

Automated Trading System for The Betfair Exchange

William Jack Alcock

Supervisor: Dr Martin Berger

Candidate Number: 105860

University of Sussex BSc Computer Science

April 4, 2016

Statement Of Originality

This report is submitted as part requirement for the degree BSc Computer Science at the University of Sussex. It is the product of my own labour except where specifically stated in the text. The report may be freely copied and distributed provided the source is acknowledged.

William Jack Alcock
April 4, 2016

Acknowledgements

I would like to acknowledge and thank

- Dr Martin Berger, University of Sussex for his encouragement, advice and support while supervising the project.
- Dr Novi Quadrianto, University of Sussex for his advice and help with machine learning models.
- Neil Thomas from Betfair and Alan Weber from Fracsoft for supplying historical Betfair market data
- Sue, Elizabeth, Poppy, Alex, Mum and Dad for their constant support and encouragement.

Summary

One page of content to be added here outlining what was implemented over the course of the project and what the final results were.

Contents

1	Statement Of Originality	1
2	Acknowledgements	2
3	Summary	3
	List of Figures	6
	List of Tables	7
4	Introduction	8
4.1	Motivation	8
4.2	Background	8
4.2.1	Betting exchanges opposed to historical bookmakers	8
4.2.2	Similarities to financial markets	9
4.2.3	Trading as opposed to gambling	10
4.2.4	Automated Trading	10
4.3	Project Objectives	10
4.3.1	First section objectives	10
4.3.2	Second section objectives	11
4.3.3	Changes in the objectives from the interim report	11
5	Professional Considerations	12
5.1	Public Interest	12
5.2	Professional Competence and Integrity	12
5.3	Duty to Relevant Authority	13
5.4	Duty to Profession	13
6	Requirements Analysis	14
6.1	The Users	14
7	Development Approach	15
7.1	Programming Languages	15
7.2	Unit Tests	15
7.3	Version Control	16
7.4	Build System	16
7.5	IDE	16
8	Design And Implementation	17
8.1	Server	17
8.1.1	Service Layer	18
8.1.2	Simulated Service Layer	18
8.1.3	Core Server Architecture	20

8.1.4	Central Configuration	21
8.1.5	The Event Bus	21
8.1.6	Data Provider	21
8.1.7	Data Model	25
8.1.8	Order Manager	25
8.1.9	Navigation Data Service	26
8.1.10	Data Store	27
8.1.11	Auto Trader	27
8.1.12	Controller	28
8.2	Web Server and Client (GUI)	29
8.2.1	Web Server	29
8.2.2	Client (GUI)	30
9	Predicting Price Movement	33
9.1	Technical Analysis	33
9.2	Historical Market Data	33
9.3	Preprocessing the Data	33
9.3.1	Betfair Price Data	34
9.3.2	Technical Analysis Indicators	34
9.4	Choosing a Model	35
9.4.1	Random Forest	36
9.5	Implementation	36
9.5.1	Extremely Random Forest	36
9.6	Optimisation	37
9.7	Results	37
9.8	Integration With The Trading System	38
10	Appendices	39
	Bibliography	40

List of Figures

4.1	Example Match Odds: Swansea vs Arsenal	9
8.1	Server Layers	17
8.2	Live and Simulated Service Layer	19
8.3	Example Back and Lay Volume	19
8.4	Core Server Architecture	20
8.5	ListMarketBook Price Projections	22
8.6	ListMarketBook Market Data Limits	22
8.7	Data Provider Polling Mechanism	24
8.8	Life Cycle of an Order	26
8.9	Web Server and Client Architecture	30
8.10	Betfair Price Increments	31
8.11	GUI Main Window	31
8.12	GUI Ladder Window	32
9.1	Decision Tree for Mammal Classification	35
9.2	Algorithm for growing a Decision Tree for an Extremely Random Forest	37

List of Tables

4.1	Example PnL Breakdown: Swansea vs Arsenal	9
8.1	Number of markets/call by price projection	23
9.1	Implied Probability of Decimal Odds	34

Introduction

4.1 Motivation

The gambling industry has been around for centuries but in recent years the nature of traditional betting markets has been revolutionised by the advent of electronic betting exchanges.

A betting exchange allows gamblers to bet against each other rather than against a bookmaker. Users can place orders on a given market to back or lay at any chosen price and as a result the contracts hosted on electronic betting exchanges share many similarities with financial markets. Betfair is one of the leading betting exchanges processing over 1.2 billion bets a year, with a trading value of £56 billion[2]

Automated trading on financial markets is wide spread and has been very well documented[42]. Now with the emergence of electronic betting exchanges it is possible to automate trading on the sporting and cultural events hosted by these exchanges applying similar strategies as those used on financial markets.

Betfair currently exposes the NG API (Application Programmers interface) to users allowing them to programatically interact with the exchange. Using the API a user can request data on available markets and place orders into a chosen market.

4.2 Background

4.2.1 Betting exchanges opposed to historical bookmakers

Historically if an individual wanted to bet on the outcome of an event they would go to a bookmaker (usually a company) who would offer them odds at which they could bet on the event. For example:

Event: Swansea vs Arsenal
Market: Swansea to Win
Odds: 2/1

In this example the bookmaker is offering odds of 2/1 for the individual to back. The bookmaker stands to lose two units for every one wagered by the individual. Bookmakers would aim to guarantee themselves a profit by offering odds lower than those they predicted to be the true odds of the event.

Betting exchanges revolutionised the practice of betting by allowing the individual the ability to also offer odds on the outcome of an event, effectively allowing them to become the bookmaker. The proprietor of the betting exchange (in this case Betfair) makes money by taking a small commission on all winning bets by simply providing a forum for individuals to place bets with each other.

As users of the exchange place orders to back or lay an event, the odds (or price) at which the back and lay orders meet derives the available price for the event.




3 selections			Back all	Lay all		
 Swansea	4.6 £2848	4.7 £4723	4.8 £1095	4.9 £1485	5 £3038	5.1 £2576
 Arsenal	1.81 £5397	1.82 £1367	1.83 £1239	1.84 £1594	1.85 £3590	1.86 £3135
 The Draw	3.9 £4016	3.95 £4276	4 £3642	4.1 £6103	4.2 £6831	4.3 £6430

Figure 4.1: Example Match Odds: Swansea vs Arsenal

As you can see in Figure 4.1, there are lay orders in the market for Swansea to win @ 4.6, 4.7 and 4.8 with back orders for Swansea to win @ 4.9, 5 and 5.1, as such the best available price to back is 4.8 (matching the lay orders), and to lay 4.9 (matching the back orders). Prices on betting exchanges are typically displayed as decimals rather than fractions with 4.8 representing 3.8/1, i.e. $4.8 * stake$ returned.

4.2.2 Similarities to financial markets

As with financial markets the price of a betting contract will fluctuate depending on the amount of money placed into the market (liquidity) and at what price it is placed. In this respect the behaviour of the markets available on a betting exchange mirror that of financial markets where for instance, the price of Vodafone stock is derived by the price at which the orders to buy it meet those to sell it.

Because the price of betting contracts will fluctuate over time users can speculate on the price of an event, potentially profiting by laying when the price is low and backing when the price is high. By predicting the odds of an event or the direction in which they believe the market will move a user can decide what price they want to back or lay a given contract.

To give an example assume a user has placed the following bets:

Lay Swansea to win @ 2.00
Back Swansea to win @ 4.00

Side	Size	Price (£)	Profit	
			Win (£)	Lose (£)
Lay	2.00	2.00	-4.00	2.00
Back	2.00	4.00	8.00	-2.00
Total			4.00	0.00

Table 4.1: Example PnL Breakdown: Swansea vs Arsenal

As shown in Table 4.1 by placing both these trades the user has removed the downside from their bet. They effectively have a free back bet for Swansea to win @ 2.00.

This free back bet can then be used to ensure the user wins a lesser amount on all eventualities of the event by backing the other outcomes (Arsenal to Win / The Draw) for a smaller amount.

4.2.3 Trading as opposed to gambling

The vast majority of individuals who bet on an event are gamblers. They will bet because they think Swansea will win or they may support Swansea or they may want the thrill of betting and potentially winning money. They will be price insensitive and they will almost certainly be backing an outcome rather than laying it.

A trader on the other hand will be interested in where the price is compared to where they think the price will move to and this will affect the price at which they are willing to back or lay an event. In contrast to the gambler the trader will probably not keep the bet on by the time the event has been realised having already closed the trade (matching the back bet with a corresponding lay bet or visa versa).

4.2.4 Automated Trading

Automating the trading of events offers several benefits to manually trading them. An automated system has the potential to place orders faster than a human and to monitor the behaviour of a wider range of markets simultaneously.

An automated system also has the potential to be more disciplined with its trading strategy than a human as its decisions are never affected by emotion, which can be a common problem with human traders. To give an example a trader will buy at a given price with the intention of exiting the trade for a loss if the market dips below a certain price. But in reality once the market reaches this price emotion can cloud their decision and they will keep the trade on, telling themselves that they know they are right and soon the price will start to climb.

4.3 Project Objectives

This project will be split into two sections:

- The first section will focus on designing and building a computer trading system for the Betfair exchange through which the user can action trades manually and run a predefined automated trading strategy on a given market.
- The second section will focus on applying machine learning techniques to historical market data to design and build a model that can be used as an indicator by an automated trading strategy.

4.3.1 First section objectives

- Write a trading system for the Betfair exchange that can:
 - Retrieve and store market data from the Betfair exchange concurrently over multiple markets.
 - Place, cancel and amend orders at the Betfair exchange concurrently over multiple markets.
 - Track and broadcast the state of orders the system has placed at the Betfair exchange.
 - Track metrics concurrently over multiple markets.

- Track a users position concurrently over multiple markets.
- Record market data and trading activity to allow the user to review it at a later date.
- Follow and action a predefined trading strategy.
- Write a GUI for the trading system that should allow the user to:
 - Navigate markets on the exchange.
 - View market information in real time.
 - Manually action trades.
 - Action a predefined trading strategy on a given number of markets.
 - Review and compare historical market and trading data.

4.3.2 Second section objectives

- Analyse historical exchange data and select a machine learning model to use on the data.
- Train, optimise and test the chosen model using the historical data.
- Report on the results produced when testing the model.
- Integrate the trained model into the trading system built in the first section of the project.
- Write a trading strategy that uses the model to make decisions and report on its success.

4.3.3 Changes in the objectives from the interim report

The interim report included the following objectives which have not been included in the final report:

- Define a language so a user can describe a trading strategy.
- Write a parser or compiler to convert a trading strategy written in the above language into steps the trading system can understand.
- Create trading strategies of differing complexities and test and report on their performance.

As the project progressed these objectives were sidelined in favour of placing more emphasis on the machine learning section of the project which was only listed as an extended objective in the interim report.

Professional Considerations

To ensure the integrity of this project it is important to comply with the British Society of Computing (BSC) Code of Conduct. Each of the four sections of the BSC Code of Conduct will be discussed:

5.1 Public Interest

From the Code of Conduct: [\[24\]](#)

You shall:

- *Have due regard for public health, privacy, security and wellbeing of others and the environment.*
- *Have due regard for the legitimate rights of Third Parties*.*
- *Conduct your professional activities without discrimination on the grounds of sex, sexual orientation, marital status, nationality, colour, race, ethnic origin, religion, age or disability, or of any other condition or requirement*
- *Promote equal access to the benefits of IT and seek to promote the inclusion of all sectors in society wherever opportunities arise.*

This does not apply to this project as no user information will be collected and the project does not involve interaction with any Third Parties

5.2 Professional Competence and Integrity

From the Code of Conduct: [\[24\]](#)

You shall:

- *Only undertake to do work or provide a service that is within your professional competence.*
- *NOT claim any level of competence that you do not possess.*
- *Develop your professional knowledge, skills and competence on a continuing basis, maintaining awareness of technological developments, procedures, and standards that are relevant to your field.*
- *Ensure that you have the knowledge and understanding of Legislation* and that you comply with such Legislation, in carrying out your professional responsibilities.*
- *Respect and value alternative viewpoints and, seek, accept and offer honest criticisms of work.*
- *Avoid injuring others, their property, reputation, or employment by false or malicious or negligent action or inaction.*

- *Reject and will not make any offer of bribery or unethical inducement.*

This project will adhere to this section of the Code of Conduct and the author has been assessed to ensure they have the correct level of competency to undertake the project. Once the project has been completed they will meet with the relevant supervisors to discuss the decisions taken during the project.

5.3 Duty to Relevant Authority

From the Code of Conduct: [\[24\]](#)

You shall:

- *Carry out your professional responsibilities with due care and diligence in accordance with the Relevant Authoritys requirements whilst exercising your professional judgement at all times.*
- *Seek to avoid any situation that may give rise to a conflict of interest between you and your Relevant Authority.*
- *Accept professional responsibility for your work and for the work of colleagues who are defined in a given context as working under your supervision.*
- *NOT disclose or authorise to be disclosed, or use for personal gain or to benefit a third party, confidential information except with the permission of your Relevant Authority, or as required by Legislation*
- *NOT misrepresent or withhold information on the performance of products, systems or services (unless lawfully bound by a duty of confidentiality not to disclose such information), or take advantage of the lack of relevant knowledge or inexperience of others.*

The author adhere to this section of the Code of Conduct.

5.4 Duty to Profession

From the Code of Conduct: [\[24\]](#)

You shall:

- *Accept your personal duty to uphold the reputation of the profession and not take any action which could bring the profession into disrepute.*
- *Seek to improve professional standards through participation in their development, use and enforcement.*
- *Uphold the reputation and good standing of BCS, the Chartered Institute for IT.*
- *Act with integrity and respect in your professional relationships with all members of BCS and with members of other professions with whom you work in a professional capacity.*
- *Notify BCS if convicted of a criminal offence or upon becoming bankrupt or disqualified as a Company Director and in each case give details of the relevant jurisdiction.*
- *Encourage and support fellow members in their professional development*

The author adhere to this section of the Code of Conduct

Requirements Analysis

The requirements were formed firstly by looking at the Betfair website and existing commercially available trading systems for electronic betting exchanges. Secondly by looking at the Betfair NG API and thirdly by looking at a basic betting strategy and formulating some requirements for a language to describe that strategy. This section will discuss pros and cons of existing solutions, what an ideal solution would entail and what parts of that solution can feasibly be implemented in the given time frame and should therefore be included in the project.

6.1 The Users

The users of the system will be traders. Their primary objective will be to effectively trade a chosen market thereby maximising profits and minimising losses.

Manually trading

- Ability to navigate markets
- Receive fast and regular updates on market activity.
- Ability to quickly place and alter orders in a given market
- Quick and accurate notifications when orders are matched.

Automated trading

- A simple and consistent way to describe a strategy that the system can understand and process.
- Ability to test a strategies behaviour without placing orders in a live market.
- Ability to review what orders the system has placed and when and why it placed those orders.
- Ability to review and replay the systems behaviour after the fact.

The software will need to be well tested and bug free as any unexpected behaviour could result in substantial financial losses.

This section will be rewritten before the final draft. I would like to completely rewrite the requirements analysis from the interim report and I am unsure how to proceed.

How closely should I stick to the interim report ?

The GUI comparison of existing systems in the interim report seems unnecessary.

I want to remove the section about creating a programming language for the user to defined a trading strategy as this hasn't been included in the implementation.

What format is required ? I would like to keep it brief outlining some functional and non functional requirements in bullet point form.

Development Approach

7.1 Programming Languages

The majority of the project will be written in Scala[27] with the exception of the GUI. The main reasons for selecting Scala above other languages were:

- Support for both functional and object orientated programming styles will help increase development speed.
- Scala code is compiled to Java Byte Code and runs on the Java Virtual Machine (JVM), as such it will run across multiple platforms and operating systems.
- There are a wide range of mature libraries available for Scala and because Scala runs on the JVM it also has access to all existing Java libraries.
- Scala encourages the use of immutable data types. This will provide an advantage considering the proposed concurrent nature of this project.
- Scala is a statically typed language because of the complex nature of this project it will be advantageous to have type errors caught at compile time.

The Akka library will be used in conjunction with Scala to build the server architecture. Akka is a toolkit and runtime for building highly concurrent distributed and resilient message-driven applications[18]. Akka is based on Erlang's Actor model, with some differences[29]. An actor provides a container for mutable State or Behaviour. Other parts of the system can communicate with an actor by sending it messages which are queued and actioned one at a time. This means that instead of having to synchronize access to parts of the code using locks, the developer can write your code in an Actor knowing it will only ever be run synchronously[1].

The GUI will be implemented in a web browser and as such will be written in JavaScript with HTML 5 and CSS. Because of the well documented difficulties writing complex applications in JavaScript[17] CoffeeScript will be used to simplify the code and AngularJS to add structure to the code and increase the declarative nature of the HTML. More in-depth reasons for choosing these technologies will be discussed later.

7.2 Unit Tests

Due to the complex nature of this project and the financial risks faced by producing trading software with bugs it has been decided that during development there will be emphasis on automated testing.

Unit tests will be produced to accompany the source code using ScalaTest and ScalaMock. The advantages of Unit tests are:

- They allow the programmer to prove the output of a given piece of code given a deterministic input.
- They allow the programmer to change existing code or add new code with the confidence that they haven't broken existing functionality code.
- They help document and define what the code is doing, looking at the unit tests gives a clearer indication of what the code is doing.
- Writing unit tests encourages the programmer to write code in small chunks that can be tested independently.

Due to the exploratory nature of the development process, it will be hard to write the unit tests before the code but where possible this will be done. In most cases unit tests will be written alongside the code. When existing code needs to be changed or bugs are found the unit tests will be changed first and then the corresponding code changed to ensure unit tests pass[43].

The Unit tests produced can be broadly split into two types:

- State tests verify that the code under test returns the right results.
- Behaviour tests verify that the code under test calls certain methods with the correct order and with the correct parameters.

7.3 Version Control

Version control will be key to the development of this project. As such I have selected to use GIT[15]. GIT is free, reliable, very well documented and has a large support community on the internet. Its also widely used in industry and the author wanted the opportunity to become proficient in using it.

7.4 Build System

The Scala Build Tool (SBT) will be used to manage dependencies and builds because of its ease of use and integration with all major Scala libraries.

7.5 IDE

IntelliJ IDEA [4] will be used as the integrated development environment (IDE) to write the code in. The author has previous experience using IntelliJ. JetBrains licence the product for free to students and it provides good support and integration for Scala, GIT and SBT.

Design And Implementation

It was decided that the system should implement a distributed client-server architecture.

The server will provide an interface for the client, respond to commands from the client and broadcast any resulting output. The server should be able to handle connections from multiple clients at once.

The client should be able to take many forms i.e. a GUI that takes commands from a user or an automated computer system.

The distributed client-server architecture was picked for the following reasons:

- The client and server can be developed in isolation.
- The server will have central control of data and resources.
- The server can be accessible from any location on any machine.

8.1 Server

As stated earlier in this report, the server will be written in Scala and built using Akka.

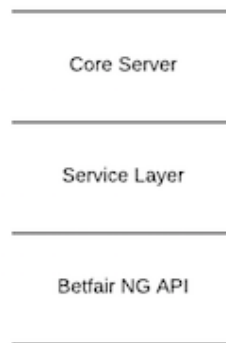


Figure 8.1: Server Layers

As shown in Figure 8.1 the server will be split into two layers:

- A service layer that provides communication with the Betfair NG API.
- The core server that will communicate with the service layer and any external systems.

This design offers the following advantages and was inspired by the 5 layer internet protocol stack[26]:

- Betfair API changes, as it did in 2014[8]. The service layer can be changed without changing the core server.
- If in the future the system needs to be adapted to use another exchange[38] this can be achieved without significant changes to the core server.

and With this design if the in the same context, This design was inspired by the 5 layer internet protocol stack

8.1.1 Service Layer

Betfair provide sample code for a number of languages to show how basic interaction with the exchange works[36]. The Scala code[32] uses Spray[23], which is built on Akka, to send and receive HTTP calls to and from the Betfair NG API. All the data sent and received from the Betfair NG API is encoded as JSON and as such the example code uses Play’s ScalaJson library[12] to convert JSON to Scala case classes and visa-versa. Both Spray and ScalaJson are well documented, well supported and used by a wide range of commercial products. Spray is built on Akka and as such integrates with existing project choices and ScalaJson provides a very concise API to validate and convert JSON into Scala case classes. For these reasons it was decided to extend the example code, adding all the missing API calls and extending the tests to cover these calls, and use that as the basis for the Service Layer. The examples in this code proved not only a good introduction to the Betfair NG API but also to Scala.

Betfair provide two environments to query for data: The live environment and a test environment that responds with delayed data. There are a number of limitations with the test environment:

1. The test environment returns delayed price data which behaves erratically compared to the live environment.
2. The test environment does not allow betting transactions to be performed on it and as such you cannot place or change orders when using it[31].

At this point it became obvious that a simulated order book would need to be implemented so the user could test a trading strategy locally without placing orders at the exchange.

8.1.2 Simulated Service Layer

As is shown in Figure 8.2 the Live Service Layer makes data requests to the exchange and place orders at the exchange. The Simulated Service Layer places orders in a local order book but still makes data requests to the exchange. Once these data requests have been received by the Simulated Service Layer it consults its order book, updates any orders in its order book that would have been matched and updates the data requests to include those locally held orders. As far as the core server is concerned the resulting data request will look identical to that outputted by the Live Service Layer.

Because the service layer doesn’t have access to the order book at the exchange some assumptions have been made as to how orders were matched.

At the exchange when an order is placed at a given price it will be assigned a queue position in the volume at that price.

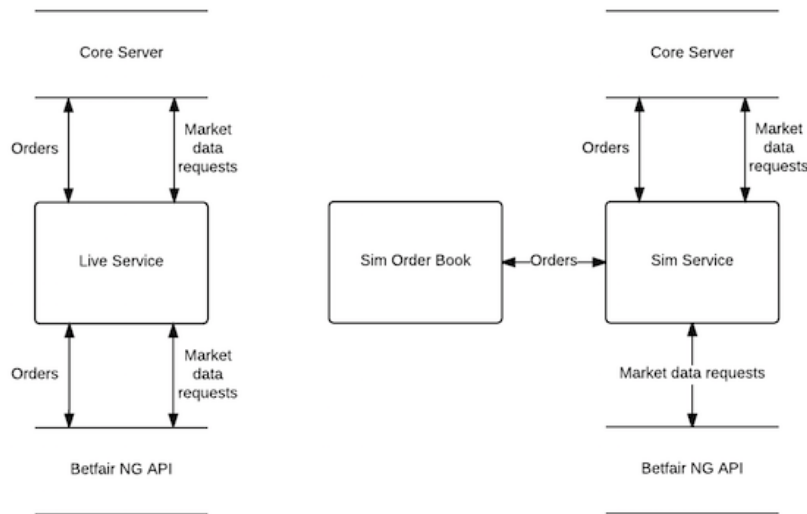


Figure 8.2: Live and Simulated Service Layer

Back	Lay
8 £18	19.5 £12

Figure 8.3: Example Back and Lay Volume

To give an example: Figure 8.3 shows that there is £12 of back orders in the market @ 19.5 (volume available to Lay) and £18 of lay orders in the market @ 8 (volume available to Back). If you placed a back order for £2 with the exchange @ 19.5 it would be placed at the back of a queue to be matched behind the £12 that is currently there and would not be matched until the volume in the queue before it had been matched, or cancelled. Because the simulated service doesn't know if the volume before it in the queue has been matched or cancelled it will make an assumption that locally held orders are not matched until all the volume at that price has traded and there is an order on the opposite side of the order book at the given price. In this case there would have to be a Lay order @ 19.5 for the order to be matched. This could result in orders taking longer to match on the Simulated Service Layer if more volume appeared in the queue after your order as it will not be matched until all the volume at the price has traded. In this respect the simulated order book provides the user with a worse case scenario, but this means if a trading strategy proves successful when tested against the Simulated Service Layer it would also have been successful when actioned on the Live Service Layer.

This functionality was implemented in the Simulated Order Book by persisting two queues, one of Lay orders in descending price order, and one of Back orders in ascending price order. When the price for a given side (Back or Lay) is updated all unmatched orders in the opposite queue are matched until an order is reached that has a price less than the current price (in the case of Lay orders) or greater than the current price (in the case of Back orders).

Using this algorithm the Simulated Order Book will rarely have to iterate the whole queue when

the price updates but in worse cases can match orders in $O(n)$. The queue has been implemented as a double linked list so that inserting new orders can be achieved in $O(n)$ time.

8.1.3 Core Server Architecture

Because all communication with the Betfair NG API, and therefore the service layer occurs asynchronously it was decided that the core server implement an event based architecture.

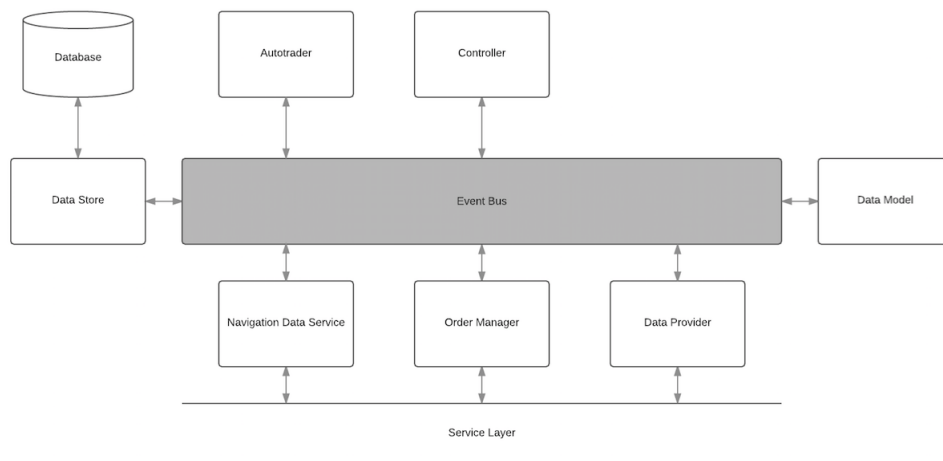


Figure 8.4: Core Server Architecture

As is shown in Figure 8.4 the Event Bus lies at the center of the architecture. Each individual component is subscribed to a channel on the Event Bus for instructions and broadcasts output on a different channel. If one of the components needs to send commands to another part of the system (i.e. the Order Manager may need to subscribe to market data updates for a specific market) it makes the request to the controller which will route the request to the correct part of the system and subscribe it to the correct channels. This is also the case for any external parts of the system that will communicate with the server (i.e. the GUI). All requests will be made to the Controller. This design means each part of the system only needs to know about the Event Bus and the Controller.

This design has several key benefits:

- Each part of the system has a low dependency on how the rest of the system is implemented.
- Each part can be built in isolation as it only relies on the event bus and the API defined by the controller.
- It will be easy to scale as each part of the server can run on independent machines in different locations.
- It will be easy to test as each part of the server can be isolated and assertions made corresponding to each input message.

- It will be easy to change the implementation of any part of the server without affecting the rest of the server as long as the format of the incoming and outgoing messages remain the same.

8.1.4 Central Configuration

What was not shown in Figure 8.4 is that each component within the server, including the service layer, has access to a predefined configuration file that is loaded when the server is initialised. This file contains, amongst other things, the names of all the channels on the Event Bus and the current session token for connectivity with the Betfair NG API. This file provides a centralised, easy to maintain space to store and track settings.

8.1.5 The Event Bus

The Event Bus was implemented by extending the existing Akka Event Bus. Rabbit MQ[30] was initially considered as a solution but the decision was made to use the Akka Event Bus because all of the components used in the system that communicate with the Event Bus were written in Scala and Akka and it seemed unnecessary to complicate the system by using an extra piece of technology. If in the future this changed it would be easy to swap out the existing Event Bus for one implemented in Rabbit MQ or a similar technology as the Event Bus only exposes two functions; publish and subscribe.

The Akka Event Bus was extended to include sub-channels. These sub-channels were implemented in the similar way to a directory structure in Linux.

For example:

Market data updates would be broadcast on the channel "MarketUpdates/EventA/MarketB" with "EventA" being the event Id and "MarketB" the market Id of the corresponding market. If a subscriber only wanted to receive updates for "MarketA" they could subscribe to "MarketUpdates/EventA/MarketA", if they wanted to subscribe to "MarketA" but didn't know the event Id of the market they could subscribe to "MarketUpdates/*/MarketA". If a subscriber wanted to receive market data updates for all markets they could subscribe to "MarketUpdates/" and if they wanted to receive updates for all markets in "EventA" they could subscribe to "MarketUpdates/EventA/".

This system greatly reduces the amount of unwanted messages sent within the Core Server.

8.1.6 Data Provider

The Data Provider is responsible for making market data requests to the service layer of the server. The two main calls to the Betfair NG API that the Data Provider calls are:

ListMarketBook

Returns a list of dynamic data about markets. Dynamic data includes prices, the status of the market, the status of selections, the traded volume, and the status of any orders you have placed in the market[33].

This call is made to retrieve the current state of a given market. As described earlier in this report a market can have multiple selections. This call will return the current exchange prices, orders and matches for each of these selections amongst other data. To make sure the system always has up to date data for a given market multiple calls need to be made every second.

There are a few issues that needed to be considered when designing this part of the system

1. The Betfair NG API states that "Calls to listMarketBook should be made up to a maximum of 5 times per second to a single marketId." [33]. If this limit is exceeded then it can result in either disconnection from the exchange or incur costs, neither is desirable.
2. When requesting data for multiple markets it is advised to reduce the number of calls by requesting data for multiple markets in one call. In the case where market data is required for 20 different markets, instead of making independent HTTP calls for each market one call can be made for all the markets. There are however limits on the amount of data that can be made in each call.
3. Market data calls can be made for varying levels of data as shown in Figure 8.5, each of these options has an associated weight as shown in Figure 8.6 and each call to ListMarketBook must not exceed a maximum weighting of 200. The price projection is set for the call and not for each individual market within the call.

PriceData	
Value	Description
SP_AVAILABLE	Amount available for the BSP auction.
SP_TRADED	Amount traded in the BSP auction.
EX_BEST_OFFERS	Only the best prices available for each runner, to requested price depth.
EX_ALL_OFFERS	EX_ALL_OFFERS trumps EX_BEST_OFFERS if both settings are present
EX_TRADED	Amount traded on the exchange.

Figure 8.5: ListMarketBook Price Projections

listMarketBook

PriceProjection	Weight
Null (No PriceProjection set)	2
SP_AVAILABLE	3
SP_TRADED	7
EX_BEST_OFFERS	5
EX_ALL_OFFERS	17
EX_TRADED	17

Figure 8.6: ListMarketBook Market Data Limits

As such calls to ListMarketBook need to be grouped by price projection. The decision was made to only support the combinations of price projections shown in Table 8.1, here you can see the maximum number of markets/call that can be made with each corresponding combination of price projections

Price Projection	Weight	Number of Markets/Call
EX_BEST_OFFERS	5	200 / 4 = 40
EX_ALL_OFFERS	17	200 / 17 = 11
EX_BEST_OFFERS & EX_TRADED	22	200 / 22 = 9
EX_ALL_OFFERS & EX_TRADED	34	200 / 34 = 5

Table 8.1: Number of markets/call by price projection

As the weight of the price projection increases so does the amount of data returned and as such they have the following hierarchy:

- EX_BEST_OFFERS requests market data for each selection to 3 prices in depth.
 - EX_ALL_OFFERS request market data for each selection for full market depth and is inclusive of EX_BEST_OFFERS
 - EX_BEST_OFFERS & EX_TRADED requests market data for each selection to 3 prices in depth and includes traded volume, this is inclusive of EX_BEST_OFFERS.
 - EX_ALL_OFFERS & EX_TRADED requests market data for full market depth and includes traded volume, this is inclusive of EX_ALL_OFFERS, EX_ALL_OFFERS and EX_BEST_OFFERS & EX_TRADED.
4. If the Data Provider has multiple requests for the same market's data but at different price projections it needs to be able to request the correct price projection from the service layer. If one of these requesters stops subscribing to data for the given market the Data Provider needs to be able to adjust its call accordingly.

To ensure that the Data Provider optimises the ListMarketBook calls whilst addressing all the issues above the following design was implemented:

- When a requester subscribes, via the Controller, to market data for a given market the Controller subscribes them to the corresponding channel on the Event Bus and sends a message to the Data Provider informing it of their subscription.
- The Data Provider maintains a set of all the subscribers for each market and the price projection's they have requested.
- At a configurable interval of no more than 250 milliseconds, to ensure calls are made less than 5 times a second, the Data Provider produces a list of all the required markets with the lowest weighted price projection necessary to accommodate each markets subscribers.
- This list is then grouped by price projection and split into the maximum sized chunks permissible for each projection.
- These chunks are then sent to an Akka Round Robin router[20] that routes the requests to a pool of actors (the number of actors in this pool is another configurable option).

- These actors send the requests concurrently to the service layer and when the data is received broadcast it on the corresponding market data channel for each market on the Event Bus.

When a subscriber no longer requires data for a given market a message is sent to the Data Provider telling it to remove the subscription from its records. This procedure is repeated as long as the Data Provider has subscribers and is depicted in Figure 8.7.

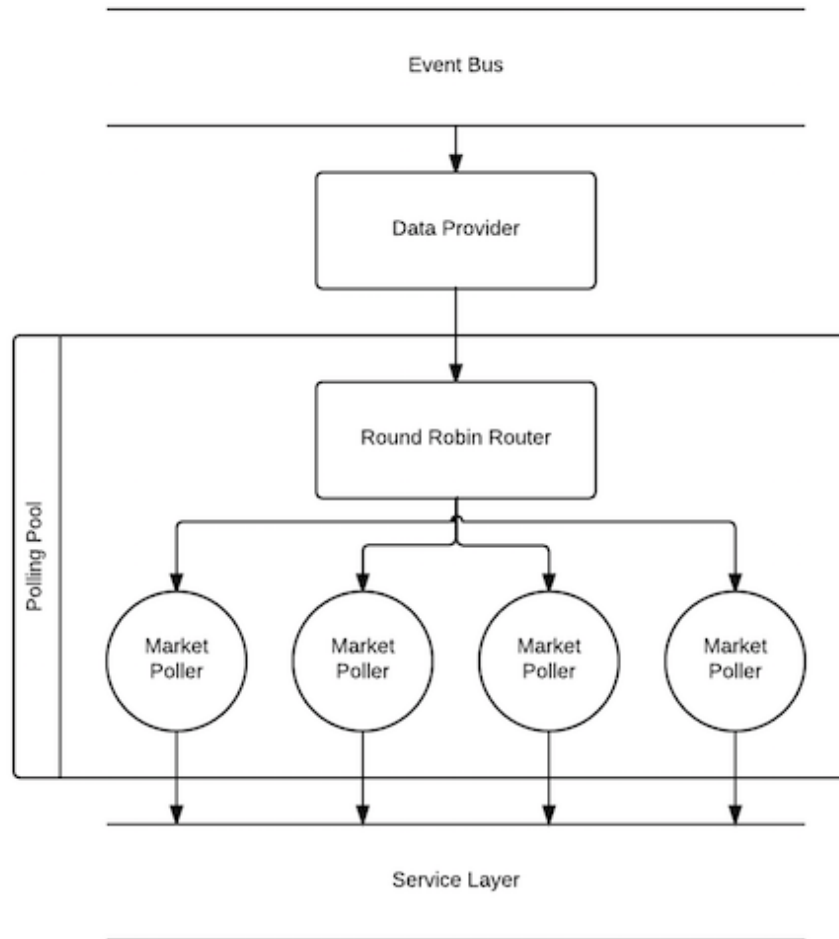


Figure 8.7: Data Provider Polling Mechanism

ListMarketCatalogue

Returns a list of information about published (ACTIVE/SUSPENDED) markets that does not change (or changes very rarely). You use ListMarketCatalogue to retrieve the name of the market, the names of selections and other information about markets[34].

This call is made less frequently compared to the ListMarketBook call and when the Data Provider receives this request it calls the service layer immediately and sends the results directly to the requester.

8.1.7 Data Model

The Data Model listens to the channel of market data output from the Data Provider and keeps a record of the last market book data sent by the Data Provider for each market. Each time it receives a new update for a given market it checks if it is different from the copy it already has. If it is different it updates its local copy and broadcasts the new copy on its output channel. These market data updates are the ones that subscribers to the system listen to, not the ones output by the Data Provider. This way subscribers to market data are not spammed with continuous updates for markets that haven't changed.

The Data Model will also, on request, send out a copy of the currently held market data for a given market directly to the requester. This service is essential because a situation can easily arise where a requester subscribes to market data updates for a given market but the market data does not change frequently, in this case the subscriber would have to wait an arbitrary amount of time for its initial view of the market data.

The Data Model, in conjunction with the Data Provider, essentially turn Betfair's NG API, which must be polled for market updates, into one which can be subscribed to and will push new updates to a subscriber when deltas in the market occur.

8.1.8 Order Manager

Order manager updates its list of tracked orders from the service layer on start up and reconciles this list periodically

The Order Manager is responsible for making requests to Place, Cancel, Update and Replace orders with the service layer. The Order Manager is also responsible for tracking, and broadcasting updates pertaining to, the status of orders once they have been placed with the service layer and keeps a record of the volume of back and lay orders matched for each market and broadcasts updates when these matches occur.

On start up the Order Manager requests a list of all the currently open orders at the Betfair exchange and creates the initial list of tracked orders from this list. At intervals of a configurable length the Order Manager repeats this process reconciling its tracked orders with the list received from the exchange.

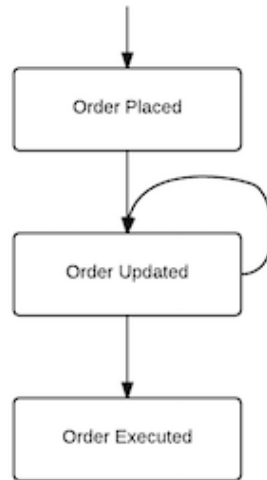


Figure 8.8: Life Cycle of an Order

The life cycle of an order is depicted in Figure 8.8. Once an order has been successfully placed with the service layer the Order Manager adds it to its list of tracked orders and tells the Controller to subscribe it to market data updates for the given market. As the Order Manager receives these updates it will compare the version of the orders in the update to those it is currently tracking. If there is a difference the Order Manager will broadcast the new status of the order. Once the order has been executed the Order Manager will remove it from its list of tracked orders and, if there are no more tracked orders for that market, tell the Controller to un-subscribe it from further market updates for the given market.

Market data updates also include the total volume of matched back and lay orders for each selection and the average price they were matched at. The Order Manager keeps a list of these matches and when the matches in the market data update differ from those it holds the Order Manager broadcasts a message alerting subscribers to the update.

order manager should send out system alerts when the maximum number of orders placed/hour exceeds those imposed by the Betfair NG API

8.1.9 Navigation Data Service

The navigation data service was implemented as a response to issues faced while writing the Client. For the Client to be able to navigate all the events and markets available on the Betfair exchange it would need to make requests for all the event and market catalogues available on the exchange. It would prove very computationally expensive for the server to request all this data and maintain an up-to-date copy of the hierarchy.

To solve this problem Betfair provides Navigation Data for Applications in the form of a compressed Json file which can be downloaded from a separate HTTP endpoint. The API states that "The file data is cached and new request for the file one an hour should be suitable for those looking to accurately recreate the Betfair navigation menu." [35]

The Navigation Data Service is responsible for downloading this file on start up, caching a copy of

the file locally and updating that cached copy on an hourly basis.

The Navigation Data Service will, on request, will send the requester the navigation data for a given event type (i.e. Football or Horse Racing). The reason the whole file isn't sent is that it exceeds 3 Megabytes in size and would have meant increasing the Akka framesize and the send and receive buffer size. The communication of this file alone was not deemed as a good enough reason to increase these settings.

8.1.10 Data Store

To be added before final draft

MongoDB[22] is used by the Data Store for persistence and the Reactive Mongo[9] driver for communication between Scala and MongoDB

8.1.11 Auto Trader

As stated earlier the interim report for this project including objectives to

- Define a language so a user can describe a trading strategy.
- Write a parser or compiler to convert a trading strategy written in the above language into steps the trading system can understand.
- Create trading strategies of differing complexities and test and report on their performance .

These objectives were sidelined so a larger emphasis could be placed on the machine learning section of the project. This decision was made in part because Akka provides a very concise way of programming a finite state machine and because there was not enough time to implement all the desired functionality in the project.

After reading the documentation for the Akka FSM[akkaFSM] it was obvious that it provided the perfect abstraction required to implement a trading strategy similar to that described in the requirements analysis. Additionally the Akka testkit provided some very good tools to test the strategy once written[21]. Using the Akka testkit it was possible to pass a strategy a mock of the servers Controller, and assert the state transitions and internal data changes the strategy would make in each of its states given all possible received events.

The Auto trader is comprised of three parts:

The Auto Trader

The main Auto Trader Actor listens to the Event Bus for three types of messages:

- Start Strategy which starts a strategy on a given market with a given set of settings
- Stop Strategy which stops a strategy with a given id running on a given market
- List Running Strategies which sends the requester a list of all the currently running strategies and which states they are in.

When the Auto Trader receives a message to start a new instance of a trading strategy on a market it creates a new FSM actor for the strategy and creates another Actor to monitor and broadcast updates on the status of the strategy as it transitions from state to state. This Actor is described below. The Auto Trader maintains a list of all running strategies and their corresponding monitors, removing entries when the strategies are finished or are stopped.

The Monitor

A new monitor Actor is created for every instance of a strategy running. The reason for designing the system this way is so that responsibility for reporting can be kept separate from them trading strategy to keep the logic as simple as possible and make it easier to test. Keeping the monitor separate also means it can be reused over different types of strategy.

The monitor broadcasts messages on the Event Bus to subscribers when the Auto Trader is started, when it changes states and when it finishes. The monitor is also responsible for clean shutting down of the strategy once it has finished.

The Strategy

To test the above infrastructure works the basic strategy trading strategy described in the appendix was built using an Akka FSM. The strategy has access to the Core Server's Controller and essentially acts as a headless client, requesting order and market updates and placing and cancelling orders with the Core Server. Later in the report a more detailed strategy will be implemented using the machine learning model trained in the second section of this project

add basic trading strategy to the appendix

8.1.12 Controller

As stated earlier all requests to the server are placed with the Controller. The Controller has three main responsibilities:

1. To subscribe and un-subscribe actors to and from the correct channels on the Event Bus. The Controller supports the following types of subscription:
 - Market Data updates
 - Order updates
 - Auto Trader updates
 - System Alerts

When an actor requests subscription for a specific market's data updates a message is also sent to the Data Provider on their behalf informing it of their subscription.

2. To watch subscribers for termination and in the event of termination un-subscribe them from all channels on the Event Bus and tell the Data Provider to cancel any subscriptions that were made on their behalf for market data.
3. To route commands to the correct parts of the core server. The commands handled by the Controller are documented in the Appendix

add Controller API documentation to Appendix

8.2 Web Server and Client (GUI)

As stated earlier in this report the Client will be designed to run in the browser and will be written in CoffeeScript using AngularJs. CoffeeScript compiles to JavaScript and allows the programmer to achieve more with less code[7]. AngularJs is very well documented and supported by Google and there is a large on-line community providing the answers to most questions, additionally I have already written several medium scale web applications in AngularJs.

Because the GUI and the Client are not the main focus of this project, but are required to provide interaction between the server and the user, the majority of decisions have been made to increase the speed of development. Initially ScalaFx[13] and ScalaJs[25] were considered as alternatives but neither are well documented and the author has little experience programming with either which would have increased development time.

8.2.1 Web Server

It was decided that the Web Server be implemented using Play Framework[11]. The project already uses Play's ScalaJson library on the server and the framework is built on Scala and Akka. This combined with the ability to compile CoffeeScript in real time as its being developed and handle non-blocking IO over either a RESTful API or a web socket made it an easy decision to make.

The Web Server communicates with the Core Server's Controller using Akka remoting[19]. When the Client loads the web page from the Web Server one of the first things it does is request a web-socket for communication with the Web Server. The Web Server then creates an Actor to handle communication with the web socket and passes the Actor a reference to the core server's Controller.

The Web Socket Actor has two responsibilities:

1. Convert JSON-RPC requests sent from the the browser, over the web-socket, into Scala case classes and forward them to the Core Server's Controller.
2. Convert Scala case classes sent from the core server into JSON-RPC responses and forward them over the web-socket to the browser.

The conversions are implemented with very little code using Play's ScalaJson library and because the web-socket is handled by an Akka Actor it can be subscribed by the Core Server's Controller to it's Event Bus. Because communication is conducted over a web socket the Web Server can push data to the client as required.

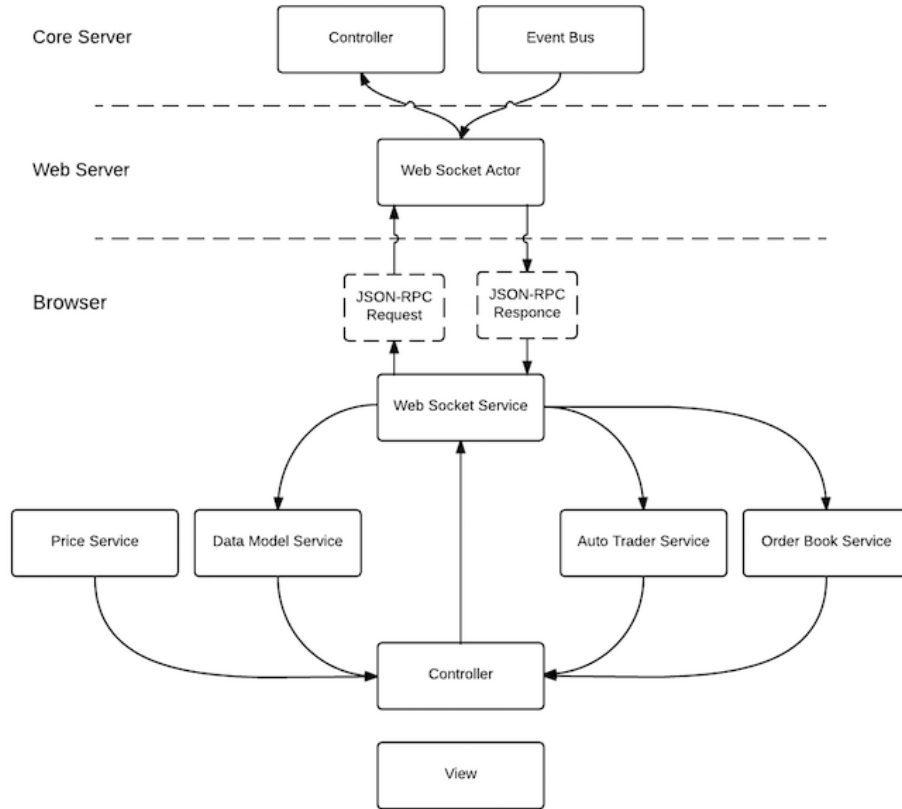


Figure 8.9: Web Server and Client Architecture

8.2.2 Client (GUI)

As stated earlier the client is implemented using AngularJs, the Web Server and Client Architecture is depicted in Figure 8.9. An angular service called the Web Socket Service controls all communication with the Web Server. Data within the browser is persisted in three other AngularJs services, AngularJs services are singletons and as such provide the perfect abstracting to share data across the GUI:

- The Data Model Service holds all market price data.
- The Order Book Service holds all the orders and matches.
- The Auto Trader Service holds all information about running strategies.

Additionally the price service contains all logic for calculating Betfair price increments. Betfair increments differ depending on the current price as shown in Figure 8.10.

Price	Increment
1.01 → 2	0.01
2 → 3	0.02
3 → 4	0.05
4 → 6	0.1
6 → 10	0.2
10 → 20	0.5
20 → 30	1
30 → 50	2
50 → 100	5
100 → 1000	10

Figure 8.10: Betfair Price Increments

The rest of the GUI code is implemented using AngularJs Controllers and Directives and HTML views. The amount of logic in the GUI has been kept to a minimum for the following reasons:

- JavaScript is not very good at handling numbers, all calculations are handled using floating points and this can lead to inaccuracies[44].
- It reduces the amount of logic that needs to be duplicated over the Client and the Server. This means if the logic in the Server changes the GUI need not change as its main responsibilities are to display the data received from the Server and communicate input from the user back to the Server.

Visual Design of the GUI

This project will not go into great depth regarding the graphical design of the GUI as it is clearly not the focus. Because of this and in an effort to increase development speed the styling has been loosely based on Betfair's website. There are two main pages the user can view in the browser:

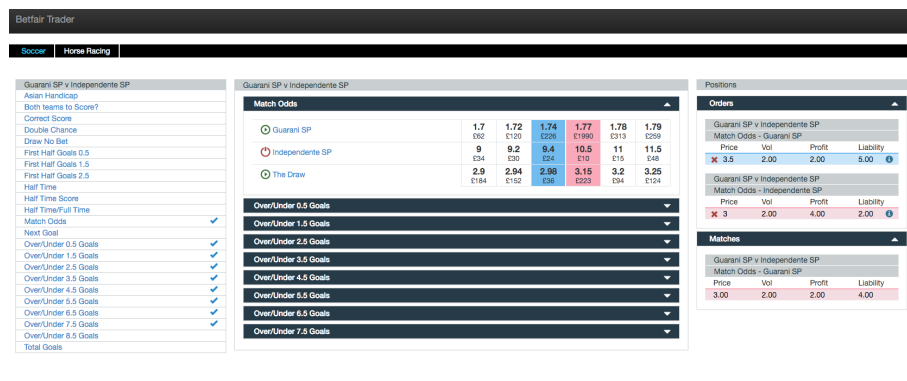


Figure 8.11: GUI Main Window

The GUI Main Window shown in Figure 8.11 is used to navigate the hierarchy of Betfair events and markets. This window is split into three sections:

- The Navigation Window (Left) which allows the user to navigate events and markets.
- The Detail Window (Center) which shows high level price data for events and markets.
- The Position Window (Right) which shows the user's orders and matches grouped by market and selection.

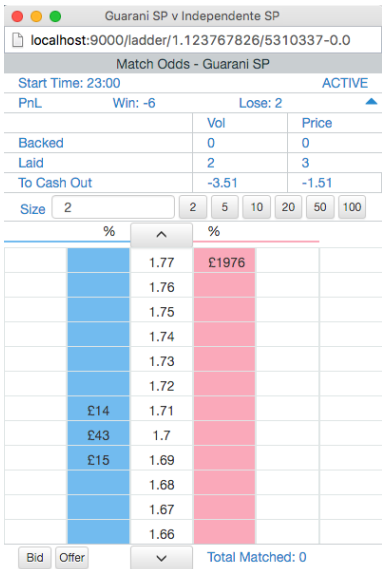


Figure 8.12: GUI Ladder Window

The GUI Ladder Window shown in Figure 8.12 allows the user to navigate the full price depth of a market, shows the user information about their PnL and position in a market and allows the user to place or cancel orders in a market

add section on data store windows once they are implemented, these will implemented using HighCharts[6]

Predicting Price Movement

9.1 Technical Analysis

There are two main schools of thought in financial markets, technical analysis and fundamental analysis. Technical analysis concentrates on the study of market action by looking at the price previous price movement of a security, fundamental analysis focuses on the economic forces of supply and demand that cause prices to move higher, lower, or stay the same[28]. Both approaches have the same desired outcome: to predict the direction of future price movement. Technical Analysis assumes that the economic forces of supply and demand are already implied by the market price.

Previous studies have shown that quantitative technical analysis can be effective when used by an automated agent to predict market direction on financial markets[40]. In Technical analysis of the financial markets: A comprehensive guide to trading methods and applications[28] John Murphy states that "One of the great strengths of technical analysis is its adaptability to virtually any trading medium and time dimension". The traditional gamblers approach is to look at the fundamentals surrounding a sporting event, i.e. the previous form of a horse or who the jockey is. This section of the report will attempt to use machine learning and quantitative technical analysis techniques to train a model from historical Betfair price data to predict market trends in real time.

9.2 Historical Market Data

The decision was made to focus this part of the project on horse racing data specifically the win markets. Betfair horse racing markets have some of the highest liquidity levels on the exchange, which helps to stabilise price movement. Markets with lower liquidity levels can exhibit erratic price movement because orders can have a larger effect on the price as they strip out the small levels of volume. Horse racing markets are especially liquid just before the event and there is a large community of people trading these pre-race markets on Betfair.[3][16]

Betfair and their third part data provider Fracsoft kindly provided me with one months historical tick data for horse racing markets from the 11th October 2014 to the 10th November 2014. This data was generated by polling the Betfair exchange four times a second, producing a record every time there was a delta in the market data.

9.3 Preprocessing the Data

The months data comprised of 97,422,530 records spread over 848 markets (races), comprising of 8358 selections (runners). The data was read into a MongoDB database, splitting the data into collections, one for each market.

To make the price data easier to analyse the data for each market would be grouped into a series of intervals, each interval representing a period of time of the same length. Each interval will have the high, low and closing price and the volume traded for that period. The aim, once a set of intervals had been created for each market was to apply technical analysis indicators to the intervals. These indicators would constitute the features used by a machine learning model to classify if the next interval in the series was going to close above (up), below (down), or at the same level (none) as the current intervals closing price.

Two sets of intervals were built for each market, one set comprising of 1 minute intervals and the other of 10 second intervals, with a view to compared a models accuracy over both time series.

Finally each interval was labelled as "UP", "DOWN" or "NONE" according to the delta in closing price compared to the closing price of the next interval in the series.

9.3.1 Betfair Price Data

Decimal Odds	Implied Probability (%)	
	Win	Lose
1.01	99	1
1.50	66.6	33.3
2.00	50	50
3.00	33.3	66.6
100.00	1	99

Table 9.1: Implied Probability of Decimal Odds

Betfair prices are represented as decimal odds. Table 9.1 shows the implied probabilities of a series of decimal odds and you can see that the price range between an implied probability of winning of 99% and 50% is 0.99 and the price range between 50% and 1% is 98.

further analysis needs to be carried out to see if converting the prices into applied probabilities has any effect on the classification result, there may be a bug in here !

9.3.2 Technical Analysis Indicators

Once the intervals had been produced and the prices normalised the technical indicators documented in the appendix were applied to the intervals.

record indicators in the appendix

All the indicators used in this project are widely used in financial trading and they were selected because their output values lie within a predefined range and the output can therefore be compared across markets. One major difference between financial markets and horse racing markets is that a horse racing market has a much shorter life span than that of a financial market. Models trained on financial markets are usually trained using data from the same market they will be used to predict. This will not be possible with the horse racing markets as there isn't enough data. The model in this project will be trained using the labelled intervals produced over all the markets in the historical data provided. It is therefore import that the indicators used can be compared across markets.

Each indicator was written in Scala and their implementations tested by downloading example data from the Chart School website[39] and creating unit tests that asserted given the input in the example data the implementation of each indicator produced the same output as that in the example data.

On its own each of these indicators is unlikely to accurately predict or confirm price movement in all the markets, all the time, but it is hoped by using a combination of indicators the model will be able to see past any noise in the data.

9.4 Choosing a Model

The decision was made to use a Random Forest to classify the data. A Random Forest is made up of an ensemble of decision trees and was initially proposed in the 2001 paper Random Forests by Leo Breiman[5]. Decision trees are expressive and easy to understand and implement due to their recursive nature[10] and this greatly influenced the decision.

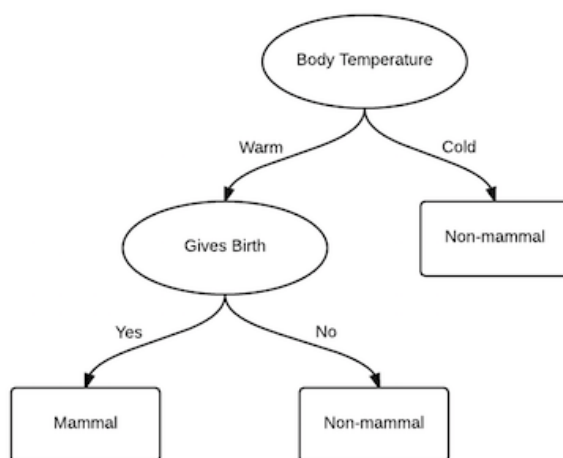


Figure 9.1: Decision Tree for Mammal Classification

, this is commonly decided using entropy[45] but other measures can be used.

Decision Trees work by splitting the data, one feature at a time, using the feature that produces the best split of the training data at that given node until the data is of a single classification or a set size. The quality of the split is commonly decided by maximising the entropy gain[45] but other measures can be used. Figure 9.1 shows an example of a simple decision tree for classifying mammals. Decision Trees are however prone to over-fitting as the model can too closely learn the training data and as such the model usually displays low bias and high variance. One solution to this problem is to prune the decision tree which is the approach that the C4.5 Decision tree algorithm takes[37].

9.4.1 Random Forest

The Random Forest model proposed by Breiman solves the problem of over-fitting by growing an ensemble of trees and letting them vote for the most popular class. Each tree is grown using a subset of the training data the same size as the training data but selected at random from the initial dataset using replacement, these subsets are known as bootstrap samples. As each tree is grown the algorithm only uses a random subset of features at each split to split on, this is known as subspace sampling. Introducing this randomness into the trees encourages diversity within the forest and reduces the tendency to over fit the data, increasing the bias and decreasing the variance. Trees are full grown without pruning until, all the instances in a node are of the same class or all the features in a node are of the same value or the number of instances in a node is less than or equal to a predefined number. This results in three options for the forest that can be optimised during the training process: the number of trees in the forest, the number of random features to choose from at each split and the number maximum number of instances in a leaf node.

9.5 Implementation

It was initially considered that the model be implemented using a tool-kit. However the only toolkit found that included a Random Forest was Apache Sparks MLlib[41], using this tool-kit required also using Apache Spark and submitting jobs to a Spark cluster. At this stage of the project introducing Spark to the list of technologies used would require a lot of time and greatly increase complexity. As such the decision was made to implement the Random Forest from scratch in Scala.

include table of training & testing data set sizes here

Once the data had been processed and labelled and the Random Forest proposed in Breiman's paper was implemented the data was split into two sections. Data from 11th October 2014 to 31st October 2014 was to be used for training and data from 1st November 2014 to 10th November 2014 was to be used for testing.

Unfortunately training the model took a very long time. The majority of the time was spent selecting the splits as the trees were grown because all of the features in the data were continuous, rather than belonging to a finite number of categories. For a dataset of n instances $n - 1$ splits would have to be considered for each feature. If three randomly picked features were considered for each split this would mean $3 * (n - 1)$ splits being compared.

One solution might have been to profile the JVM whilst the model was being trained to see if it could be optimised to run faster but instead a revised version of the algorithm was used.

9.5.1 Extremely Random Forest

Extremely Random Forests differ from Random Forests in two major ways and have, in some instances, been shown to produce better results[14]:

1. Each tree is trained using the whole of the training data, not a sub-sample.
2. As the trees are grown the splits are picked at random. If there were 3 random features being considered at each split a random value would be selected for each feature within the range

that feature had in the dataset being split. The resulting entropy of splitting on the 3 features would then be compared using these random values for the split and the one that produced the best split chosen.

This algorithm removes the randomness introduced in Breiman's algorithm by sub-sampling the data set used for each tree but then introduces randomness by picking the value to split on at random. Selecting a value at random is computationally faster than comparing all the possible splits and will scale better as a random number can be picked in constant time and as such will not be effected as the size of the data increases.

The algorithm to grow the trees used in the Extremely Random Forest is show in 9.2 and has been taken from Extremely randomized trees: Pierre Geurts, Damien Ernst, Louis Wehenkel[14]

```

Split_a_node(S)
Input: the local learning subset  $S$  corresponding to the node we want to split
Output: a split  $[a < a_c]$  or nothing
– If Stop_split(S) is TRUE then return nothing.
– Otherwise select  $K$  attributes  $\{a_1, \dots, a_K\}$  among all non constant (in  $S$ ) candidate attributes;
– Draw  $K$  splits  $\{s_1, \dots, s_K\}$ , where  $s_i = \text{Pick\_a\_random\_split}(S, a_i), \forall i = 1, \dots, K$ ;
– Return a split  $s_*$  such that  $\text{Score}(s_*, S) = \max_{i=1, \dots, K} \text{Score}(s_i, S)$ .

Pick_a_random_split(S, a)
Inputs: a subset  $S$  and an attribute  $a$ 
Output: a split
– Let  $a_{\max}^S$  and  $a_{\min}^S$  denote the maximal and minimal value of  $a$  in  $S$ ;
– Draw a random cut-point  $a_c$  uniformly in  $[a_{\min}^S, a_{\max}^S]$ ;
– Return the split  $[a < a_c]$ .

Stop_split(S)
Input: a subset  $S$ 
Output: a boolean
– If  $|S| < n_{\min}$ , then return TRUE;
– If all attributes are constant in  $S$ , then return TRUE;
– If the output is constant in  $S$ , then return TRUE;
– Otherwise, return FALSE.

```

Figure 9.2: Algorithm for growing a Decision Tree for an Extremely Random Forest

9.6 Optimisation

Once implemented the forest was trained and the number of trees, number of features to split on and leaf size optimised using cross-validation on the training set. The result can be seen below

Add optimisation results once analysis on converting the price data has been completed

9.7 Results

Once the optimised the following results were achieved training the classifier with the settings selected during optimisation.

I will add the results before the final draft, I was thinking I could test for bugs in the classifier by loading the data into python and testing it using the sklearn toolkit. If the results are very different I will know there is a bug in my software

9.8 Integration With The Trading System

The trained model will be serialised to mongoDB and made available to use by the auto trader

this section to be added before the final draft

Appendices

Bibliography

- [1] Akka. *Akka Actors*. <http://doc.akka.io/docs/akka/2.4.2/general/actors.html>. URL: 2016 (visited on 03/31/2016).
- [2] Betfair. *A Betting Revolution*. 2015. URL: <http://corporate.betfair.com/about-us/a-betting-revolution.aspx> (visited on 03/31/2016).
- [3] Betfair. *Cold Trading On Betfair*. 2016. URL: <https://betting.betfair.com/cold-trading-on-betfair.html> (visited on 03/31/2016).
- [4] Jet Brains. *IntelliJ IDEA*. 2016. URL: <https://www.jetbrains.com/idea/> (visited on 03/31/2016).
- [5] Leo Breiman. “Random forests”. In: *Machine learning* 45.1 (2001), pp. 5–32.
- [6] High Charts. *High Charts*. 2016. URL: <http://www.highcharts.com/> (visited on 03/31/2016).
- [7] CoffeeScript.org. *CoffeeScript overview*. 2016. URL: <http://coffeescript.org/#overview> (visited on 03/31/2016).
- [8] Betfair community. *API NG Migration update*. 2014. URL: <http://community.betfair.com/service/go/thread/view/94166/30402691/api-ng-migration-update> (visited on 03/31/2016).
- [9] Stephane Godbillon & contributors. *Reactive Mongo*. 2016. URL: <http://reactivemongo.org/> (visited on 03/31/2016).
- [10] Peter Flach. *Machine Learning: The Art and Science of Algorithms That Make Sense of Data*. New York, NY, USA: Cambridge University Press, 2012. ISBN: 1107422221, 9781107422223.
- [11] Play Framework. *Play Framework*. 2015. URL: <https://www.playframework.com/> (visited on 03/31/2016).
- [12] Play framework. *ScalaJSON*. 2016. URL: <https://www.playframework.com/documentation/2.5.x/ScalaJson> (visited on 03/31/2016).
- [13] Scala Fx. *ScalaFx*. 2016. URL: <http://www.scalafx.org/> (visited on 03/31/2016).
- [14] Pierre Geurts, Damien Ernst, and Louis Wehenkel. “Extremely randomized trees”. In: *Machine learning* 63.1 (2006), pp. 3–42.
- [15] GIT. *GIT fast version control*. 2016. URL: <https://git-scm.com> (visited on 03/31/2016).
- [16] Google. *Betfiar Horse Racing pre race trading Google search*. 2016. URL: <https://www.google.co.uk/webhp?sourceid=chrome-instant&ion=1&espv=2&ie=UTF-8#q=betfair+horse+racing+pre+race+trading> (visited on 03/31/2016).
- [17] Haskell.org. *The JavaScript Problem*. 2015. URL: https://wiki.haskell.org/The_JavaScript_Problem (visited on 03/31/2016).
- [18] Lightbend Inc. *Akka*. 2016. URL: <http://akka.io/> (visited on 03/31/2016).
- [19] Lightbend Inc. *Akka Remoting*. 2016. URL: <http://doc.akka.io/docs/akka/snapshot/scala/remoting.html> (visited on 03/31/2016).

- [20] Lightbend Inc. *Akka Round Robin Router*. 2016. URL: http://doc.akka.io/docs/akka/snapshot/scala/routing.html#RoundRobinPool_and_RoundRobinGroup (visited on 03/31/2016).
- [21] Lightbend Inc. *Testing and Debugging Finite State Machines*. 2016. URL: <http://doc.akka.io/docs/akka/2.4.2/scala/fsm.html#testing-and-debugging-finite-state-machines> (visited on 03/31/2016).
- [22] MongoDB Inc. *MongoDB*. 2016. URL: <https://www.mongodb.com/> (visited on 03/31/2016).
- [23] Typesafe Inc. *Spray*. 2016. URL: <http://spray.io/> (visited on 03/31/2016).
- [24] The Chartered Institute for IT. *BCS Code of Conduct*. 2016. URL: <http://www.bcs.org/category/6030> (visited on 03/29/2016).
- [25] Scala Js. *ScalaJs*. 2016. URL: <https://www.scala-js.org/> (visited on 03/31/2016).
- [26] James F Kurose and Keith W Ross. *Computer networking: a top-down approach*. Addison-Wesley, 2007.
- [27] Ecole Polytechnique Federale de Lausanne (EPFL). *Scala*. 2016. URL: <http://www.scala-lang.org/> (visited on 03/31/2016).
- [28] John J Murphy. *Technical analysis of the financial markets: A comprehensive guide to trading methods and applications*. Penguin, 1999.
- [29] Stack Overflow. *how-does-akka-compare-to-erlang*. 2010. URL: <http://stackoverflow.com/questions/4491039/how-does-akka-compare-to-erlang> (visited on 03/31/2016).
- [30] Pivotal. *Rabbit MQ*. 2016. URL: <https://www.rabbitmq.com/> (visited on 03/31/2016).
- [31] Betfair Developer Program. *Application Keys*. 2016. URL: <https://api.developer.betfair.com/services/webapps/docs/display/1smk3cen4v3lu3yomq5qye0ni/Application+Keys> (visited on 03/31/2016).
- [32] Betfair Developer Program. *Betfair Service NG*. 2016. URL: <https://github.com/city81/betfair-service-ng> (visited on 03/31/2016).
- [33] Betfair Developer Program. *List Market Book*. 2016. URL: <https://api.developer.betfair.com/services/webapps/docs/display/1smk3cen4v3lu3yomq5qye0ni/listMarketBook> (visited on 03/31/2016).
- [34] Betfair Developer Program. *List Market Catalogue*. 2016. URL: <https://api.developer.betfair.com/services/webapps/docs/display/1smk3cen4v3lu3yomq5qye0ni/listMarketCatalogue> (visited on 03/31/2016).
- [35] Betfair Developer Program. *Navigation Data For Applications*. 2015. URL: <https://api.developer.betfair.com/services/webapps/docs/display/1smk3cen4v3lu3yomq5qye0ni/Navigation+Data+For+Applications> (visited on 03/31/2016).
- [36] Betfair Developer Program. *Sample Code*. 2016. URL: <https://api.developer.betfair.com/services/webapps/docs/display/1smk3cen4v3lu3yomq5qye0ni/Sample+Code> (visited on 03/31/2016).
- [37] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.
- [38] Honest Betting Reviews. *Betting Exchanges Compared*. 2016. URL: <http://www.honestbettingreviews.com/betting-exchanges/> (visited on 03/31/2016).
- [39] Chart School. *Technical Indicators and Overlays*. 2016. URL: http://stockcharts.com/school/doku.php?id=chart_school:technical_indicators (visited on 03/31/2016).

- [40] Cyril Schoreels, Brian Logan, and Jonathan M Garibaldi. “Agent based genetic algorithm employing financial technical analysis for making trading decisions using historical equity market data”. In: *Intelligent Agent Technology, 2004.(IAT 2004). Proceedings. IEEE/WIC/ACM International Conference on*. IEEE. 2004, pp. 421–424.
- [41] Apache Spark. *Mllib ensembles*. 2016. URL: <http://spark.apache.org/docs/latest/mllib-ensembles.html> (visited on 03/31/2016).
- [42] London The Government Office for Science. *The Future of Computer Trading in Financial Markets (2012)*. 2012. URL: https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/289431/12-1086-future-of-computer-trading-in-financial-markets-report.pdf (visited on 03/31/2016).
- [43] Andrew Trenk. *Testing on the Toilet: Testing State vs. Testing Interactions*. 2013. URL: <http://googletesting.blogspot.co.uk/2013/03/testing-on-toilet-testing-state-vs.html> (visited on 03/31/2016).
- [44] Modern Web. *JavaScript floating point numbers*. 2016. URL: <http://modernweb.com/2014/02/17/what-every-javascript-developer-should-know-about-floating-points/> (visited on 03/31/2016).
- [45] Wolfram Math World. *Entropy*. 2016. URL: <http://mathworld.wolfram.com/Entropy.html> (visited on 03/31/2016).