IN-PLAY TENNIS MARKET MAKING USING TRANSFER LEARNING-BASED LSTM AND ENHANCED MARKOV MODELS

K. Chan, O. Ajomale

4th Year Project Final Report

Department of Electronic &
Electrical Engineering

UCL

Supervisor: Prof. Philip Treleaven
Advisor: John Goodacre

(Credit of this work goes to the research of John Goodcare)

4th May 2023

Table of Contents

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

I have read and understood UCL's and the Department's statements and guidelines concerning plagiarism.

I declare that all material described in this report is my own work except where explicitly and individually indicated in the text. This includes ideas described in the text, figures, and computer programs.

This report contains N pages (excluding this page and the appendices) and XX words.

Signed: K. Chan, O. Ajomale Date: 04/05/23

Acknowledgements

We express our sincere gratitude to John Goodacre for his invaluable guidance and support as our advisor. John's expertise and insightful contributions have influenced and shaped our work. We are indebted to him for his ideas and research that have formed a solid foundation for our project.

In-Play Tennis Market Making Using Transfer Learning-Based LSTM Models and Enhanced Markov Models

K. Chan, O. Ajomale

This report presents an Advanced tennis match prediction and market making model that combines the strength of a hierarchal Markov Chain model with a sophisticated ensemble model. The goal of this project is to develop a more accurate and effective method for predict the outcome of tennis match by considering the hierarchical structure of tennis and the individual player statistics and then use this model to develop an intelligent market maker algorithm that can profit off these enhanced predictions.

We developed a transfer learning-based LSTM model allowing us to forecast temporal match winning probabilities of tennis matches. Forecasting capabilities of multiple models, including multivariate and univariate models, were developed, and compared using advanced cross-validation methods. Initial findings indicate that more work may be necessary to exploit the full capabilities of LSTMs in in-play tennis betting and integrate them into market making strategies. However, a groundwork has been established for future exploration in the area.

Furthermore, we developed a Markov chain model captures the essence of tennis matches by dividing them into games, sets, and matches, and considers the current set and game scores to update the match state. Thus offers a solid foundation for our tennis model as it provides an understanding of the dynamics of a Tennis Match. However, the model has its limitations, so to enhance it, A serve win percentage prediction model based on the players historical performance features and the current score of the game is integrated into the Markov chain Model.

An ensemble model, comprised of Gradient Boosting, Random Forest, and Multi-Layer Perceptron algorithms, was developed using feature engineering, feature selection, and model optimisation techniques such as cross-validation and grid search to improve enhance our Markov chain model.

Results from testing different models and strategies in market making indicate that some strategies offer an accurate and comprehensive approach to predicting tennis odds in betting exchanges, thus leading to the market maker profiting on a simulated tennis market.

Future research could focus on exploring additional features, models, and optimisation techniques to further enhance the predictive capabilities of this integrated approach.

1 Introduction

In recent decades, sports betting and gambling has become one of largest entertainment industries in the developed world, with a global market size reaching USD $231 billion in 2022. With the advent of betting exchanges, more serious punters have turned towards putting their bets in such markets rather than with bookmakers who often put out unfavourable odds for bettors.

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

Betting exchanges offer extensive functionalities in their application programming interface (API) which allow bettors to use computer programs to assist, or even take over, their betting decisions. Tennis is a popular sport for algorithmic prediction and betting due to its systematic scoring structure. Stochastic modelling and statistical analysis of tennis matches is an extensive field of study for the purpose of prediction, betting, or media presentation.

Our aim in this project is to devise and implement an advanced Markov chain-based model to predict the probabilistic outcomes of tennis matches. We integrate this model into a score inference algorithm in a multivariate LSTM model capable of multi-step time-series forecasting of betting market information to be used by an intelligent market maker.

2 Review of Literature and Previous Work

The literature review broadly focuses on modelling in-play tennis and market-making in the context of sports betting. There is a range of specific topics worth exploring, including trading and betting biases, market microstructure dynamics, stake size optimisation, game structure modelling of tennis, and machine learning methods for tennis modelling.

2.1 On Tennis Modelling and Psychological Considerations

There is a substantial amount of literature on the application of Markov models in tennis. Markov chains have been used to model tennis matches due to the highly structural nature of the tennis scoring system. Markov models have been applied and explored by Huang (2011) as a tool for score inferencing [1], and by Easton and Uylangco (2010) to explore the market efficiency of betting exchanges [2].

Huang (2011) showed the capabilities of Markov chain models in inferring in-play tennis scores using an algorithm which compares the Markov modelled match probabilities given by subsequent states (next point scored in a match) to live implied probabilities. Huang's algorithm achieved an extremely high degree of success and robustness at inferring point-by-point progression, achieving over 90% accuracy between multiple games at score inference. Huang also presented a suitable method of initiating the point winning probabilities on serve for a tennis Markov chain. Huang gives high level overview of the structure of their algorithm but does not delve into detail the fine tuning of parameters and variables such as the gap between the next state Markov probabilities and the rate at which the win-on-serve probabilities are updated. As such, it may be difficult to adapt their work into our models. However, we may use their algorithm structure as a framework for our feature extraction process, as we will use a similar approach to Lerner et al. (2019) presented in the following section.

Jackson and Mosurski (2012) explores the statistical implications of psychological variances of in-play and inter-play tennis matches. In their study, Jackson and Mosurski have found significant effects of heavy defeats, serve-breaks, and in-play dynamics on the point-winning probabilities on serve. We conclude that it is appropriate to consider how we can integrate these factors into our market making models.

2.2 On Machine Learning in Tennis and Sports Betting

There are multiple approaches to apply machine learning techniques to enhance in-play and pre-match tennis prediction. Our market making model will aim to combine time series forecasting and tennis modelling techniques to aid our market maker to make intelligent bids in the market.

A Stanford project by Lerner et al. (2019) investigates tennis prediction by applying in-play and pre-game data to a Long Short-Term Memory neural network model [3]. DeepTennis, the prediction model developed by Lerner et al., combines predictions by models developed by Gollub (2017), in-play scoring data, serve statistics, and run distance statistics to train an LSTM model capable of predicting the matching wining probabilities of a given player. DeepTennis was able to show state-of-the-art performance compared to existing varieties and achieved a 79.5% net accuracy at predicting implied tennis odds. It is worth mentioning that the model developed by Lerner et al. focuses on predicting the *current* probabilities. If we are to adapt a model for the purpose of market making, it would be appropriate to develop a model which forecasts the *future* probabilities. Furthermore, it is important that we base our market making decisions on more than historical data.

When training time-series models, we may find difficulties in training effective models when there is limited amount of historical time series data available within a single match. Xu et al. (2021) explores and show the possibility of using transfer learning to enhance the capabilities of LSTM-based stock price forecasting models [4]. Xu et al. postulates that transfer learning may be applied to stock price prediction when there is a lack of historical data available for a stock. Xu et al. was able to compare the performance of transfer learned to direct learned models and found that transfer learning was able to mitigate the limitations of a small sample size, while models trained on larger datasets show no significant improvements when transfer learning is employed. We can apply the same concepts into our in-play forecasting models, which would help address the issues of overfitting and lack of sample size.

2.3 On Market Making in Tennis and Sports Betting

While the saying that *"Past Performance Is Not Indicative of Future Results"* may not always be true, we should caution ourselves when basing our market making decisions on historical data which are too far into the future. Our market making decisions should be based on a multi-faceted approach which considers not only historical price movements, data, and statistics, but also psychological aspects of market behaviour, and the analyses of risk factors of investments. Occhiphinti (2015) provides a case of market making in a horse betting market [5]. In their study, Occhiphinti develops mathematical models in modelling the spread, price movement, and liability of a horse in the market.

Trader or bettor biases often affect betting markets due to the speculative nature of sports betting, resulting in market inefficiencies which more scrupulous traders may exploit. In an article posted in the Journal of Economics and Finance, Krieger et al. [6] specifically focus on the phenomenon of recency bias by bettors in National Football League games in 2007-2019, and how this creates profitable opportunities for more prudent bettors.

3 Goals and Objectives

1. To research and develop an enhanced Markov model to model tennis games
    a. Research and develop simple Markov models on python
    b. Research and develop methods to enhance the Markov models to address the weaknesses of Markov models in the context of tennis betting
    c. Implement Enhanced Model on simulated data and subsequently on real-life exchanges and beat the starting prices of tennis bets
2. Implement a marker maker and backtest on both simulated and historical exchange tennis odds
    a. Develop a simple market maker for tennis betting using python

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

b. Enhance the implementation to predict or forecast favourable prices and moments for market-making
c. Implement methods to account for and correct biases such as the favourite-longshot bias in the market
d. Implement methods to predict the optimum stake sizes in the market
e. Implement methods for the market-maker to react to events throughout the match.
f. Implement methods to determine the optimum position in the market using existing pre-play and in-play data.
g. Back-test the market maker on simulated and real exchange prices

4 Background and research of work

Our research aims to find solutions to developing an improved Markov chain model and advanced market-making model in tennis betting using statistical analysis, feature extraction, and machine learning techniques. In this section, we introduce a high-level overview of the application of our work in the game of tennis, betting markets, and in-play tennis betting. We then explain the theoretical and scientific knowledge underpinning our work, including feature engineering and extraction from datasets, Markov chains, solutions to enhancing a tennis Markov chain using machine learning ensemble models and developing a transfer learning LSTM market making algorithm.

4.1 Structure of tennis

The general structure of tennis comprises three primary levels, from highest to lowest, the match level, the set level, and the game level. In conventional game-level scoring, the first player to win four or more points with a 2-point margin takes the game. As a result, if both players reach 3 points, or the score goes to 40-40 or 'deuce', the game is played until a player gains a 2-point margin over the other. Scoring in the match and set levels may have more variation depending on the competition rules. In general, sets are won when a player wins at least 6 games, with a 2-point (i.e., games won) lead over their opponent. In the event where the set score is tied 6-6, a tiebreaker game is played which determines the set winner. The format of the tie-breaker game is also subject to substantial variation depending on the competition authority. The most common case is the '12-point tiebreaker' where a player wins with at least 7 points with a 2-point margin. Scoring in the match level is usually less complicated in which men's singles are often played in a best-of-five series, while women's singles are usually played in a best-of-three series. Competition authorities may also implement other different rules and intricacies.

4.2 Betting exchanges

Betting exchanges operate similarly to bookmakers, in which users can place bets on the outcome of sporting, political, or other events. Betting exchanges act as liquid markets where users play their bets against each other rather than the odds determined by bookmakers. Users can place *back*[1] or *lay*[2] bets at selected prices on discrete outcomes, which are matched by an opposite bet, typically using a cross-matching algorithm. While betting exchanges take a portion of profits, they are easier to generate profit from due to the non-inclusion of bookmakers over rounds. In our work, we focus on

[1] Bettor bets on the outcome to occur. For example, to back a tennis player is to bet on them to win.
[2] Opposite of a back bet where bettor bets on an outcome to not occur. For example, laying a tennis player is to bet on them to lose.

4.3 Market making

Market makers provide liquidity while profiting from illiquid markets by offering high volume trades at different bid and ask prices or a 'bid-ask spread' in short. In the context of equity markets, a market maker would bid for a security (e.g., a stock) at a slightly lower price than the price it asks for the same security.

The striking similarities between equity markets and betting exchanges allow traders to implement market making into sports bets by taking advantage of the spread between back or lay bets. In addition, Betfair's cross-matching algorithm makes it easier for offers to be matched when an opposite and equal back or lay bet does not exist.

Our market making program will consider the following elements when determining the optimal market making strategy:

- Optimum stake sizes and order limits.
- Market position (back and lay prices to take).
- Market momentum.
- Market depth.
- Expected future events that may affect the market.

To determine an optimum stake size, we will use the Kelly criterion in the context of tennis betting which is computed by:

$$K\% = \frac{bp - q}{b}$$

Where:

$K\%$ is the Kelly percentage that is a fraction of the portfolio to bet,

b are the decimal odds minus 1,

p is the expected probability of coming on out top in the bet,

q is the expected probability of losing the bet.

We determine a good inventory limit by considering the market depth and momentum. The market depth is the variance in the prices available to lay or back, while the market momentum is the quantification of how well the market can sustain an upward or downward trend. We use the Volume-Weighted Average Price (VWAP) to compute an indicator for market momentum, as such:

$$VWAP = \frac{\sum P \times V}{TV}$$

Where:

P is the traded price of a security,

V is the volume traded at that price,

TV is the total traded volume of the time period.

VWAP has shown to be a good indicator for momentum when used as a comparison to the price trends of a stock or security.

Betting markets are unique in that they often display an extremely high degree of *market efficiency*[3], which we will discuss further in section 4.5. This means that an effective market making should consider how in-play events and dynamics will affect the market. We will attempt to achieve this by developing an LSTM model capable of regressing over in-play data, statistics, as well as market information to be integrated into our market making program.

4.4 Betfair tennis markets and in-play betting

Betfair allows bettors to place gambles on different 'markets,' which define the odds of different outcomes for an event (a single match). For example, in 'Set Betting,' bettors would place bets on the final match score (number of sets won by each player) of an event, and in 'Total Games' betting, one would place bets on whether the final number of games played would be higher or lower than a number.

The primary and most popular market is the 'Match Odds' market, where punters would place bets on an event's winning or losing player. When a back bet of stake B is placed on back odds price O_0 and matched with an opposite lay bet using Betfair's cross-matching algorithm, and the odds at the same book position changes to the new price O_1, we can find the returns on a cash-out by:

$$R_{back} = \frac{O_0}{O_1} \times B \tag{4.2}$$

Conversely, when one acts as a bookmaker and places a lay bet with a layer's liability of L, the returns on a cash-out would be

$$P = 1 - \frac{1}{O} \tag{4.3}$$

The return on a cash-out with liability L can be found to

$$R_{lay} = \frac{P_0}{P_1} \frac{L}{P_0 - 1} \tag{4.4}$$

or if we take $\frac{L}{P_0 - 1} = B$ where B is the opposite backer's stake,

$$R_{lay} = \frac{P_0}{P_1} B \tag{4.5}$$

As we can expect, a back bettor would want the backing odds to decrease, while a lay bettor would want the backing odds to increase. When an outcome is settled, the backer's odds will decrease to 1 if the backed player wins, while the layer's odds will decrease to 1 if the backed player loses.

[3] How well the prices and movements of a market fully and fairly reflects all past information in existence.

For statistical and modelling purposes, we can convert market odds into inferred probabilities by simply taking the reciprocal of the odds by

$$P_1 = \frac{1}{odds} \tag{4.4}$$

where P_1 is the inferred probability of player 1 winning the match and $odds$ are the backer's odds. We elaborate on this further in section 4.5, explaining how previous works have shown that the exchange prices reflect actual match-winning probabilities.

4.5 Inferring the Probability of Winning on Serve and Match-Winning Probabilities

The most common method of inferring point-winning probability on serve is using the historical probabilities of players winning their service game. Klaasen and Magnus (2001) [7] found that the average point-winning probabilities on serve are 0.645 for men and 0.559 for women. Huang (2011), mentioned in section **Error! Reference source not found.**, uses this a s a probability inference tool in their model by comparing historical point-winning data and the implied probabilities arising from market prices. For instance, if the implied probabilities are equal, then the point-winning probability must be like the average probabilities. On the contrary, if the implied probabilities favour one player, we can find their point-winning probability on serve by fitting the implied probabilities into a fixed sum of 1.29 for men or 1.118 for women.

Easton and Uylangco (2010) successfully modelled implied match-winning probabilities from Betfair odds by simply taking the reciprocal of the bid's midpoint and asking prices for the player more favoured to win. Easton and Uylangco build upon the work of Klaassen and Magnus (2003) and found that the match-winning probabilities from Klaassen and Magnus' Markov chain model show a high correlation to the probabilities implied by match odds. Easton and Uylangco concluded that betting exchange prices reflect in-play match winning probabilities and that betting markets like Betfair exchange display high market efficiency. Furthermore, Easton and Uylangco discuss how both the Markov model and implied betting odds fail to incorporate the psychological implications of service breaks. Neither probability reflects the evidence of players losing a more considerable proportion of serve points after a service break. As such, a significant area of psychological dynamics in tennis is to be explored, which we will explore further in section 4.7.

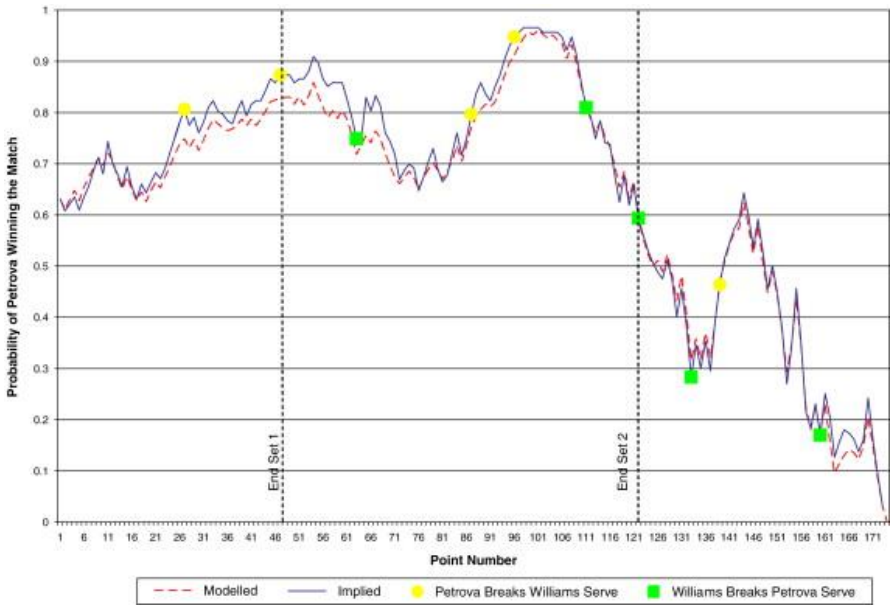

Figure 1, Implied and modelled winning probability of Petrova winning the match against Williams, 19 January 2007. (Easton and Uylangco, 2010)

4.6 Markov chains and tennis modelling

Markov chains are stochastic and probabilistic time-dependent processes that contain the *Markov property*[4]. In processes that comprise the Markov property, future events depend only on the immediate present, and such stochastic processes are memoryless.

The well-structured nature of tennis makes it easier to model than other sports using Markov models. Various levels of a tennis match (i.e., the match, set, and game levels) can be modelled as discrete-time Markov chains. Discrete-time Markov chains are a sequence of random variables that hold the Markov property. These processes can be mathematically defined as follows:

$$Pr(X_{n+1} = x \mid X_1 = x_1, X_2 = x_2, \dots, X_n = x_n) = Pr(X_{n+1} = x \mid X_n = x_n)$$, *for a given sequence of random variables* $\{X_n \mid n = 1, 2, \dots\}$, *is stated to be a discrete-time Markov chain*.

In our discrete-time Markov chain, we treat every possible winning or intermediate score outcome of every level of a tennis match as defined states, which the random variable X_t may take from the state space.

In a simple tennis Markov chain model, such as the ones developed by Lui (2001) and Klaassen and Magnus (2011), we assume that a tennis match always holds the Markov property. The naive assumption of the Markov property means that we assume that the probability distributions of subsequent points scored are independent and identical (i.i.d.).

These approaches assume that the probability of winning on serve remains constant throughout matches. However, research has shown that this is untrue and that many physical and psychological factors can change throughout the game, making crucial points such as match points or points in a set or match's later stages more challenging to win.

[4] Processes that only depend on the present state and no past states.

4.7 Psychological Variations in a Tennis Match

In section 2.1 we discussed the work of Jackson and Mosurski (2012) in analysing the statistical implications of psychological variations in tennis matches. Here, we will discuss this further and how we may incorporate these considerations into our model.

Jackson and Mosurski conducted a study on 501 Wimbledon and U.S. Open matches from 1987 to 1988, analyzing 1847 sets. Their findings suggest that psychological momentum plays a significant role in tennis prediction. They observed that player performance and the probability of winning a set fluctuate throughout a match. To investigate this further, they developed logistic regression models. One model focused solely on player ability based on ATP rankings, while the other model considered both player ability and the ongoing score of the match. Incorporating the ongoing score provided a better fit to the actual results, regardless of accounting for normal randomness.

From these findings, we can conclude that the progression of tennis scores may not deeply hold the Markov property. We should also consider how psychological momentum arising from score progression may point-winning probabilities. While previous findings show how the success of winning a set is affected by momentum, we must convert these probabilities into the success of winning a serve to be useful in our modelling approach. It is also highly plausible that momentum effects can explain the likelihood of winning both sets and games or even points. Furthermore, the psychological effect of breakpoints or tiebreakers may influence first and second serve percentage, making these points more difficult to win [8].

We incorporate these variables into our work through numerous methods. To model pre-play psychological and other systemic effects on point winning percentage, we train an ensemble model on head-to-head statistics and find the influence of features such as current form, rank difference, and the number of points won in a match. To model in-play psychological effects, we use an LSTM neural network to extract such features in the temporal domain through market information such as the progress of the inferred match odds, inferred match score, bet-lay spread, and the back or lay volumes as a proportion of the total traded volumes.

4.8 Datasets and Data Processing

Our machine learning models are trained using datasets which are appropriately selected for their respective intentions. We train and validate our ensemble model to predict a point-winning probability on serve on Jeff Sackmann's *ATP Tennis Rankings, Results, and Stats* dataset available on GitHub, which contains data on ATP individual player rankings, ranking points, and head-to-head match results and statistics of ATP tournaments.

Our market making LSTM model is trained and validated on historical information on tennis betting markets obtained from Betfair Historical Data. Our dataset includes market data from July 2018. We will focus on modelling and validating in-play market data in the 2018 Wimbledon Championships due to higher volumes leading to lower uncertainties, and a greater cross-matching frequency compared to other matches.

4.8.1 Betfair Historical Data

Betfair provides timestamped historical data available for purchase on the *Betfair Historical Data* archive. The *Historical Data* database shows information which are updated at a frequency up to 50 milliseconds and includes timestamped market information such as match start times, runner or player identification information, event IDs, and market changes. Data

on market changes includes changes in the price and volumes of available back and lay odds for each runner, last price traded or matched in the market, and the total traded volume of each runner.

4.8.2 ATP Tennis Rankings, Results, and Stats

Jeff Sackmann's *ATP Tennis Rankings, Results, and Stats* contains head-to-head statistics, results, and player rankings of Men's singles and doubles spanning from 1968 to 2023 and is consistently updated as new ATP matches are played. Our ensemble model will train on extracted features from statistics in head-to-head matches and ranking data for unique players. We will extract these features on data between the period of 2011 and July 2018 to ensure that there is enough training and testing data available, while avoiding the look-ahead bias.

4.9 Long short-term memory and recurrent neural networks

Recurrent neural networks (RNN) belong to a class of artificial neural networks where nodes within the network exhibit cyclic behaviour whereby the input of a node is a function of the previous outputs of the same node. This behaviour allows recurrent neural networks to capture features in the temporal domain more effectively when compared to classical feedforward neural networks or convolutional neural networks. Recurrent neural networks' capabilities in storing past dependencies allow them to be effective at autoregressive tasks.

LSTM was developed to address the vanishing and exploding gradient problem in traditional RNNs during backpropagation through time (BPTT). In BPTT, an RNN is treated as an unfolded deep feedforward neural network. Gradients in earlier timesteps become unstable due to their dependence on all previous weights, leading to vanishing or exploding gradients. This instability limits the ability of traditional RNNs to learn long-term dependencies, resulting in deficient performance and longer training times. LSTM solves this problem by introducing specialized memory cells that selectively retain and update information over time.

The basic architecture of the most common LSTM layer consists of an input gate, a forget gate, an output gate, and two states, a cell state, and a hidden state, which are transferred to the next time step cell.

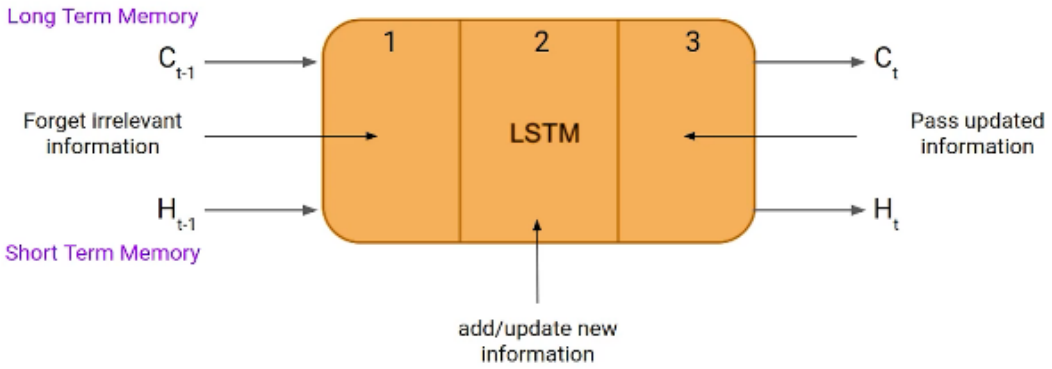

Figure 2, Simple diagram of an LSTM layer (Analytics Vidhya, 2021) [9].

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

Figure 2 shows the inputs and outputs of an LSTM layer. C_t denotes the cell state at time t, while H_t denotes the hidden state for the LSTM layer at time t. When new information is passed, the network determines whether old information should be forgotten by computing

$$f_t = \sigma(x_t U_f + H_{t-1} W_f)$$

Where:

x_t is the input vector at t,

U_f weight associated with the input,

H_{t-1} is the hidden state of the previous time step,

W_f is the weight matrix associated with the hidden state.

The result of f_t will be between 1 or 0 due sigmoid function. The layer forgets all previous information if $f_t = 0$ and retains all previous information if $f_t = 1$.

The input gate quantifies the importance of new information of information input into the network.

$$i_t = \sigma(x_t U_i + H_{t-1} W_i)$$

Where:

U_i weight associated with the input,

W_i is the weight matrix associated with the hidden state.

As with before, the input gate determines information to be important if $i_t = 1$, vice-versa.

The LSTM layer updates itself with new information by:

$$N_t = \tanh(x_t U_c + H_{t-1} W_c)$$

Where:

U_c weight associated with the input,

W_c is the weight matrix associated with the hidden state.

The cell state is then updated with this information by

$$c_t = f_t c_{t-1} + i_t N_t$$

which is a combination of all previous computations.

Finally, the LSTM activates its output gate based on the hidden state and output weights:

$$o_t = \sigma(x_t U_o + H_{t-1} W_o)$$

We then update the hidden state of the next time step accordingly by

$$H_t = o_t \tanh(c_t)$$

The structure of LSTMs (Long Short-Term Memory) solves the vanishing gradient problem by allowing the network to selectively retain or forget past information based on its

importance, facilitating more effective feature extraction from long-term dependencies. We will train and apply an LSTM model to aid our market making process by providing forecasted probabilities based on the in-play progression of scores and market information.

4.10 Addressing the Limitations of a Limited Sample Size

The typical best-of-five match lasts for 2 hours and 45 minutes. If we want to make the market in most of the duration of a match, this leaves us a minimal amount of training data within the match to be made available to train, yet alone validate, our models.

We may employ multiple options and techniques to overcome the difficulties of limited historical time series data. For example, we may augment our training data in hopes that the generalisation capability of our model is improved. Another technique is to employ transfer learning on previous matches played in a similar setting (i.e., court type, scoring system, singles or doubles, men's, or women's). Our approach would encompass training models using different techniques, hyperparameters, and architectures and finding the best-performing model through cross-validation on validation data.

4.10.1 Data Augmentation

Data augmentation is a machine learning technique that reduces overfitting and increases generalisation when training machine learning models by slightly modifying the input data before fitting a model. Data augmentation techniques are widely exploited in computer vision and digital image processing. For example, images would typically be randomly rotated, mirrored, colour filtered, or stretched before being passed into a machine learning algorithm for training.

We can apply similar principles to time series data by adding Gaussian noise or phase-shifting random time series segments. We may also augment our time series data by flipping the input. In doing so, we hope the models can capture a more significant variance of auto-regressive and cross-sectional patterns in the dataset.

4.10.2 Transfer Learning

Transfer learning is a machine learning technique which involves adapting or reusing models previously trained to perform other tasks on another similar or related goal. Transfer learning is commonly applied to computer vision or natural language processing due to the extensive amount of data and processing power required to train a model to perform these tasks.

A pre-trained model, often trained on a significantly larger dataset for a similar task, is used as a starting point in transfer learning. The motivation behind transfer learning is that pre-trained models would have gained knowledge that would apply to different yet similar tasks, especially when faced with limited sample sizes and long computational times.

Our approach uses transfer learning to help the model avoid overfitting the limited time series history available in a match while also increasing the generalisation capabilities of models to infer predictions from unseen patterns in features and data. Using pre-trained weights would also help decrease the computational complexity required to train an initial model.

Xu et al. who were mentioned earlier implemented transfer learning on a LSTM model by first taking a pre-trained model trained on enough samples, then replacing the final fully connect layer before training the model. To effectively employ transfer learning, we assume that individual matches are i.i.d., which may not always be the case. However, we may

assume that dependence effects between matches are insignificant, and that the results of previous matches played do not have any effect on the in-play dynamics of subsequent matches.

4.11 Feature Extraction and Training the Neural Network

We consider a range of features which we could infer from the *Historical Data* provided by Betfair. These features will include in-play inferenced score information and market information. We will undertake a feature selection process using cross-validation on a validation dataset.

4.11.1 Market Features

Occhiphinti (2015), who we mentioned earlier in section 2.3, developed a market making strategy in horse betting, which we could adapt into a similar tennis market making algorithm. The basis of Occhiphinti's market making strategy is to quantify the liability of each runner in a race and determine the optimal market position based on this liability. Occhiphinti develops models to find the future expected back-lay mid-price and the expected back-lay spread in the market. The expected mid-price is calculated as:

$$E_{ltp} = Best\ BP \times (1 - Pup) + Best\ LP \times Pup$$

Where:

E_{ltp} is the expected mid-price and, therefore, an indication of the traded price,

$Best\ BP$ is the best price available to back,

$Best\ LP$ is the best price available to lay,

Pup is the probability of upward price movement.

Pup is intuitively computed as:

$$Pup = \frac{Best\ back\ volume}{Best\ back\ volume + Best\ lay\ volume}$$

We will include this metric as a regression variable in the forecasting model.

To quantify uncertainty in the market, we obtain a spread between the back and lay prices and then divide by the most recent traded price. This will also be used to regress our neural network.

4.11.2 Score Inference

Our score inference algorithm is based on *Heuristic 1* of Huang's work discussed in section 2.1. In the most basic variation of the algorithm that Huang developed, win-on-serve odds are initialised based on the assumptions made in 4.5. These probabilities are taken as the serve probabilities at time t = 0. Match winning probabilities are then computed for the next score state using a tennis hierarchical Markov chain model, and we compare these probabilities to the market implied match probabilities to determine whether a player has scored. Huang's algorithm goes into greater complexity, including fine-tuning the serving probabilities as the match goes on, detecting false-positive crossings, and adjusting the algorithm to detect multiple crossings or changes at once. We will implement this approach to obtain feature vectors of scores throughout matches.

Using our inferenced scores, we can obtain a match probability using our ensemble learning enhanced Markov model and attempt to regress our neural network using this as a feature.

5 Method

Our model will comprise four stages of development. First, we develop a suitable Markov chain tennis model to be used for score inference purposes. Second, we build upon this model and enhance it using an ensemble model to determine the probabilities of winning on serve. Third, we develop and train a transfer learning LSTM model to forecast an expected match probability. Finally, we combine all previous steps and develop as well as backtest market making programs of different strategies.

5.1 Markov modelling

To model a tennis match, we divide it into separate levels represented by individual Markov chains: game level, set level, and match level. A special Markov chain is developed for tie-break situations. The game level focuses on the probability of the serving player scoring a point during their service game. By assuming point independence, we can model an entire game using these probabilities. The Markov chain represents the possible transitions between winning, losing, and deuce states. This approach allows us to capture the dynamics of a single game within a tennis match.

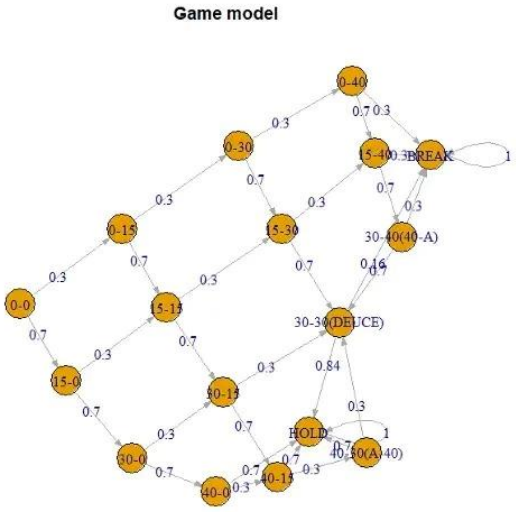

Figure 3, Markov chain representing game level. (Cararo, 2021)

We begin by developing a function to calculate the transitional probability matrix for the game level. This matrix represents the probabilities of transitioning from one state to another. Another function utilises this matrix to determine the steady state probabilities, which indicate the long-term probabilities in each game steady state. By iteratively multiplying the transition matrix, we can converge on the stable state probabilities. These probabilities reflect the likelihood of the serving player winning the game at any state. Finally, we multiply the final matrix with the state matrix to obtain the likelihood of winning from any state.

The tie-breaking model is like the game model but has its own special purpose. This Markov chain is only in the eventuality of a tie break. This is when both players have both won 6 games. A special game is then played to determine the winner of the set. Within this game a

player serves once then the next player serves twice, then the initial player serves the twice. And it continues to go on like that. The fundamental principles of the functions used for this Markov chain are the same as before. The first function takes the probability of each player scoring a point when they serve and builds the transition matrix. The next function then finds the steady state probabilities for the Markov chain and multiplies final matrix by the state matrix of the state to get the winning probabilities from that state. The figure below gives a visual representation of the Markov chain of the tie-break game.

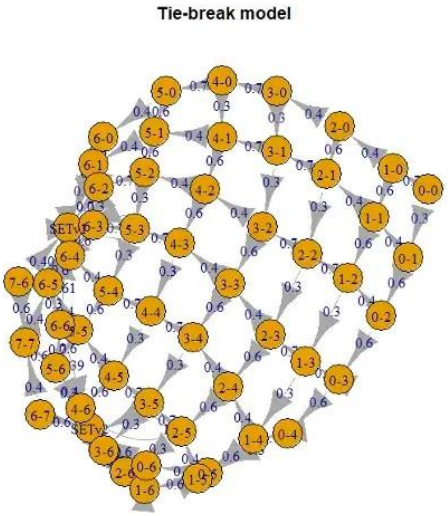

Figure 4, Visual representation of Markov chain modelling tie-break game [23]. (Cararo, 2021)

With these Markov chains created. It is then possible to build the Markov for the next level of the tennis match, the Set level. The principles are followed with a function being developed to create the transition matrix for the set level Markov chain. The probabilities needed are the probability of each player winning their service game and the probability of the player being investigated winning the tie-break game. With this being fed into the next function to find the stable state probabilities we can then obtain the probabilities of the investigated player winning a set and by multiplying the set state matrix of any given state at the level. The function can give the probability of winning a set from any state at the set level.

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

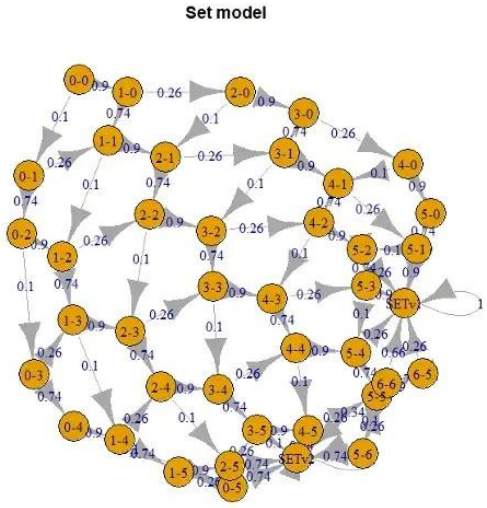

Figure 5, Visual representation of the Markov chain modelling a set. (Cararo, 2021)

With this level completed we can develop the Markov chain for the final and highest level, the match level. The same principles follow. The probability of winning a set from the previous function is fed into the function to build a transition matrix for the match level. The next function takes this matrix and outputs the final matrix that holds the stable state probabilities. Which when multiplied by the state matrix of any state at the match level will give the winning probabilities from that given state.

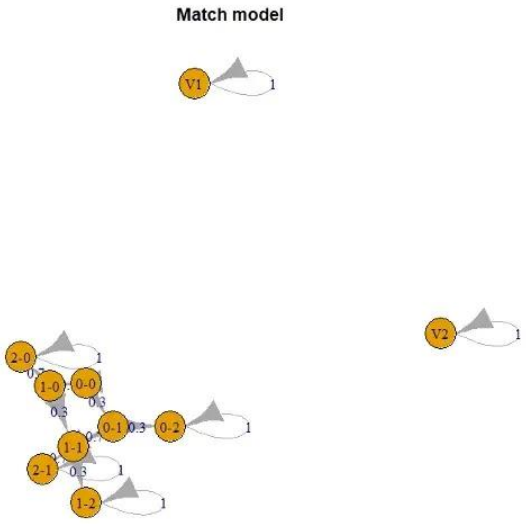

Figure 6, Visual representation of match level Markov chain. (Cararo, 2021)

By combining all the Markov chains, we have a model that can give the winning probabilities of an entire match from any point of a tennis match.

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

5.2 Markov Model Enhancement

Our aim of enhancing a vanilla Markov model is to ensure the accuracy of the point-winning probabilities on serve which are input into the Markov chain to calculate a match winning probability. Relying on historical serve-winning probabilities may prove unreliable as varying sources would give different information and data may be collected from different time frames. To increase the amount of control over the data we use, we train an ensemble model to predict the serve-winning probabilities by regressing over a range of features extracted from Sackmann's *ATP Tennis Rankings, Results, and Stats*.

5.2.1 Feature Engineering

To account for the limitations of the Markov chain model and the player momentum we first must perform preprocessing and feature engineering on the dataset. The first script in the code structure focuses on extracting and processing relevant features from historical tennis match data. It starts by combining multiple years of tennis match data into a single dataset.

```
import pandas as pd
import itertools

# Load the dataset
data = pd.read_csv("atp_matches_2023.csv")
data2 = pd.read_csv("atp_matches_2022.csv")
data3 = pd.read_csv("atp_matches_2021.csv")
data4 = pd.read_csv("atp_matches_2020.csv")
data5 = pd.read_csv("atp_matches_2019.csv")
data6 = pd.read_csv("atp_matches_2018.csv")
data7 = pd.read_csv("atp_matches_2017.csv")
data8 = pd.read_csv("atp_matches_2016.csv")
data9 = pd.read_csv("atp_matches_2015.csv")
data10 = pd.read_csv("atp_matches_2014.csv")
data11 = pd.read_csv("atp_matches_2013.csv")
data12 = pd.read_csv("atp_matches_2012.csv")
data13 = pd.read_csv("atp_matches_2011.csv")

data = pd.concat([data, data2,
data3,data4,data5,data6,data7,data8,data9,data10,data11,data12,data13])
data.to_csv("combined.csv",index=False)
```

It does this by reading CSV files for each year and concatenating them into a single Data Frame. The combined data is then saved into a file called "combined_csv." Next, the script defines functions for calculating various features for each player, such as average rank point difference, recent form, average aces per match, and more. These functions are designed to extract meaningful statistics that could potentially influence a player's performance in a match.

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

```
def average_rank_point_difference(player):
    rank_point_diff_as_winner = data[data['winner_id'] ==
player]['winner_rank_points'] - data[data['winner_id'] ==
player]['loser_rank_points']
    rank_point_diff_as_loser = data[data['loser_id'] ==
player]['loser_rank_points'] - data[data['loser_id'] ==
player]['winner_rank_points']
    total_rank_point_diff = rank_point_diff_as_winner.sum() +
rank_point_diff_as_loser.sum()
    total_matches = data[(data['winner_id'] == player) | (data['loser_id'] ==
player)].shape[0]
    return total_rank_point_diff / total_matches
# 2. Recent form
def recent_form(player, num_matches=10):
    recent_matches = data[((data['winner_id'] == player) | (data['loser_id']
== player))].tail(num_matches)
    wins = recent_matches[recent_matches['winner_id'] == player].shape[0]
    return wins / num_matches

# 3. Average aces per match
def average_aces(player):
    aces = data[(data['winner_id'] == player)]['w_ace'].sum() +
data[(data['loser_id'] == player)]['l_ace'].sum()
    total_matches = data[(data['winner_id'] == player) | (data['loser_id'] ==
player)].shape[0]
    return aces / total_matches

# 4. Average double faults per match
def average_double_faults(player):
    dfs = data[(data['winner_id'] == player)]['w_df'].sum() +
data[(data['loser_id'] == player)]['l_df'].sum()
    total_matches = data[(data['winner_id'] == player) | (data['loser_id'] ==
player)].shape[0]
    return dfs / total_matches

# 5. First serve win percentage
def first_serve_win_percentage(player):
    first_serves_won = data[(data['winner_id'] == player)]['w_1stWon'].sum() +
data[(data['loser_id'] == player)]['l_1stWon'].sum()
    first_serves_made = data[(data['winner_id'] == player)]['w_1stIn'].sum() +
data[(data['loser_id'] == player)]['l_1stIn'].sum()
    return first_serves_won / first_serves_made
```

After defining these functions, the script iterates over all unique players in the dataset and computes the features for each player by calling the previously defined functions. The resulting features are then stored in a Data Frame called ‘features_df.’

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

```
# Calculate features for all players
unique_players = data['winner_id'].append(data['loser_id']).unique()
features_df = pd.DataFrame(columns=['player_id',
'recent_form','average_rank_point_difference', 'average_aces',
'average_double_faults', 'first_serve_win_percentage',
'second_serve_win_percentage', 'break_point_save_percentage'])

for player in unique_players:
    player_features = {
        'player_id': player,
        'recent_form': recent_form(player),
        'average_rank_point_difference':
average_rank_point_difference(player),
        'average_aces': average_aces(player),
        'average_double_faults': average_double_faults(player),
        'first_serve_win_percentage': first_serve_win_percentage(player),
        'second_serve_win_percentage': second_serve_win_percentage(player),
        'break_point_save_percentage': break_point_save_percentage(player),
    }
    features_df = features_df.append(player_features, ignore_index=True)
```

To prepare the data for the machine learning model, the script generates all combinations of server_points and receiver_points and creates a new dataset containing all combinations of players and valid scores. This dataset is merged with the ‘features_df’ Data Frame into a single Data Frame called ‘merged_dataset.’ The merged dataset is then saved into a file called “new_features.csv.” Generating the combinations of server and receiver points is an essential step in creating the game transition matrix for the hierarchical Markov chain model. The purpose of this is to represent all game states and the probabilities of transitioning between them based on the players’ serve win percentages.

By considering all combinations of server and receiver points, the model can accurately capture the dynamics of a tennis game at any given moment. This comprehensive representation allows the model to track the game’s progress as points are won or lost and adjust the probabilities of winning the match accordingly.

In addition, considering all point combinations enables the model to capture the unique rules of tennis scoring, such as the deuce and advantage points. This comprehensive approach ensures that the model accurately reflects the true nature of the game, resulting in more reliable and accurate match outcome predictions.

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

```
# Generate all possible combinations of server_points and receiver_points
all_scores = list(itertools.product(range(5), repeat=2))

# Remove invalid score combinations (e.g., 3-3, 4-4)
valid_scores = [score for score in all_scores if score not in [(3, 3), (4,
4)]]

# Generate a new dataset containing all combinations of players and valid
scores
combined_dataset = pd.DataFrame(columns=['player_id', 'server_points',
'receiver_points'])

for player_id in unique_players:
    for server_points, receiver_points in valid_scores:
        row = {
            'player_id': player_id,
            'server_points': server_points,
            'receiver_points': receiver_points
        }
        combined_dataset = combined_dataset.append(row, ignore_index=True)

features_df = features_df.dropna()

# Merge the datasets
merged_dataset = pd.merge(combined_dataset, features_df, on='player_id')

# Save the new features to a CSV file
merged_dataset.to_csv("new_features.csv", index=False)
```

Feature engineering plays a vital role in enhancing the predictive capabilities of the tennis match outcome model. By creating new features that capture essential aspects of player performance, the model can better understand the nuances of individual player characteristics and strategies, which leads to more accurate predictions of match outcomes.

For example, the serve win percentage was engineered as a new feature. This single metric combines the first and second serve win percentages, providing a more comprehensive representation of a player's overall serving effectiveness. Since serving is a critical aspect of tennis and often influences the outcome of matches, incorporating serve win percentage as a feature allows the model to better capture the serving capabilities of individual players.

By leveraging feature engineering to incorporate relevant and informative player characteristics, the model can more accurately identify patterns and relationships within the data. This, in turn, allows the model to make more precise predictions, which is crucial for applications like betting, which is the end goal of this project.

5.2.2 Ensemble Model

The next step was training the model for predicting the serve win probability. The second script uses the processed data to train an ensemble model that predicts a player's serve win percentage based on these features. It begins by loading the dataset from the "new_features.csv" file. It calculates the serve win percentage as the average of the first and second serve win percentages for each player, and then removes unnecessary columns from the dataset. The data is split into features (X) and target variable (y) for model training.

To train and evaluate the machine learning models, the script first splits the data into train and test sets. It then defines GradientBoostingRegressor, RandomForestRegressor, and MLPRegressor models along with their corresponding parameter grids. Each model was selected due to its unique strengths and capabilities in capturing various aspects of the data.

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

```
import pandas as pd
from sklearn.feature_selection import RFECV
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.ensemble import RandomForestRegressor, VotingRegressor,
GradientBoostingRegressor
from sklearn.neural_network import MLPRegressor
from sklearn.metrics import mean_absolute_error, r2_score
import joblib

# Load the new dataset
data = pd.read_csv("new_features.csv")

# Calculate the serve win percentage as the average of first serve win
percentage and second serve win percentage
data['serve_win_percentage'] = (data['first_serve_win_percentage'] +
data['second_serve_win_percentage']) / 2

# Drop 'first_serve_win_percentage' and 'second_serve_win_percentage' columns
data = data.drop(['first_serve_win_percentage', 'second_serve_win_percentage',
'player_id'], axis=1)

# Split the data into features and target variable
X = data.drop("serve_win_percentage", axis=1)
y = data["serve_win_percentage"]

# Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create the GradientBoostingRegressor, RandomForestRegressor, and
MLPRegressor models
xgb = GradientBoostingRegressor(random_state=42)
rfr = RandomForestRegressor(random_state=42)
mlp = MLPRegressor(random_state=42)

# Define the parameter grid for each model
xgb_params = {'n_estimators': [50, 100, 200], 'learning_rate': [0.01, 0.1,
0.2], 'max_depth': [3, 4, 5]}
rfr_params = {'n_estimators': [50, 100, 200], 'max_depth': [3, 4, 5]}
mlp_params = {'hidden_layer_sizes': [(50,), (100,), (50, 50)], 'activation':
['relu', 'tanh'], 'learning_rate_init': [0.001, 0.01, 0.1]}
```

Gradient Boosting Regressor (XGBoost) is a powerful and popular machine learning algorithm that builds an ensemble of decision trees in a stage-wise additive manner. It minimises a differentiable loss function by using gradient descent, which allows it to capture complex patterns and interactions between features. XGBoost is known for its high performance and accuracy, making it a strong choice for predicting serve win percentages.

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

Random Forest Regressor is an ensemble learning method that operates by constructing a multitude of decision trees at training time and outputting the mean prediction of the individual trees. This model is highly effective at handling large datasets with a mix of continuous and categorical variables, and it is robust to overfitting. The random selection of features at each node in the tree makes the model less prone to bias, and the aggregation of multiple trees reduces variance, providing a more stable prediction.

Multi-layer Perceptron Regressor (MLP) is a type of feedforward artificial neural network that consists of multiple layers of nodes connected by weights. The model learns to map input features to output predictions by iteratively updating the weights using backpropagation. MLP is capable of modelling complex, non-linear relationships between features, and it can adapt to a wide range of problem domains. The flexibility of the model's architecture, such as the number of hidden layers and activation functions, makes it a versatile choice for predicting serve win percentages.

By combining these individual models into an ensemble, we can take advantage of their complementary strengths, leading to improved overall performance. Each model captures various aspects of the data, and the ensemble model aggregates their predictions to produce a more accurate and robust result.

A grid search with cross-validation is performed to find the best parameters for each model. Cross-validation is a technique that is used to evaluate the performance of the models by training and testing the models on different subsets of the data. This helps to avoid overfitting, as the model's performance is assessed on multiple sets of data, ensuring that it can generalise well to new data. The Grid search aspects tries different combinations of the parameters in the parameter grid to discover which combination provides the best accuracy. This helps with the hyper parameter tuning of the model to obtain the best possible model for the task at hand.

```
# Perform Grid Search and Cross Validation to find the best parameters for
each model
xgb_grid = GridSearchCV(xgb, xgb_params, cv=5,
scoring='neg_mean_absolute_error', n_jobs=-1)
rfr_grid = GridSearchCV(rfr, rfr_params, cv=5,
scoring='neg_mean_absolute_error', n_jobs=-1)
mlp_grid = GridSearchCV(mlp, mlp_params, cv=5,
scoring='neg_mean_absolute_error', n_jobs=-1)

xgb_grid.fit(X_train, y_train)
rfr_grid.fit(X_train, y_train)
mlp_grid.fit(X_train, y_train)
```

The best models from the grid search results are retrieved and combined into an ensemble model called VotingRegressor. This ensemble model is designed to improve the overall prediction accuracy by leveraging the strengths of each of the individual models.

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

```
# Retrieve the best models
xgb_best = xgb_grid.best_estimator_
rfr_best = rfr_grid.best_estimator_
svr_best = svr_grid.best_estimator_

# Create the ensemble model with the best models
ensemble = VotingRegressor([('xgb', xgb_best), ('rfr', rfr_best), ('svr',
svr_best)])

# Fit the ensemble model
ensemble.fit(X_train, y_train)

# Make predictions on the test set
y_pred = ensemble.predict(X_test)

# Calculate the performance metrics
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f'Test Mean Absolute Error: {mae}')
print(f'Test R^2 Score: {r2}')
```

5.2.3 Integration into Markov Chain Model

Following the Development of the ensemble. It was integrated into the hierarchical Markov chain model previously developed. This was achieved by incorporating the ensemble model into the 'prob_game' function which calculates the game-winning probabilities.

```
def prob_game(player_features, s_game):
    ensemble_model = joblib.load('ensemble_model.pkl')
    player_features_df = pd.DataFrame([player_features])
    matrix = game_trans_matrix(ensemble_model,player_features_df)
    matrix = np.linalg.matrix_power(matrix, 40)
    matrix = pd.DataFrame(data=matrix, index=col_row_names, columns=col_row_names)
    probs = np.dot(s_game, matrix)
    return probs
```

The ensemble model is used to populate the game transition matrix. To achieve this, the function game_trans_matrix was developed. This function plays an integral role in integrating the ensemble in the Markov chain Model. The function takes the ensemble model and the player features as inputs. Within the function there is a dictionary called transitions which maps each point score state to a list of next states. The function firsts create an empty game transition matrix, tMat1, with the same dimensions as the initial game matrix, game_mat, and with index and column names as col_row_names. The function then iterates over each points score state in the col_row_names. The iteration skips over the Win and Lose states as this are the absorbing states with fixed probabilities all the other states the function extracts the server and receiver points.

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

The player features are then updated within the current server and receiver points. The features are then passed to the ensemble model to predict the probability of the player winning the point on his serve. These probabilities are then used to assigned to their respective current states. The probability of transitioning to the next state where the server wins the point is set to the serve win percentage, while the probability of transitioning to the next state where the receiver wins the point is set to the complement of the serve win percentage (1 - serve win percentage).

```python
def game_trans_matrix(ensemble_model, player_features):
    matrix = game_mat
    point_mapping = {'0': 0, '15': 1, '30': 2, '40': 3, 'A': 4}

    tMat1 = pd.DataFrame(data=matrix, index=col_row_names, columns=col_row_names)

    possible_transitions = {
        "0-0": ["15-0", "0-15"],
        "15-0": ["30-0", "15-15"],
        "0-15": ["15-15", "0-30"],
        "30-0": ["40-0", "30-15"],
        "15-15": ["30-15", "15-30"],
        "0-30": ["15-30", "0-40"],
        "40-0": ["Win", "40-15"],
        "30-15": ["40-15", "30-30(DEUCE)"],
        "40-15": ["Win", "40-30(A-40)"],
        "40-30(A-40)": ["Win", "30-30(DEUCE)"],
        "0-40": ["15-40", "Lose"],
        "15-40": ["30-40(40-A)", "Lose"],
        "30-40(40-A)": ["30-30(DEUCE)", "Lose"],
        "15-30": ["30-30(DEUCE)", "15-40"],
        "30-30(DEUCE)": ["40-30(A-40)", "30-40(40-A)"],
    }

    for state in col_row_names:
        if state in ['Win', 'Lose']:
            continue
        next_states = possible_transitions[state]
        if "(" in state :
            server_points, receiver_points = state.split("(")[0].split("-")
        else:
            server_points, receiver_points = state.split("-")

        if server_points == "A":
            server_points = 3
        if receiver_points == "A":
            receiver_points = 3

        server_points = point_mapping[server_points]
        receiver_points = point_mapping[receiver_points]

        updated_player_features = player_features.copy()
        updated_player_features['server_points'] = server_points
        updated_player_features['receiver_points'] = receiver_points
```

After iterating over all the states, the probabilities for ‘Win’ to ‘Win’ and ‘Lose’ to ‘Lose’ transitions are set to 1, to ensure that these absorbing states are handled correctly. The function then returns the final matrix.

After the final matrix is passed back to the prob_game function, the function then calculates the game winning probabilities by raising the matrix to the power of 40 and is then multiplied by the initial game-state. The value 40 was chosen arbitrarily as it is assumed that the game winning probabilities would have converged by that point. This enhanced version of the prob game function will give a more accurate insight into how a game will play out and which will

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

then make the higher levels more accurate. By integrating the ensemble model into the hierarchical Markov chain model, the probabilities of winning individual points, games, sets, and the match, are now more accurate and reflective of the players' historical performance. This allows the model to provide better insights into the outcome of a tennis match and simulate more realistic match scenarios.

```
203             updated_player_features = player_features.copy()
204             updated_player_features['server_points'] = server_points
205             updated_player_features['receiver_points'] = receiver_points
206 
207             # Predict serve win percentage for the next state
208             ppoint_server = ensemble_model.predict(updated_player_features)[0]
209             ppoint_ret = 1 - ppoint_server
210 
211             for next_state in next_states:
212                 if state.split("-")[0] != next_state.split("-")[0]:
213                     tMat1.at[state, next_state] = ppoint_server
214                 else:
215                     tMat1.at[state,next_state] = ppoint_ret
216 
217 
218         tMat1.at["Lose", "Lose"] = 1
219         tMat1.at["Win", "Win"] = 1
220 
221         return tMat1
222 
```

5.3 In-Play Match Winning Probability Forecasting

Our market making program will integrate our forecasting model. The process begins with the exploration of the *Historical Data* dataset, which provides us valuable market data for analysis. Next, we construct an LSTM model architecture specifically tailored to our task of probability forecasting. Moving on, we employ feature engineering and data processing techniques to generate multiple feature vectors. These features will be used by our LSTM model for regression and forecasting purposes. Leveraging the power of transfer learning, we train our model to capitalise on pre-existing data of historical matches. Finally, we evaluate and cross-validate various models with different variations and hyperparameters, ensuring that we select the best models for integration into our market maker.

5.3.1 Developing the Model Architecture

Our model architecture comprises two vertically stacked LSTM layers, and two densely connected outputs as shown in figure 7.

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

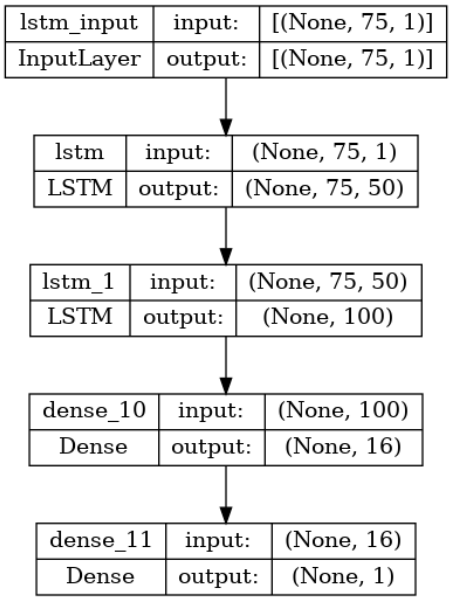

Figure 7, LSTM model architecture.

Stacking LSTM layers vertically will allow for better representation of more complex temporal features. By adding multiple LSTM layers on top of each other, the network can learn hierarchical representations of the input data. This approach is particularly effective in time series autoregression tasks, where capturing long-term dependencies is crucial. The stacked LSTM layers allow for the extraction of high-level abstract representations at different temporal scales, leading to enhanced model performance and the ability to capture intricate patterns and dynamics in the data.

5.3.2 Feature Extraction and Pre-processing

The necessary script and functions for this section can be found in the files "preprocess_dataset.py" and "preprocessing.py". Additionally, for a more interactive presentation of the feature extraction and preprocessing stage, you can refer to the Jupyter notebook "preprocessing.ipynb".

To start, we load the market data files line-by-line, and appending a of market and runner changes for each player into two lists. From the top and bottom of the file, we can obtain general market information such as runner IDs, runner names, match start timestamp, and the match winner. We will use this information later in our feature extraction process.

Once we have compiled a list of market changes for each player, we will construct an array of the average of the last traded prices between the two players. This array will be resampled at an interval of 5 seconds, as with the rest of our features. The Nyquist Theorem guides our choice of a sampling rate that is at least twice the highest frequency component to ensure we capture sufficient information. Given that the shortest tennis rallies typically have intervals of 10 seconds between them, we opt for a 5-second sampling rate to account for such dynamics. We convert the odds into the match winning probabilities for runner 1, and we extend the end of the Figure 8 shows the mid-point of the last traded price in the Wimbledon 2018 quarter-final matchup between Rafael Nadal and Juan Martin del Potro.

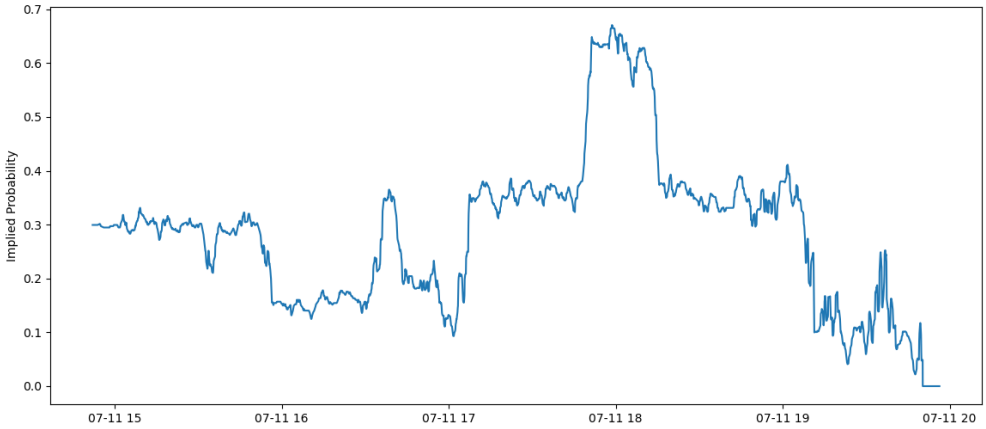

Figure 8, Probabilities implied by last prices traded of Nadal vs del Potro, Wimbledon 2018.

With the availability of implied match probabilities, we may now use a score inference algorithm to obtain a progress of scores across the match. Functions related to score inference can be found in the file "score_inference.py."

To commence the process, we initialize the win-on-serve probabilities, which will serve as inputs for our Markov model. This initialisation involves using a graphical method to compare the changes in Markov match probabilities as the win-on-serve probabilities of the players vary. Specifically, we observe how the Markov match probabilities fluctuate when the win-on-serve probability ranges from 0 to 0.645, while the total win-on-serve probability must adhere to the constraint of being equal to 1.29. We then compare the initial market implied probabilities to this observation as an initialisation to the win-on-serve probabilities.

From then on, our inference algorithm repeatedly checks for the condition whether the market implied odds would cross the thresholds of the Markov match probabilities for the next score state. We also build a degree of robustness to large and turbulent changes in the implied probabilities. This is achieved by holding off an update in the score until the algorithm is certain that this change would be permanent, that is, the implied probabilities do not return to the previous level within the next k time steps.

In our process, we were able to infer the match score of success, but we were unable to obtain an accurate inference of the set or game level scores, indicating the need to better fine-tune parameters such as the thresholds gap. It is important to note that large jumps are usually caused by serve breaks rather than an increase in match score. Therefore, our algorithm attempts to find the level of probabilities that correspond to a 0-0 set score for every match score the best. We use our inferred scores to obtain a vector of the match probabilities computed by our enhanced Markov model and use one hot encoding to include the scores in our features dataset. Figure 9 shows the normal and enhanced Markov probabilities compared to the market implied probabilities after score inference. The Markov odds show a to the market implied probabilities.

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

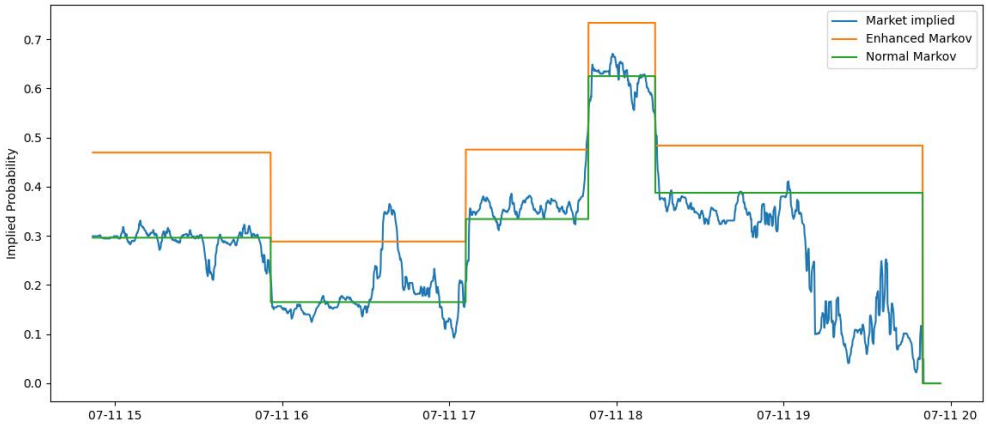

Figure 9, Markov, and market implied probabilities.

After we have obtained last traded prices and inferred match scores, we will process the back and lay prices and volumes for each player. For ease of training, we use a Hampel filter to detect and remove outliers from the back and lay volumes and prices. We clean up our data further using a 10-minute moving average filter. Phase shift incurred by the filtered will be compensated by a shift backwards of (N - 1) / 2 timesteps on the vector. Missing data points from resampling are filled and interpolated linearly. Finally, we compute a *pup* and spread using the back/lay price and volume data. Figure 10 shows *pup* plotted with the implied match winning probabilities for Milos Raonic.

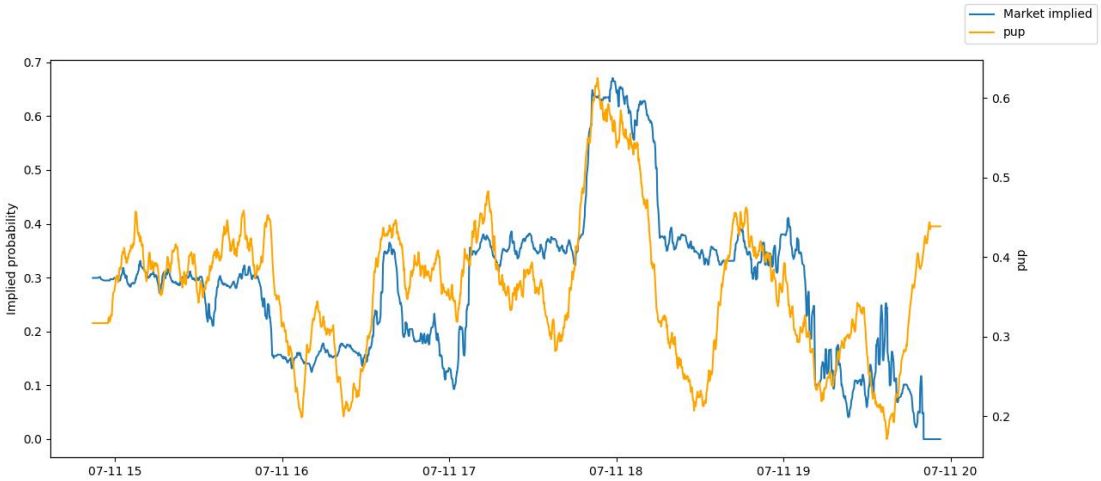

Figure 10, Market implied probabilities and pup.

Once we have compiled a data frame of our extracted features as well as inferred scores, we will save our features in a csv file contained in a folder. We repeat this process for all Historical Data files for Wimbledon 2018 men's singles matches.

5.3.3 Model Training, Cross-Validation, and Model Selection

In our model training and selection process, we explore different hyperparameters, layers, and architectures and find the best performing model based on a walk-forward forecasting validation.

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

In training our model, we have explored the use of sequence-to-sequence models for forecasting, but with limited success. Moreover, we compare variations of vanilla and bidirectional LSTM models with different hyperparameters.

During cross-validation, we set aside an initial 20% of validation match data to train on. We employ transfer learning by replacing the final two fully connected layers and retraining the model for a defined number of epochs on initial validation data.

Benchmarking of models and forecasts can be found in the "model_benchmark.ipynb" notebook. To benchmark the forecasting capabilities of a model, we would use our model to forecast three minutes of match probabilities ahead at fixed intervals. We retrain our models every 5 forecasts to update the model using new data. To determine the accuracy of forecasts, we take all values for the final forecast timesteps and find the errors compared to the true values and find the average RMSE (Root Mean Squared Error) or MSE (Mean Square Error). Figure 11 shows a walk-forward forecast validation on the match between Gilles Simon and Matthew Ebden.

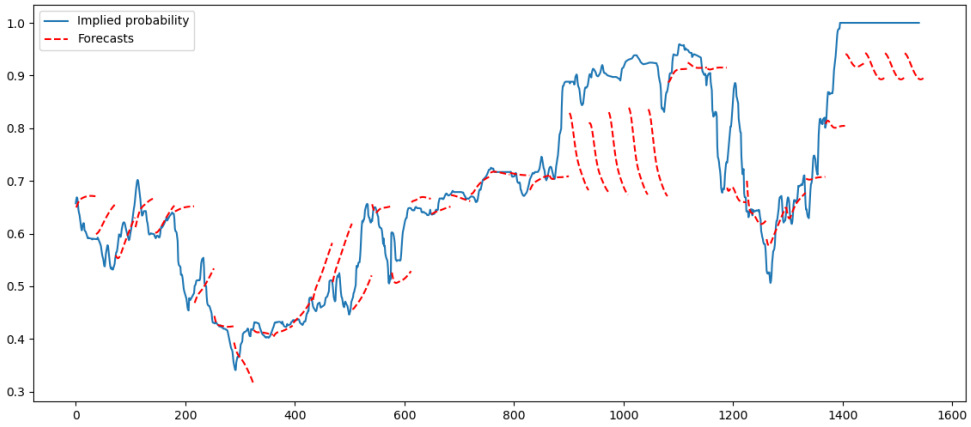

Figure 11, Forecast walk-forward validation results, Simon vs Ebden, Wimbledon 2018.

5.4 Market Making

With the completion of the enhance Markov model we can develop the Market Maker algorithm. The algorithm is structured within a Python class named Market Maker. Here some of the key functions and how they implemented.

The adjust_inventory_limit method dynamically adjusts the inventory limit, which is the maximum amount of money that can be bet on a specific runner. The adjustment is based on the market depth and momentum for that runner. By doing so, the model becomes more flexible and adaptive to market conditions, potentially leading to more profitable bets.

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

```python
def adjust_inventory_limit(self, runner_id):
    base_limit = self.base_inventory_limit
    market_depth = self.simulated_market.get_market_depth(runner_id)

    if market_depth < 1000:
        depth_factor = 0.25
    elif market_depth < 5000:
        depth_factor = 0.5
    else:
        depth_factor = 1

    market_momentum = self.get_market_momentum(runner_id)
    if market_momentum > 0:
        momentum_factor = 1 + market_momentum
    else:
        momentum_factor = 1 / (1 - market_momentum)

    position = self.positions[runner_id]
    adjustment_factor = 1 + (abs(position) / base_limit) * self.inventory_limit * depth_factor * momentum_factor
    adjusted_inventory_limit = base_limit * adjustment_factor

    # Set upper and lower limits for the adjusted inventory limit
    upper_limit = base_limit * 2  # You can set this value as needed
    lower_limit = base_limit * 0.5  # You can set this value as needed

    # Ensure the adjusted inventory limit is within the specified bounds while still using the original calculation
    adjusted_inventory_limit = max(min(adjusted_inventory_limit, upper_limit), lower_limit)

    return adjusted_inventory_limit
```

The update_scores_from_row function updates the current set and game scores based on the provided data row. This is critical in a dynamic sport like tennis where the scores and momentum can change rapidly, affecting the betting odds significantly. We use the scores processed within this function as an input for the Markov Model.

```python
def update_scores_from_row(self, row):
    if self.initial_server_id is None:
        self.initial_server_id = row["Server"]
        self.initial_receiver_id = row["Receiver"]
    if int(row["ServerGames"]) >= 6 or int(row["ReceiverGames"]) >= 6:
        return
    if row["Server"] == self.initial_server_id:
        set_score = f"{row['ServerSets']}-{row['ReceiverSets']}"
        game_score = f"{row['ServerGames']}-{row['ReceiverGames']}"
    else:
        set_score = f"{row['ReceiverSets']}-{row['ServerSets']}"
        game_score = f"{row['ReceiverGames']}-{row['ServerGames']}"

    self.current_scores = {
        "set_score": set_score,
        "game_score": game_score,
        "server_id": row["Server"],
        "receiver_id": row["Receiver"]
    }
```

The Kelly criterion method implements the Kelly Criterion, a well-known strategy used in gambling and investing to determine the optimal size of a series of bets, given the odds and probability of winning.

Functions like calculate_vwap, get_market_momentum, and get_runner_traded_volume are used to gather and process market data. They calculate the Volume Weighted Average Price

(VWAP), the momentum in the market, and the traded volume for a specific runner, respectively. This data provides essential insights into the state of the market and helps inform the betting strategy.

```
def kelly_criterion(self, odds, probability):
    return (odds * probability - (1 - probability)) / odds

def get_runner_traded_volume(self, runner_id):
    traded_volume = 0
    for market_data in self.simulated_market.market_data:
        for runner_data in market_data["bets"]:
            if runner_data["id"] == runner_id:
                traded_volume += runner_data["tv"]

    return traded_volume

def calculate_vwap(self, runner_id):
    total_volume = self.get_runner_traded_volume(runner_id)
    if total_volume == 0:
        return None

    weighted_sum = 0
    for market_data in self.simulated_market.market_data:
        for runner in market_data["bets"]:
            if runner["id"] == runner_id:
                weighted_sum += runner["ltp"] * runner["tv"]

    return weighted_sum / total_volume

def get_market_momentum(self, runner_id):
    ltp_list = []
    for market_data in self.simulated_market.market_data:
        for runner in market_data["bets"]:
            if runner["id"] == runner_id and runner["ltp"]:
                ltp_list.append(runner["ltp"])

    if len(ltp_list) < 2:
        return 0

    return ltp_list[-1] - ltp_list[-2]
```

The get_optimal_prices method calculates the optimal bid and ask prices for a runner. It uses the VWAP as a reference price and adjusts it based on the position risk and market momentum. This ensures that the model places bets at the most optimal prices, thereby maximizing potential profits and minimizing losses.

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

```python
def get_optimal_prices(self, best_back, best_lay, runner_id):
    # Calculate the VWAP and use it as the reference price
    vwap = self.calculate_vwap(runner_id)
    if vwap is None:
        vwap = (best_back + best_lay) / 2

    # (Existing logic for calculating optimal prices)
    position = self.positions[runner_id]
    inventory_limit = self.inventory_limit if self.inventory_limit != 0 else 1
    position_risk = abs(position) / inventory_limit

    bid_adjustment = (1 - self.ema_alpha) * (1 + position_risk)
    ask_adjustment = (1 + self.ema_alpha) * (1 - position_risk)

    bid_price = vwap * bid_adjustment
    ask_price = vwap * ask_adjustment

    # Incorporate market momentum
    momentum = self.get_market_momentum(runner_id)
    bid_price += momentum
    ask_price += momentum

    return bid_price, ask_price
```

The place_bets method in the Market Maker class is the heart of the betting strategy. It combines various calculations and strategies to determine the optimal bets to place for each runner. Here is a more detailed look at this function and its components.

Adjust Inventory Limit: The function starts by adjusting the inventory limit for each runner. This is done by calling the adjust_inventory_limit method, which uses market depth and momentum to dynamically change the maximum amount of money that can be bet on a specific runner. This provides flexibility and adaptability to the betting strategy, allowing it to react more efficiently to changing market conditions.

Calculate Optimal Prices: The get_optimal_prices method is then called to calculate the optimal bid and ask prices for a runner. The method uses the Volume Weighted Average Price (VWAP) as a reference price and adjusts it based on the position risk and market momentum. This ensures the model places bets at the most advantageous prices, maximizing potential profits and minimizing losses.

Determine Bet Sizes: To determine the size of the bets, the function applies the Kelly Criterion via the Kelly criterion method. This helps to balance the potential for profit against the risk of loss, ensuring a sustainable betting strategy over the long term.

Factor in Win Probabilities: An important aspect of the place_bets function is that it factors in the win probabilities for each runner. These probabilities are calculated using the Markov model, which provides a sophisticated way to predict the outcome of a game based on current and past states.

Place the Bets: After all these calculations, the function places the bets on the market. It ensures that the sum of the amounts bet on all runners does not exceed the total amount available for betting.

```python
def place_bets(self):
    position_balance_weight = 0.3
    # Set maximum and minimum position limits for each runner
    max_position = 1000
    min_position = -1000
    # Update win probabilities using the Markov model
    mis, sis, gis, tbis = markov_model.initiate_markov_states()
    player1_features = {
        'server_points': 2,
        'receiver_points': 1,
        'recent_form': 0.6,
        'average_rank_point_difference': -485,
        'average_aces': 13.08,
        'average_double_faults': 2.94,
        'break_point_save_percentage': 0.67
    }
    player2_features = {
        'server_points': 2,
        'receiver_points': 1,
        'recent_form': 1,
        'average_rank_point_difference': 485,
        'average_aces': 7.66,
        'average_double_faults': 1.52,
        'break_point_save_percentage': 0.68
    }

    win_probabilities = markov_model.tennis_model(player1_features, player2_features, self.current_scores["set_score"], self.current_scores["game_score"], mis, sis, gis, tbis)

    win_probabilities_dic = {
        2519549: float(win_probabilities[0]),
        2251402: float(win_probabilities[1])
    }
    print(self.current_scores)
    for runner_data in self.simulated_market.market_data[-1]["bets"]:
        runner_id = runner_data["id"]
        best_back = runner_data["odds"]["back"][0][0] if runner_data["odds"]["back"] else None
        best_lay = runner_data["odds"]["lay"][0][0] if runner_data["odds"]["lay"] else None

        if best_back is None or best_lay is None:
            continue
```

```python
    # Adjust the inventory limit for the current runner
    self.inventory_limit = self.adjust_inventory_limit(runner_id)
    print(f"Adjusted inventory limit for runner {runner_id}: {self.inventory_limit}")

    bid_price, ask_price = self.get_optimal_prices(best_back, best_lay, runner_id)

    win_probability = win_probabilities_dic[runner_id]
    print(runner_id)
    print(win_probability)
    position = self.positions[runner_id]

    # Calculate the weighted win probability for each runner
    weighted_win_probability = win_probability * (1 - position_balance_weight * (1 - abs(position) / self.inventory_limit))
    print(weighted_win_probability)

    # Use the win probabilities in the calculation of the inventory risk

    inventory_risk = abs(position) / (self.inventory_limit * weighted_win_probability)

    bid_adjustment = (1 - self.ema_alpha) * (1 + inventory_risk)
    ask_adjustment = (1 + self.ema_alpha) * (1 - inventory_risk)

    bid_price = bid_price * bid_adjustment
    ask_price = ask_price * ask_adjustment

    kelly_fraction = 0.6  # Adjust this value between 0 and 1 to control the weight of the win probabilities

    kelly_back = kelly_fraction * self.kelly_criterion(best_back, weighted_win_probability)
    kelly_lay = kelly_fraction * self.kelly_criterion(best_lay, 1 - weighted_win_probability)
    # Calculate the maximum allowed order sizes based on the inventory limit
    max_back_order_size = max(0, self.inventory_limit - position)
    max_lay_order_size = max(0, self.inventory_limit + position)

    # Calculate the desired order sizes based on the kelly fractions
    desired_back_order_size = max(0, position + self.inventory_limit * kelly_back)
    desired_lay_order_size = max(0, position - self.inventory_limit * kelly_lay)


    # Limit the order sizes to the maximum allowed values
    back_order_size = min(desired_back_order_size, max_back_order_size)
    lay_order_size = min(desired_lay_order_size, max_lay_order_size)
```


GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

```
# Limit the order sizes to the maximum allowed values
back_order_size = min(desired_back_order_size, max_back_order_size)
lay_order_size = min(desired_lay_order_size, max_lay_order_size)

order_back = self.simulated_market.place_order(runner_id, bid_price, "back", back_order_size)
order_lay = self.simulated_market.place_order(runner_id, ask_price, "lay", lay_order_size)


if order_back is not None:
    self.positions[runner_id] += order_back
    self.total_wagered += bid_price * back_order_size
    self.bets += 1
if order_lay is not None:
    self.positions[runner_id] -= order_lay
    self.total_wagered += ask_price * lay_order_size
    self.bets += 1

# Ensure that the position doesn't exceed the maximum and minimum position limits
self.positions[runner_id] = max(min(self.positions[runner_id], max_position), min_position)
profit = self.calculate_profit(2519549)
self.profits.append(profit)
self.inventory_sizes.append(self.positions[runner_id])

if profit > 0:
    self.num_winning_trades += 1
self.total_trades += 1
self.win_rates.append(self.num_winning_trades / self.total_trades)
```

6 Experimental Results

6.1 LSTM Model

We have trained and validated 9 different variants of LSTM transfer learning models on forecasting performance on the matchup between Gilles Simon and Matthew Ebden in Wimbledon 2018. These models are:

- Model A – 10 Epochs with only 1 feature (ltp)
- Model B – 5 Epochs with only 1 feature (ltp)
- Model C – 10 Epochs with 5 features (ltp, runner 1 and runner 2 pup, runner 1 and 2 spreads)
- Model D – 10 Epochs with ltp and scores
- Model E – 10 Epochs with ltp and pups
- Model F – 10 Epochs with data augmentation (flipped time series 50% of the time)
- Model G – 10 Epochs with enhanced Markov odds
- Model H – 10 Epochs with only 1 feature (ltp) using Bidirectional LSTM first layer

Our results are shown in table 1.

Table 1, Cross-validation of models.

Model	**Forecast MSE/Loss**
A	0.0132
B	0.0138
C	0.0187

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

D	0.0383
E	0.0257
F	0.016
G	0.0153
H	0.017

After evaluating multiple forecasting models, we have determined that Model A exhibits the lowest forecasting error. Therefore, we have chosen Model A as our preferred forecasting model.

6.1.1 Experiment 1: del Potro vs Simon

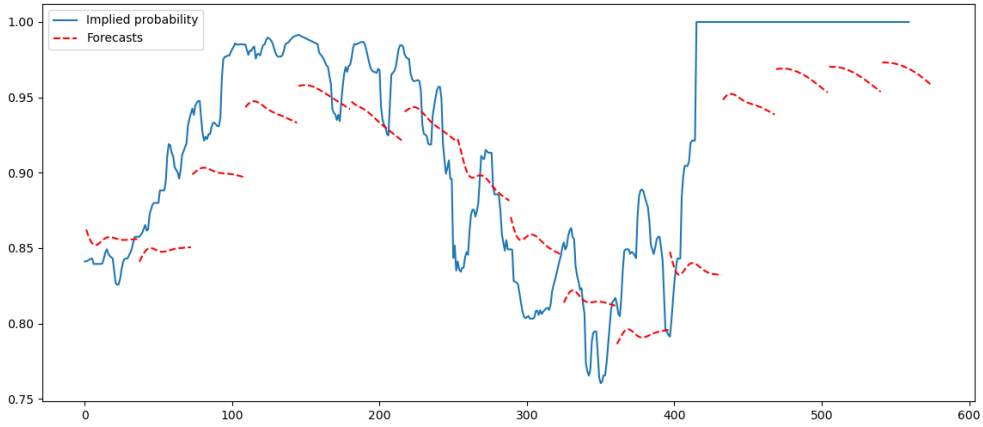

Figure 12, Forecasts of del Potro vs Simon, Wimbledon 2018.

Results for del Potro vs Simon indicate a mediocre prediction fit of our model to the test data. Although we achieved an MSE of 0.00452, there are noticeable issues with the model's performance. First, there is a clear delay compared to the market data, indicating that our model may not be keeping up with real time changes. Moreover, our forecasts show slight variation in direction, indicating that the model may not be extracting complex features from the data, but is merely following the trend.

These results may be an indication of a lack of sufficient sample size available within the match to train the model. Del Potro was able to defeat Simon within 70 minutes in this match, leaving us only 14 minutes of data for initial training. To address these issues, we can attempt to train on a larger sample size of historical matches before fitting the initial data for this match.

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

6.1.2 Experiment 2: Raonic vs Isner

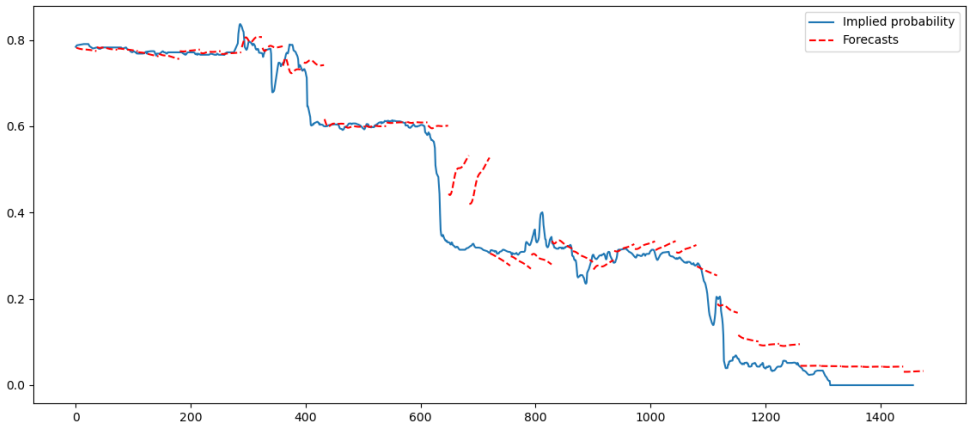

Figure 13, Forecasts of Raonic vs Isner, Wimbledon 2018.

Results for Raonic vs Isner show that our forecasts do indeed match closely with the implied probabilities. While we achieved an MSE of 0.0056, we observed that the LSTM model struggles to accurately predict significant jumps or declines in probabilities resulting from scoring and serve breaks. To enhance our model's predictive capabilities, we can explore the development of a more effective multivariate regression model.

6.2 Market Maker

To Evaluate the Market maker algorithm, a script was created that simulates a Betting exchange using historical data from the Betfair Exchange. The Simulated Market class is a component of the simulation framework for testing and evaluating the market maker algorithm. The class represents a simulated market environment, where the market maker algorithm can interact and make trading decisions based on data. The class has several methods for updating market data, placing orders, and simulating real-time market data. Here is a breakdown of the class and its methods.

The constructor initializes the Simulated Market instance with market definitions, runners, and match data. Market definitions contain general information about the market, such as the market type, name, etc. Runners refer to the potential outcomes in the market, and match data contains the actual match results.

The next method updates the odds of a runner. Depending on the side ("back" or "lay"), it updates the respective odds for a runner. It also updates the total amount traded (tv) and last traded price (ltp) if provided. The next method then prints out the current state of the market. It displays the market name, runners, and their respective back and lay odds.

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

```python
class SimulatedMarket:
    def __init__(self, market_definition, runners,market_data):
        self.market_definition = market_definition
        self.runners = runners
        self.odds = {runner["id"]: {"back": [], "lay": []} for runner in runners}
        self.market_data = []
        self.match_data = match_data

    def update_odds(self, runner_id, odds, side, tv=None, ltp=None):
        self.odds[runner_id][side] = odds
        if tv is not None:
            self.odds[runner_id]["tv"] = tv
        if ltp is not None:
            self.odds[runner_id]["ltp"] = ltp

    def display_market(self):
        print("Market:", self.market_definition["name"])
        for runner in self.runners:
            print(runner["name"])
            print("Back:", self.odds[runner["id"]]["back"])
            print("Lay:", self.odds[runner["id"]]["lay"])
```

Following that is another method that processes a new odds update. For each runner in the update, it updates the back and lay odds, and adds this new snapshot of the market to its history of market data. The method after that calculates the market depth for a runner. It does this by summing up the amounts available for backing and laying at all prices. The next method places an order in the market. Depending on the side, it adds the order to the back or lay side of the order book for the runner. It also ensures that orders are sorted by price, with the best price at the front.

```python
def process_odds_data(self, odds_update):
    snapshot = {"bets": []}
    for item in odds_update:
        runner_id = item["id"]
        if "atb" in item:
            self.update_odds(runner_id, item["atb"], side="back")
        if "atl" in item:
            self.update_odds(runner_id, item["atl"], side="lay")
        snapshot["bets"].append({
            "id": runner_id,
            "odds": self.odds[runner_id],
            "tv": item.get("tv", 0),  # Adding the 'tv' value
            "ltp": item.get("ltp", 0)  # Adding the 'ltp' value
        })
    self.market_data.append(snapshot)


def get_market_depth(self, runner_id):
    market_depth = 0
    for market_data in self.market_data:
        for runner in market_data["bets"]:
            if runner["id"] == runner_id:
                if runner["odds"]["back"]:
                    market_depth += sum([price_data[1] for price_data in runner["odds"]["back"]])
                if runner["odds"]["lay"]:
                    market_depth += sum([price_data[1] for price_data in runner["odds"]["lay"]])
    return market_depth

def place_order(self, runner_id, price, side, order_size):
    if side == "back":
        current_best_back = self.odds[runner_id]["back"][0][0] if self.odds[runner_id]["back"] else 0
        if price > current_best_back:
            self.odds[runner_id]["back"].insert(0, [price, order_size])
        else:
            self.odds[runner_id]["back"].append([price, order_size])
    elif side == "lay":
        current_best_lay = self.odds[runner_id]["lay"][0][0] if self.odds[runner_id]["lay"] else float("inf")
        if price < current_best_lay:
            self.odds[runner_id]["lay"].insert(0, [price, order_size])
        else:
            self.odds[runner_id]["lay"].append([price, order_size])
    return order_size  # Add this line to return the order size
```


GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

Finally, the last method simulates real-time market data. It uses a generator to read a JSON file line by line, processing odds updates and market definition updates as they come. The market maker updates scores and places bets at regular intervals. The market display is updated after each iteration.

In the main script, an instance of Simulated Market is created and used to test the Market Maker algorithm. It loads the market definitions and runners from a JSON file, and match data from a CSV file. The market maker is then run on this simulated market, with its performance metrics calculated at the end.

```
    def simulate_real_time_market(self, market_maker,score_update_interval, json_file, delay=0,sampling_interval = 20):
        json_data_gen = json_file_generator(json_file)
        current_point = 0
        iteration = 1  # Add this line to keep track of the number of iterations

        for i, entry in enumerate(json_data_gen):
            if i % sampling_interval != 0:
                continue
            if 'marketDefinition' in entry['mc'][0]:
                self.market_definition = entry['mc'][0]['marketDefinition']
                self.runners = self.market_definition['runners']

            if 'rc' in entry['mc'][0]:
                self.process_odds_data(entry['mc'][0]['rc'])
                if iteration % score_update_interval == 0:
                    if current_point < len(self.match_data):
                        market_maker.update_scores_from_row(self.match_data.iloc[current_point])
                        current_point += 1
                market_maker.place_bets()
                simple_mm.place_bets()
                print(market_maker.positions)
                print(iteration)
                self.display_market()
                print("\n---\n")
                print(delay)
                time.sleep(delay)
                iteration +=1

# Use the function to load and process the JSON data
market_definitions, runners = load_and_process_json_data("new.json")

match_data = pd.read_csv("match_points.csv")  # Replace with the path to your Excel file
market = SimulatedMarket(market_definitions, runners, match_data)


# Create an instance of the MarketMaker class

market_maker = MarketMaker(market)

simple_mm = SimpleMarketMaker(market)


# Simulate real-time market updates with a 1-second delay between each update
market.simulate_real_time_market(market_maker,48, "new.json", delay=0)
```


GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

```
actual_outcome_id = 2519549  # Change this to the actual player ID of the match outcome
metrics = market_maker.calculate_metrics()
metrics2 = simple_mm.calculate_metrics()
print(metrics)
print(metrics2)

import matplotlib.pyplot as plt

plt.figure(figsize=(10,6))
plt.plot(range(len(market_maker.cumulative_profit)), market_maker.cumulative_profits)
plt.xlabel('Number of Trades')
plt.ylabel('Cumulative Profit')
plt.title('Cumulative Profit Over Time')
plt.show()
plt.savefig("figure1.png")

plt.figure(figsize=(10,6))
plt.hist(market_maker.profits_per_trade, bins=50)
plt.xlabel('Profit')
plt.ylabel('Number of Trades')
plt.title('Histogram of Profits Per Trade')
plt.show()
plt.savefig("figure2.png")

plt.figure(figsize=(10,6))
plt.plot(range(len(market_maker.win_rates)), market_maker.win_rates)
plt.xlabel('Number of Trades')
plt.ylabel('Win Rate')
plt.title('Win Rate Over Time')
plt.show()
plt.savefig("figure3.png")
```

The plots below visualise the metrics used to evaluate the market maker. To help further evaluate the model a simpler Market Maker that does not take into consideration the win probabilities of the players over the course of the match.

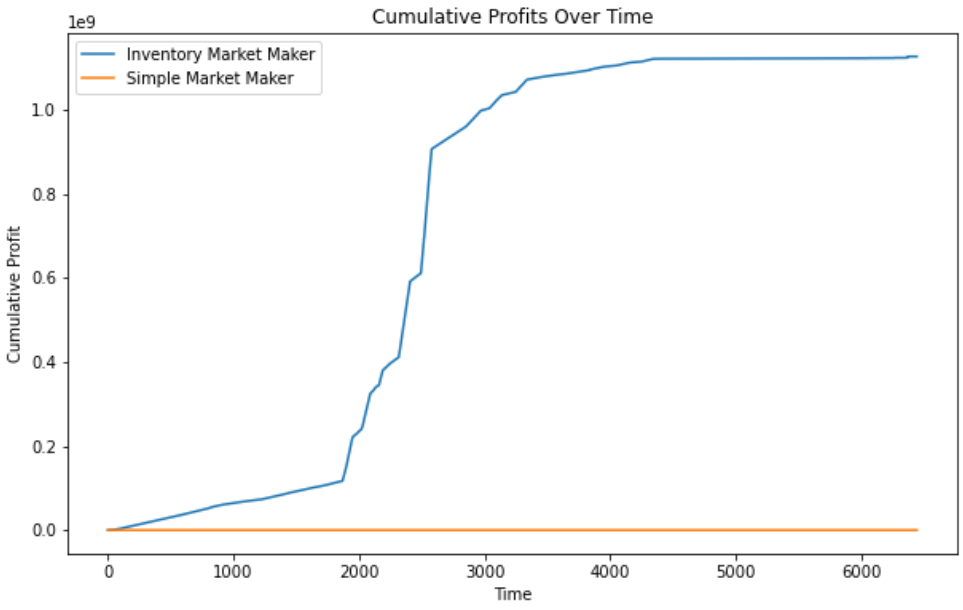

Figure 14, Market maker result

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

The comparative analysis of the cumulative profits between the Market Maker algorithm and the Simple Market Maker algorithm reveals a stark contrast in their performance. The Market Maker algorithm significantly outperforms the Simple Market Maker, which was unable to realize a profit.

Upon examination of the profit graph for the Market Maker, it is apparent that its shape is akin to a logarithmic curve, indicating a rapid initial growth rate which gradually slows over time. This is a typical trait of effective market making strategies where the algorithm capitalizes on market inefficiencies and discrepancies in the initial stages, leading to a surge in profits. As the market matures and inefficiencies diminish, the rate of profit growth decelerates.

These results are a compelling testament to the effectiveness of the Market Maker algorithm. However, it should be noted that this is still a simulated market and may not fully mimic the actual betting exchange for a tennis match.

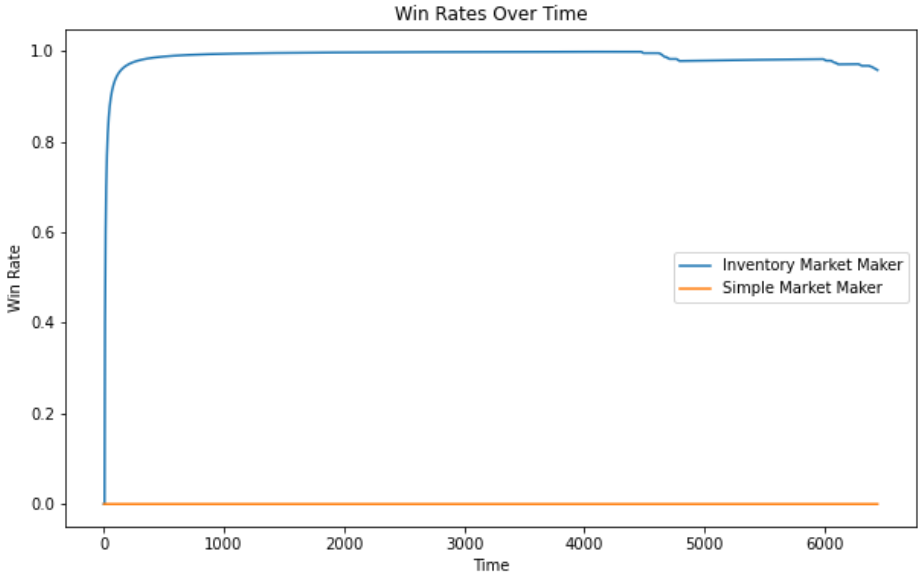

Figure 15, Market maker result

The win rate graph of the Market Maker algorithm presents a unique 'cliff' shape, which is characterized by an immediate surge followed by a plateau. This suggests that the Market Maker quickly ascends to a high success rate early in its operations, and then maintains this elevated level of performance consistently throughout the remaining period.

The initial spike can be attributed to the algorithm's ability to exploit inefficiencies in the market at the onset. The Market Maker's adaptive bid-ask spread, and dynamic order placement strategies allow it to secure profitable trades from the beginning, hence the immediate rise in the win rate.

The plateau phase following the initial spike indicates the algorithm's robust performance consistency. After the initial surge, the win rate stabilizes at a high value, suggesting that the Market Maker continues to win trades at a consistently high rate. The plateau indicates that

the algorithm has found an equilibrium state where it effectively balances risk and reward to maintain a high win rate.

This 'cliff' pattern signifies an efficient market-making strategy. The swift rise and sustained high win rate demonstrate the algorithm's effectiveness in not just identifying and capitalizing on initial market opportunities, but also in adapting to evolving market conditions to consistently secure profitable trades.

In addition to the graphically represented metrics, the simulation has provided a few other important statistical indicators that provide additional insights into the performance of the Market Maker algorithm.

Total Profit: The total profit generated by the Market Maker is approximately 1.127 billion units. This is a substantial return and underscores the effectiveness of the algorithm's trading strategy. The magnitude of this profit demonstrates the algorithm's ability to capitalize on market inefficiencies and generate significant returns, making it a robust tool for high-frequency, automated trading.

Return on Investment (ROI): The ROI, which is the ratio of net profit to the total amount invested, is approximately 3.13. This indicates that for every unit of currency invested, the algorithm returned about 3.13 units. An ROI greater than 1 signifies a profitable investment, and in this case, the ROI of 3.13 indicates a phenomenally successful trading strategy.

Hit Rate: The hit rate, also known as the win rate, is 0.5 or 50%. This means that half of all trades executed by the Market Maker resulted in profit. Given the high-risk, high-reward nature of market making, a hit rate of 50% is commendable. This rate further confirms the algorithm's ability to maintain a balance between risk and reward, effectively managing the inherent risks in high-frequency trading.

These metrics, in conjunction with the graphical data, paint a comprehensive picture of the Market Maker algorithm's performance. Its ability to generate substantial profits, high ROI, and maintain a commendable hit rate, all indicate the effectiveness and efficiency of the algorithm in a simulated trading environment. This makes it a potentially valuable tool for real-world trading operations.

While the results of the simulation are impressive, it is crucial to consider that these outcomes were derived in a controlled, simulated environment. The behaviour of financial or betting markets can be influenced by a multitude of unpredictable factors, including but not limited to large economic shifts, significant news events, and changes in market sentiment. As a result, the Market Maker's actual performance in real-world trading may differ from the results observed in the simulation.

The simulated environment allows for perfect market data and execution, with no slippage or transaction costs, and it ensures that every order placed gets filled. In contrast, real-world markets may have limitations such as latency, slippage, and transaction costs, which can impact the trading algorithm's effectiveness. Moreover, the liquidity of the real-world market may not always be sufficient to fill all orders placed by the algorithm, potentially impacting the profitability.

Additionally, the simulated market lacks the competitive aspect of real-world markets, where multiple market makers and traders are simultaneously operating, potentially impacting market prices and availability.

Therefore, while the simulation provides valuable insights into the potential of the Market Maker algorithm, caution should be exercised when transitioning to real-world markets. It would be advisable to conduct extensive testing and adjustments in a live, low-risk environment before scaling up operations. The algorithm's parameters and strategies may need to be adjusted to account for the unique characteristics and challenges of live markets.

In conclusion, the simulated results offer a promising outlook for the Market Maker algorithm's potential, but these results should be viewed as an indication of potential rather than a guarantee of similar performance in a live market setting.

7 Evaluation and future work

Markov Model

The Markov model has shown considerable potential in predicting market behaviours. However, there are several areas where this approach could be further refined and expanded.

- Data Enrichment: The model could be improved by incorporating additional player-specific and game-specific data. For instance, factors such as player fatigue, injury status, historical performance under similar conditions, or even psychological factors could significantly influence the transition probabilities.
- Incorporating Temporal Dynamics: The current model assumes a static transition matrix. The transition probabilities could vary over time, influenced by numerous factors such as changes in players' form, weather conditions, or game dynamics. Using a time-dependent Markov model could capture these temporal dynamics more effectively.
- Multi-level Hierarchical Model: The hierarchical model could be expanded to more levels, capturing more granular states of the game. For instance, a middle level could be added to represent different game phases or strategies.
- Advanced Machine Learning Techniques: More advanced machine learning techniques could be used to better predict the transition matrix's values. For instance, deep learning models like Recurrent Neural Networks (RNNs) could be used, which can capture long-term dependencies in the data.

LSTM Model

The LSTM model has shown promise at univariate regression and forecasts of market implied probabilities. However, there are a few key areas that could be addressed and improved:

- Multivariate regression: Currently, we are only regressing on one variable to generate forecasts. However, we recognise the potential of enhancing the model with multiple variables and features into the regression analysis.
- Nonmarket features: Our model solely relies on market variables for predictions. Nonmarket features may potentially be more accurate predictors compared to market variables which are affected by inefficiencies and delays.

- Limited sample size: Overall, our transfer learning model trains on a limited amount of 6 samples of match data. Naturally, a greater sample size would allow for a more generalised and representative training, while increasing the robustness of models.

Market maker

The Market Maker algorithm has performed well in simulated environments, but there are several areas for further research and improvement as we move towards real-world markets.

- Adaptive Strategies: The market maker could employ adaptive strategies that change based on the state of the market. For instance, during high volatility periods, the market maker could widen the spread to protect against potential losses.
- Risk Management: More sophisticated risk management techniques could be incorporated to limit potential losses. This could include setting maximum loss thresholds, dynamically adjusting the size of the orders based on the current risk level or employing hedging strategies.
- Incorporating Market Sentiment: The algorithm could potentially benefit from incorporating market sentiment data. Sentiment analysis could be used to predict market movements and adjust the market-making strategy accordingly.
- Real-time Optimization: The parameters of the algorithm could be optimized in real-time based on the performance. Reinforcement learning could potentially be used for this, where the algorithm learns the optimal actions based on the reward (profit) it receives.

In conclusion, while the results so far are promising, there are many potential areas for future work. These improvements could enhance the accuracy and profitability of the Markov model and the Market Maker algorithm, providing a solid foundation for their application in real-world markets.

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

8 Team Contribution

O. Ajomale:

- Development of Markov chain model
- Ensemble model training and integration into Markov model
- Development of market making program

K. Chan:

- Development of LSTM forecasting model
- Development of score inference algorithm
- Developing structure of the project

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

9 References

[1] Xinzhuo Huang, William Knottenbelt, and Jeremy Bradley, “Inferring tennis match progress from in-play betting odds”. Final year project. Imperial College London, South Kensington Campus, London, SW7 2AZ, 2011.

[2] S. Easton and K. Uylangco, “Forecasting outcomes in tennis matches using within-match betting markets,” *International Journal of Forecasting*, vol. 26, no. 3, pp. 564–575, 2010.

[3] Sven Lerner, Dipika Badri, and Kevin Monogue. “DeepTennis: Mid-Match Tennis Predictions CS230- Fall 2019”, Stanford University, 2019.

[4] H. Xu, B. Xu, J. He, and J. Bi, “Deep Transfer Learning based on LSTM model in stock price forecasting,” 5th International Conference on Crowd Science and Engineering, 2021. doi:10.1145/3503181.3503194

[5] Dario Occhipinti. “Betting exchanges: a market maker process,” 2016

[6] K. Krieger, J. L. Davis, and J. Strode, “Patience is a virtue: Exploiting behavior bias in gambling markets,” *Journal of Economics and Finance*, vol. 45, no. 4, pp. 735–750, 2021.

[7] F. J. Klaassen and J. R. Magnus, “Are points in tennis independent and identically distributed? evidence from a dynamic binary panel data model,” *Journal of the American Statistical Association*, vol. 96, no. 454, pp. 500–509, 2001.

[8] D. Jackson and K. Mosurski, “Heavy defeats in tennis: Psychological momentum or random effect?,” *CHANCE*, vol. 10, no. 2, pp. 27–34, 1997.

[9] S. Saxena, “Learn about long short-term memory (LSTM) algorithms,” Analytics Vidhya, https://www.analyticsvidhya.com/blog/2021/03/introduction-to-long-short-term-memory-lstm/ (accessed May 12, 2023).

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

10 Appendices

Appendix A: Markov chain program

```
def game_trans_matrix(ppoint_server):
    ppoint_ret = 1 - ppoint_server
    matrix = np.zeros((17,17))
    tMat1 = pd.DataFrame(data = matrix, index=col_row_names,columns=col_row_names)
    tMat1.at["0-0","15-0"] = ppoint_server
    tMat1.at["15-0","30-0"] = ppoint_server
    tMat1.at["0-15","15-15"] = ppoint_server
    tMat1.at["30-0","40-0"] = ppoint_server
    tMat1.at["15-15","30-15"] = ppoint_server
    tMat1.at["0-30","15-30"] = ppoint_server
    tMat1.at["40-0","Win"] = ppoint_server
    tMat1.at["30-15","40-15"] = ppoint_server
    tMat1.at["40-15","Win"] = ppoint_server
    tMat1.at["40-30(A-40)","Win"] = ppoint_server
    tMat1.at["0-40","15-40"] = ppoint_server
    tMat1.at["15-40","30-40(40-A)"] = ppoint_server
    tMat1.at["30-40(40-A)","30-30(DEUCE)"] = ppoint_server
    tMat1.at["15-30","30-30(DEUCE)"] = ppoint_server
    tMat1.at["30-30(DEUCE)","40-30(A-40)"] = ppoint_server

    tMat1.at["0-0","0-15"] = ppoint_ret
    tMat1.at["15-0","15-15"] = ppoint_ret
    tMat1.at["0-15","0-30"] = ppoint_ret
    tMat1.at["30-0","30-15"] = ppoint_ret
    tMat1.at["15-15","15-30"] = ppoint_ret
    tMat1.at["0-30","0-40"] = ppoint_ret
    tMat1.at["40-0","40-15"] = ppoint_ret
    tMat1.at["30-15","30-30(DEUCE)"] = ppoint_ret
    tMat1.at["40-15","40-30(A-40)"] = ppoint_ret
    tMat1.at["40-30(A-40)","30-30(DEUCE)"] = ppoint_ret
    tMat1.at["0-40","Lose"] = ppoint_ret
    tMat1.at["15-40","Lose"] = ppoint_ret
    tMat1.at["30-40(40-A)","Lose"] = ppoint_ret
    tMat1.at["15-30","15-40"] = ppoint_ret
    tMat1.at["30-30(DEUCE)","30-40(40-A)"] = ppoint_ret


    tMat1.at["Lose","Lose"] = 1
    tMat1.at["Win","Win"] = 1


    return tMat1
```

Game Transition matrix function

```
def prob_game(ppoint_server,s_game):
    matrix = game_trans_matrix(ppoint_server)
    temp = matrix
    for i in range(50):
        matrix = np.dot(matrix,matrix)
    matrix = pd.DataFrame(data = matrix, index=col_row_names,columns=col_row_names)
    probs = np.dot(s_game,matrix)
    return probs
```

Stable state probability function for Game level

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

```
def tie_break(ppoint_srv1, ppoint_srv2):
    states = ["0-0","0-1","1-0","1-1","2-0","0-2","3-0","2-1",
              "1-2","0-3","4-0","3-1",
              "2-2","1-3","0-4","5-0",
              "4-1", "3-2","2-3","1-4",
              "0-5","5-1","4-2","3-3",
              "2-4","1-5","5-2","4-3","3-4",
              "2-5","5-3","4-4","3-5","5-4",
              "4-5", "5-5","6-5","5-6",
              "6-6","SETv1","SETv2","6-0",
              "6-1","6-2","6-3","6-4","4-6",
              "3-6","2-6","1-6","0-6","7-7","7-6","6-7"]
    matrix = np.zeros((54,54))
    tMat2 = pd.DataFrame(data = matrix, index=states,columns=states)
    tMat2.at["0-0","1-0"] = ppoint_srv1
    tMat2.at["3-0","4-0"] = ppoint_srv1
    tMat2.at["2-1","3-1"] = ppoint_srv1
    tMat2.at["1-2","2-2"] = ppoint_srv1
    tMat2.at["0-3","1-3"]  = ppoint_srv1
    tMat2.at["4-0","5-0"] = ppoint_srv1
    tMat2.at["3-1","4-1"] = ppoint_srv1
    tMat2.at["2-2","3-2"] = ppoint_srv1
    tMat2.at["1-3","2-3"] = ppoint_srv1
    tMat2.at["0-4","1-4"] = ppoint_srv1
    tMat2.at["6-1","SETv1"] = ppoint_srv1
    tMat2.at["5-2","6-2"] = ppoint_srv1
    tMat2.at["4-3","5-3"] = ppoint_srv1
    tMat2.at["3-4","4-4"]  = ppoint_srv1
    tMat2.at["2-5","3-5"] = ppoint_srv1
    tMat2.at["1-6","2-6"] = ppoint_srv1
    tMat2.at["6-2","SETv1"] = ppoint_srv1
    tMat2.at["5-3","6-3"] = ppoint_srv1
    tMat2.at["4-4","5-4"] = ppoint_srv1
    tMat2.at["3-5","4-5"]  = ppoint_srv1
    tMat2.at["2-6","3-6"] = ppoint_srv1
    tMat2.at["6-5","SETv1"] = ppoint_srv1
    tMat2.at["5-6","6-6"] = ppoint_srv1
    tMat2.at["6-6","7-6"] = ppoint_srv1

    tMat2.at["0-0","0-1"] = 1 - ppoint_srv1
    tMat2.at["3-0","3-1"] = 1 - ppoint_srv1
    tMat2.at["2-1","2-2"] = 1 - ppoint_srv1
    tMat2.at["1-2","1-3"] = 1 - ppoint_srv1
    tMat2.at["0-3","0-4"]  = 1 - ppoint_srv1
    tMat2.at["4-0","4-1"] = 1 - ppoint_srv1
    tMat2.at["3-1","3-2"] = 1 - ppoint_srv1
    tMat2.at["2-2","2-3"] = 1 - ppoint_srv1
    tMat2.at["1-3","1-4"] = 1 - ppoint_srv1
```

Tie-break transition matrix function

```
def prob_tie(ppoint_srv1,ppoint_srv2,s_tb):
    matrix = tie_break(ppoint_srv1,ppoint_srv2)
    for i in range(1000):
        matrix = np.dot(matrix,matrix)
    matrix = pd.DataFrame(data = matrix, index=col_row_names2,columns=col_row_names2)
    probs = np.dot(s_tb,matrix)
    return probs
```

Stable state probability function for tie-break

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

```
def set(pwin1,pwin2,ptie1):
    matrix = np.zeros((41,41))
    tMat3 = pd.DataFrame(data = matrix, index=col_row_names3,columns=col_row_names3)
    tMat3.at["0-0","1-0"] = pwin1
    tMat3.at["2-0","3-0"] = pwin1
    tMat3.at["1-1","2-1"] = pwin1
    tMat3.at["0-2","1-2"] = pwin1
    tMat3.at["4-0","5-0"]  = pwin1
    tMat3.at["3-1","4-1"] = pwin1
    tMat3.at["2-2","3-2"] = pwin1
    tMat3.at["1-3","2-3"] = pwin1
    tMat3.at["0-4","1-4"] = pwin1
    tMat3.at["5-1","SETv1"] = pwin1
    tMat3.at["4-2","5-2"] = pwin1
    tMat3.at["3-3","4-3"] = pwin1
    tMat3.at["2-4","3-4"] = pwin1
    tMat3.at["1-5","2-5"]  = pwin1
    tMat3.at["5-3","SETv1"] = pwin1
    tMat3.at["4-4","5-4"] = pwin1
    tMat3.at["3-5","4-5"] = pwin1
    tMat3.at["5-5","6-5"] = pwin1

    tMat3.at["0-0","0-1"] = 1 - pwin1
    tMat3.at["2-0","2-1"] = 1 - pwin1
    tMat3.at["1-1","1-2"] = 1 - pwin1
    tMat3.at["0-2","0-3"] = 1 - pwin1
    tMat3.at["4-0","4-1"]  = 1 - pwin1
    tMat3.at["3-1","3-2"] = 1 - pwin1
    tMat3.at["2-2","2-3"] = 1 - pwin1
    tMat3.at["1-3","1-4"] = 1 - pwin1
    tMat3.at["0-4","0-5"] = 1 - pwin1
    tMat3.at["5-1","5-2"] = 1 - pwin1
    tMat3.at["4-2","4-3"] = 1 - pwin1
    tMat3.at["3-3","3-4"] = 1 - pwin1
    tMat3.at["2-4","2-5"] = 1 - pwin1
    tMat3.at["1-5","SETv2"]  = 1 - pwin1
    tMat3.at["5-3","5-4"] = 1 - pwin1
    tMat3.at["4-4","4-5"] = 1 - pwin1
    tMat3.at["3-5","SETv2"] = 1 - pwin1
    tMat3.at["5-5","5-6"] = 1 - pwin1
```

Set level transition matrix function

```
def prob_set(pwin1,pwin2,ptie1,s_set):
    matrix = set(pwin1,pwin2,ptie1)
    for i in range(1000):
        matrix = np.dot(matrix,matrix)
    matrix = pd.DataFrame(data = matrix, index=col_row_names3,columns=col_row_names3)
    probs = np.dot(s_set,matrix)
    return probs
```

Stable state probability function for set level

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

```python
def match(pset_v1):
    pset_v2 = 1-pset_v1
    matrix = np.zeros((10,10))
    tMat4 = pd.DataFrame(data = matrix, index=col_row_names4,columns=col_row_names4)
    tMat4.at["0-0","1-0"] = pset_v1
    tMat4.at["1-0","2-0"] = pset_v1
    tMat4.at["0-1","1-1"] = pset_v1
    tMat4.at["1-1","2-1"] = pset_v1

    tMat4.at["0-0","0-1"] = pset_v2
    tMat4.at["1-0","1-1"] = pset_v2
    tMat4.at["0-1","0-2"] = pset_v2
    tMat4.at["1-1","1-2"] = pset_v2

    # Set stationary states
    tMat4.at["2-0","2-0"] = 1
    tMat4.at["2-1","2-1"] = 1
    tMat4.at["0-2","0-2"] = 1
    tMat4.at["1-2","1-2"] = 1

    tMat4.at["V1","V1"] = 1
    tMat4.at["V2","V2"] = 1
    print(tMat4.shape)
    return tMat4
```

Match level transition level matrix function

```python
def prob_match(pset_v1,s_match):
    matrix = match(pset_v1)
    for i in range(1000):
        matrix = np.dot(matrix,matrix)
    matrix = pd.DataFrame(data = matrix, index=col_row_names4,columns=col_row_names4)
    probs = np.dot(s_match,matrix)
    return probs
```

Steady state probability function for match level

GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/

```
import numpy as np
import pandas as pd

# Constants
global col_row_names, col_row_names2, col_row_names3, col_row_names4
col_row_names = ["0-0", "0-15", "15-0", "15-15", "30-0", "0-30", "40-0",
"30-15", "15-30", "0-40", "40-15", "15-40",
                 "30-30(DEUCE)", "40-30(A-40)", "30-40(40-A)", "HOLD",
"BREAK"]
col_row_names2 = ["0-0", "0-1", "1-0", "1-1",
                  "2-0", "0-2", "3-0", "2-1",
                  "1-2", "0-3", "4-0", "3-1",
                  "2-2", "1-3", "0-4", "5-0",
                  "4-1", "3-2", "2-3", "1-4",
                  "0-5", "5-1", "4-2", "3-3",
                  "2-4", "1-5", "5-2", "4-3", "3-4",
                  "2-5", "5-3", "4-4", "3-5", "5-4",
                  "4-5", "5-5", "6-5", "5-6",
                  "6-6", "SETv1", "SETv2", "6-0",
                  "6-1", "6-2", "6-3", "6-4", "4-6",
                  "3-6", "2-6", "1-6", "0-6", "7-7", "7-6", "6-7"]
col_row_names3 = ["0-0", "0-1", "1-0", "1-1",
                  "2-0", "0-2", "3-0", "2-1",
                  "1-2", "0-3", "4-0", "3-1",
                  "2-2", "1-3", "0-4", "5-0",
                  "4-1", "3-2", "2-3", "1-4",
                  "0-5", "5-1", "4-2", "3-3",
                  "2-4", "1-5", "5-2", "4-3", "3-4",
                  "2-5", "5-3", "4-4", "3-5", "5-4",
                  "4-5", "5-5", "6-5", "5-6",
                  "6-6", "SETv1", "SETv2"]
col_row_names4 = ["0-0", "0-1", "1-0", "1-1", "2-0", "0-2", "2-1", "1-2",
"V1", "V2"]
matrix = np.zeros((1, 17))
game_initial_state = pd.DataFrame(data=matrix, columns=col_row_names)
game_initial_state.at[0, "0-0"] = int(1)
matrix = np.zeros((1, 54))
tb_initial_state = pd.DataFrame(data=matrix, columns=col_row_names2)
tb_initial_state.at[0, "0-0"] = 1
matrix = np.zeros((1, 41))
set_initial_sate = pd.DataFrame(data=matrix, columns=col_row_names3)
set_initial_sate.at[0, "0-0"] = 1
matrix = np.zeros((1, 10))
match_initial_state = pd.DataFrame(data=matrix, columns=col_row_names4)  #
match_initial_state.at[0, "0-0"] = 1


def game_trans_matrix(ppoint_server):
    ppoint_ret = 1 - ppoint_server
    matrix = np.zeros((17, 17))
    tMat1 = pd.DataFrame(data=matrix, index=col_row_names,
columns=col_row_names)
    tMat1.at["0-0", "15-0"] = ppoint_server
    tMat1.at["15-0", "30-0"] = ppoint_server
    tMat1.at["0-15", "15-15"] = ppoint_server
    tMat1.at["30-0", "40-0"] = ppoint_server
    tMat1.at["15-15", "30-15"] = ppoint_server
    tMat1.at["0-30", "15-30"] = ppoint_server
    tMat1.at["40-0", "HOLD"] = ppoint_server
    tMat1.at["30-15", "40-15"] = ppoint_server
    tMat1.at["40-15", "HOLD"] = ppoint_server
```


GitHub repository: https://github.com/dyce135/Algorithmic_Tennis_Betting/