-based trading strategy, as we have previously demonstrated in the prior chapter.

16.3 Optimising Strategies

Up until this point we have concentrated on *model selection* and optimising the underlying statistical model that (might) form the basis of a trading strategy. However, a predictive model and a functioning, profitable algorithmic strategy are two different entities. We now turn our attention to optimising parameters that have a direct effect on profitability and risk metrics.

To achieve this we are going to make use of the event-driven backtesting software that was described in a previous chapter. We will consider a particular strategy that has three parameters associated with it and search through the space formed by the cartesian product of parameters, using a grid search mechanism. We will then attempt to maximise particular metrics such as the Sharpe Ratio or minimise others such as the maximum drawdown.

16.3.1 Intraday Mean Reverting Pairs

The strategy of interest to us in this chapter is the "Intraday Mean Reverting Equity Pairs Trade" using the energy equities AREX and WLL. It contains three parameters that we are capable of optimising: The linear regression lookback period, the residuals z-score entry threshold and the residuals z-score exit threshold.

We will consider a range of values for each parameter and then calculate a backtest for the strategy across each of these ranges, outputting the total return, Sharpe ratio and drawdown characteristics of each simulation, to a CSV file for each parameter set. This will allow us to ascertain an optimised Sharpe or minimised max drawdown for our trading strategy.

16.3.2 Parameter Adjustment

Since the event-driven backtesting software is quite CPU-intensive, we will restrict the parameter range to three values per parameter. This will give us a total of $3^3 = 27$ separate simulations to carry out. The parameter ranges are listed below:

- OLS Lookback Window - $w_l \in \{50, 100, 200\}$
- Z-Score Entry Threshold - $z_h \in \{2.0, 3.0, 4.0\}$
- Z-Score Exit Threshold - $z_l \in \{0.5, 1.0, 1.5\}$

To carry out the set of simulations a cartesian product of all three ranges will be calculated and then the simulation will be carried out for each combination of parameters.

The first task is to modify the `intraday_mr.py` file to include the `product` method from the `itertools` library:

```
# intraday_mr.py

..
from itertools import product
..
```

We can then modify the `__main__` method to include the generation of a parameter list for all three of the parameters discussed above.

The first task is to create the actual paramater ranges for the OLS lookback window, the zscore entry threshold and the zscore exit threshold. Each of these has three separate variations leading to a total of 27 simulations.

Once the ranges are created the `itertools.product` method is used to create a cartesian product of all variations, which is then fed into a list of dictionaries to ensure that the correct keyword arguments are passed to the `Strategy` object.

Finally the backtest is instantiated with the `strat_params_list` forming the final keyword argument:

```
if __name__ == "__main__":
    csv_dir = '/path/to/your/csv/file'  # CHANGE THIS!
    symbol_list = ['AREX', 'WLL']
    initial_capital = 100000.0
    heartbeat = 0.0
    start_date = datetime.datetime(2007, 11, 8, 10, 41, 0)

    # Create the strategy parameter grid
    # using the itertools cartesian product generator
    strat_lookback = [50, 100, 200]
    strat_z_entry = [2.0, 3.0, 4.0]
    strat_z_exit = [0.5, 1.0, 1.5]
    strat_params_list = list(product(
        strat_lookback, strat_z_entry, strat_z_exit
    ))

    # Create a list of dictionaries with the correct
    # keyword/value pairs for the strategy parameters
    strat_params_dict_list = [
        dict(ols_window=sp[0], zscore_high=sp[1], zscore_low=sp[2])
        for sp in strat_params_list
    ]

    # Carry out the set of backtests for all parameter combinations
    backtest = Backtest(
        csv_dir, symbol_list, initial_capital, heartbeat,
        start_date, HistoricCSVDataHandlerHFT, SimulatedExecutionHandler,
        PortfolioHFT, IntradayOLSMRStrategy,
        strat_params_list=strat_params_dict_list
    )
    backtest.simulate_trading()
```

The next step is to modify the `Backtest` object in `backtest.py` to be able to handle multiple parameter sets. We need to modify the `_generate_trading_instances` method to have an argument that represents the particular parameter set on creation of a new `Strategy` object:

```
# backtest.py

..
    def _generate_trading_instances(self, strategy_params_dict):
        """
        Generates the trading instance objects from
        their class types.
        """
        print("Creating DataHandler, Strategy, Portfolio and ExecutionHandler for")
        print("strategy parameter list: %s..." % strategy_params_dict)
        self.data_handler = self.data_handler_cls(
            self.events, self.csv_dir, self.symbol_list, self.header_strings
        )
```

```
        self.strategy = self.strategy_cls(
            self.data_handler, self.events, **strategy_params_dict
        )
        self.portfolio = self.portfolio_cls(
            self.data_handler, self.events, self.start_date,
            self.num_strats, self.periods, self.initial_capital
        )
        self.execution_handler = self.execution_handler_cls(self.events)
..
```

This method is now called within a strategy parameter list loop, rather than at construction of the `Backtest` object. While it may seem wasteful to recreate all of the data handlers, event queues and portfolio objects for each parameter set, it ensures that all of the iterators have been reset and that we are truly starting with a "clean slate" for each simulation.

The next task is to modify the `simulate_trading` method to loop over all variants of strategy parameters. The method creates an output CSV file that is used to store parameter combinations and their particular performance metrics. This will allow us later to plot performance characteristics across parameters.

The method loops over all of the strategy parameters and generates a new trading instance on every simulation. The backtest is then executed and the statistics calculated. These are stored and output into the CSV file. Once the simulation ends, the output file is closed:

```
# backtest.py

..
    def simulate_trading(self):
        """
        Simulates the backtest and outputs portfolio performance.
        """
        out = open("output/opt.csv", "w")

        spl = len(self.strat_params_list)
        for i, sp in enumerate(self.strat_params_list):
            print("Strategy %s out of %s..." % (i+1, spl))
            self._generate_trading_instances(sp)
            self._run_backtest()
            stats = self._output_performance()
            pprint.pprint(stats)

            tot_ret = float(stats[0][1].replace("%",""))
            cagr = float(stats[1][1].replace("%",""))
            sharpe = float(stats[2][1])
            max_dd = float(stats[3][1].replace("%",""))
            dd_dur = int(stats[4][1])

            out.write(
                "%s,%s,%s,%s,%s,%s,%s,%s\n" % (
                    sp["ols_window"], sp["zscore_high"], sp["zscore_low"],
                    tot_ret, cagr, sharpe, max_dd, dd_dur
                )
            )

        out.close()
```

On my desktop system, this process takes some time! 27 parameter simulations across more than 600,000 data points per simulation takes around 3 hours. The backtester has not been parallelised at this stage, so concurrent running of simulation jobs would make the process a lot faster. The output of the current parameter study is given below. The columns are given by

OLS Lookback, ZScore High, ZScore Low, Total Return (%), CAGR (%), Sharpe, Max DD (%), DD Duration (minutes):

```
50,2.0,0.5,213.96,20.19,1.63,42.55,255568
50,2.0,1.0,264.9,23.13,2.18,27.83,160319
50,2.0,1.5,167.71,17.15,1.63,60.52,293207
50,3.0,0.5,298.64,24.9,2.82,14.06,35127
50,3.0,1.0,324.0,26.14,3.61,9.81,33533
50,3.0,1.5,294.91,24.71,3.71,8.04,31231
50,4.0,0.5,212.46,20.1,2.93,8.49,23920
50,4.0,1.0,222.94,20.74,3.5,8.21,28167
50,4.0,1.5,215.08,20.26,3.66,8.25,22462
100,2.0,0.5,378.56,28.62,2.54,22.72,74027
100,2.0,1.0,374.23,28.43,3.0,15.71,89118
100,2.0,1.5,317.53,25.83,2.93,14.56,80624
100,3.0,0.5,320.1,25.95,3.06,13.35,66012
100,3.0,1.0,307.18,25.32,3.2,11.57,32185
100,3.0,1.5,306.13,25.27,3.52,7.63,33930
100,4.0,0.5,231.48,21.25,2.82,7.44,29160
100,4.0,1.0,227.54,21.01,3.11,7.7,15400
100,4.0,1.5,224.43,20.83,3.33,7.73,18584
200,2.0,0.5,461.5,31.97,2.98,19.25,31024
200,2.0,1.0,461.99,31.99,3.64,10.53,64793
200,2.0,1.5,399.75,29.52,3.57,10.74,33463
200,3.0,0.5,333.36,26.58,3.07,19.24,56569
200,3.0,1.0,325.96,26.23,3.29,10.78,35045
200,3.0,1.5,284.12,24.15,3.21,9.87,34294
200,4.0,0.5,245.61,22.06,2.9,12.52,51143
200,4.0,1.0,223.63,20.78,2.93,9.61,40075
200,4.0,1.5,203.6,19.55,2.96,7.16,40078
```

We can see that for this particular study the parameter values of $w_l = 50$, $z_h = 3.0$ and $z_l = 1.5$ provide the best Sharpe ratio at $S = 3.71$. For this Sharpe ratio we have a total return of 294.91% and a maximum drawdown of 8.04%. The best total return of 461.99%, albeit with a maximum drawdown of 10.53% is given by the parameter set of $w_l = 200$, $z_h = 2.0$ and $z_l = 1.0$.

16.3.3 Visualisation

As a final task in strategy optimisation, we are now going to visualise the performance characteristics of the backtester using Matplotlib, which is an extremely useful step when carrying out initial strategy research. Unfortunately we are the situation where we have a three-dimensional problem and so performance visualisation is not straightforward! However, there are some partial remedies to the situation.

Firstly, we could fix the value of one parameter and take a "parameter slice" through the remainder of the "data cube". For instance we could fix the lookback window to be 100 and then see who the variation in z-score entry and exit thresholds affects the Sharpe Ratio or the maximum drawdown.

To achieve this we will use Matplotlib. We will read the output CSV and reshape the data such that we can visualise the results.

Sharpe/Drawdown Heatmap

We will fix the lookback period of $w_l = 100$ and then generate a 3×3 grid and "heatmap" of the Sharpe ratio and maximum drawdown for the variation in z-score thresholds.

In the following code we import the output CSV file. The first task is to filter out the lookback periods that are not of interest ($w_l \in \{50, 200\}$). Then we reshape the remaining performance data into two 3×3 matrices. The first represents the Sharpe ratio for each z-score threshold combination while the second represents maximum drawdown.

Here is the code for creating the Sharpe Ratio heatmap. We first import Matplotlib and NumPy. Then we define a function called `create_data_matrix` which reshapes the Sharpe Ratio data into a 3×3 grid. Within the `__main__` function we open the CSV file (make sure to change the path on your system!) and exclude any lines not referencing a lookback period of 100.

We then create a blue-shaded heatmap and apply the correct row/column labels using the z-score thresholds. Subsequently we place the actual value of the Sharpe Ratio onto the heatmap. Finally, we set the ticks, labels, title and then plot the heatmap:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# plot_sharpe.py

import matplotlib.pyplot as plt
import numpy as np


def create_data_matrix(csv_ref, col_index):
    data = np.zeros((3, 3))
    for i in range(0, 3):
        for j in range(0, 3):
            data[i][j] = float(csv_ref[i*3+j][col_index])
    return data


if __name__ == "__main__":
    # Open the CSV file and obtain only the lines
    # with a lookback value of 100
    csv_file = open("/path/to/opt.csv", "r").readlines()
    csv_ref = [
        c.strip().split(",")
        for c in csv_file if c[:3] == "100"
    ]
    data = create_data_matrix(csv_ref, 5)

    fig, ax = plt.subplots()
    heatmap = ax.pcolor(data, cmap=plt.cm.Blues)
    row_labels = [0.5, 1.0, 1.5]
    column_labels = [2.0, 3.0, 4.0]

    for y in range(data.shape[0]):
        for x in range(data.shape[1]):
            plt.text(x + 0.5, y + 0.5, '%.2f' % data[y, x],
                     horizontalalignment='center',
                     verticalalignment='center',
                     )

    plt.colorbar(heatmap)

    ax.set_xticks(np.arange(data.shape[0])+0.5, minor=False)
    ax.set_yticks(np.arange(data.shape[1])+0.5, minor=False)
    ax.set_xticklabels(row_labels, minor=False)
    ax.set_yticklabels(column_labels, minor=False)

    plt.suptitle('Sharpe Ratio Heatmap', fontsize=18)
    plt.xlabel('Z-Score Exit Threshold', fontsize=14)
```

```
    plt.ylabel('Z-Score Entry Threshold', fontsize=14)
    plt.show()
```

The plot for the maximum drawdown is almost identical with the exception that we use a red-shaded heatmap and alter the column index in the `create_data_matrix` function to use the maximum drawdown percentage data.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# plot_drawdown.py

import matplotlib.pyplot as plt
import numpy as np


def create_data_matrix(csv_ref, col_index):
    data = np.zeros((3, 3))
    for i in range(0, 3):
        for j in range(0, 3):
            data[i][j] = float(csv_ref[i*3+j][col_index])
    return data


if __name__ == "__main__":
    # Open the CSV file and obtain only the lines
    # with a lookback value of 100
    csv_file = open("/path/to/opt.csv", "r").readlines()
    csv_ref = [
        c.strip().split(",")
        for c in csv_file if c[:3] == "100"
    ]
    data = create_data_matrix(csv_ref, 6)

    fig, ax = plt.subplots()
    heatmap = ax.pcolor(data, cmap=plt.cm.Reds)
    row_labels = [0.5, 1.0, 1.5]
    column_labels = [2.0, 3.0, 4.0]

    for y in range(data.shape[0]):
        for x in range(data.shape[1]):
            plt.text(x + 0.5, y + 0.5, '%.2f%%' % data[y, x],
                     horizontalalignment='center',
                     verticalalignment='center',
                     )

    plt.colorbar(heatmap)

    ax.set_xticks(np.arange(data.shape[0])+0.5, minor=False)
    ax.set_yticks(np.arange(data.shape[1])+0.5, minor=False)
    ax.set_xticklabels(row_labels, minor=False)
    ax.set_yticklabels(column_labels, minor=False)

    plt.suptitle('Maximum Drawdown Heatmap', fontsize=18)
    plt.xlabel('Z-Score Exit Threshold', fontsize=14)
    plt.ylabel('Z-Score Entry Threshold', fontsize=14)
    plt.show()
```

The heatmaps produced from the above snippets are given in Fig 16.3.3 and Fig 16.3.3:

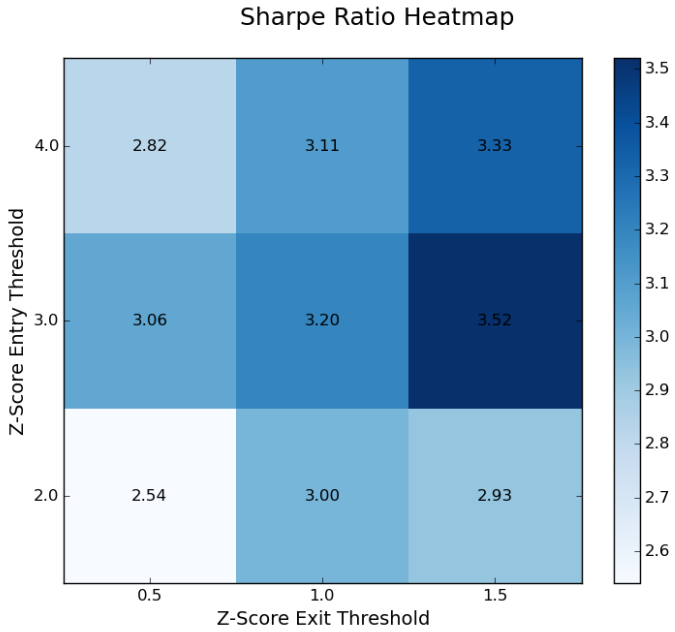


Figure 16.1: Sharpe Ratio heatmap for z-score entry/exit thresholds

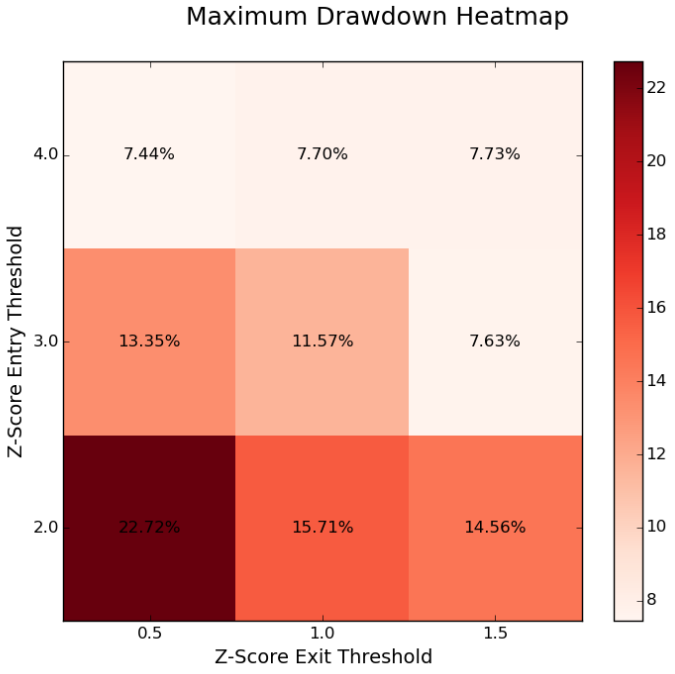


Figure 16.2: Maximum Drawdown heatmap for z-score entry/exit thresholds

At $w_l = 100$ the differences betwee the smallest and largest Sharpe Ratios, as well as the smallest and largest maximum drawdowns, is readily apparent. The Sharpe Ratio is optimised for larger entry and exit thresholds, while the drawdown is minimised in the same region. The Sharpe Ratio and maximum drawdown are at their worst when both the entry and exit thresholds are low.

This clearly motivates us to consider using relatively high entry and exit thresholds for this strategy when deployed into live trading.

Bibliography

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

[2] David. Barber. *Bayesian Reasoning and Machine Learning*. Cambridge University Press, 2012.

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

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

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

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

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

[8] Tibshirani Robert. Friedman Jerome. Hastie, Trevor. *The Elements of Statistical Learning: Data Mining, Inference and Prediction, 2nd Ed*. Springer, 2011.

[9] Witten Daniela. Hastie Trevor. Tibshirani Robert. James, Gareth. *An Introduction to Statistical Learning: with applications in R*. Springer, 2013.

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

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

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

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

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

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

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

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