A aerial view of a football field

Description automatically generated



**Jan Vermeerbergen**

**Final Work - Bachelor Toegepaste Informatica**

**Erasmushogeschool Brussel**

**14/06/2024**

**Beating the**

**bookmakers**

**at their own game!**

**A data-science deep dive into predicting**

**football games with machine learning models**

Table of Contents

[Introduction 4](#_Toc168608669)

[1. Methodology 5](#_Toc168608670)

[1.1. Data Collection 5](#_Toc168608671)

[1.2. Types of Data Collected 6](#_Toc168608672)

[1.3. Data Cleaning and Preprocessing 6](#_Toc168608673)

[1.4. Handling Missing Data 7](#_Toc168608674)

[1.5. Data Engineering and Feature selection 7](#_Toc168608675)

[1.5.1. Team Form 7](#_Toc168608676)

[1.5.2. ELO rating 7](#_Toc168608677)

[1.5.3. Average points per game 8](#_Toc168608678)

[1.5.4. Average goals scored/conceded per game 8](#_Toc168608679)

[1.5.5. Rolling averages 9](#_Toc168608680)

[1.5.6. Expected Goals (XG) 9](#_Toc168608681)

[1.5.7. Team vs Opponent metric gaps 9](#_Toc168608682)

[2. Model Development 10](#_Toc168608683)

[2.1. Model 1: XGBoost Classifier 10](#_Toc168608684)

[2.2. Model 2: Gradient Boosting Classifier 10](#_Toc168608685)

[2.3. Model 3: Logistic Regression 11](#_Toc168608686)

[2.4. Model 4: Random Forest Classifier 12](#_Toc168608687)

[2.5. Model 5: Adaboost Classifier 13](#_Toc168608688)

[2.6. Model 6: LGBM Classifier 14](#_Toc168608689)

[2.7. Model 7: MLP Classifier 14](#_Toc168608690)

[2.8. Hyperparameter Tuning 16](#_Toc168608691)

[2.8.1. XGBoost Classifier 17](#_Toc168608692)

[2.8.2. Random Forest Classifier 17](#_Toc168608693)

[2.8.3. Logistic Regression 18](#_Toc168608694)

[2.8.4. Gradient Boosting 18](#_Toc168608695)

[2.8.5. Randomized Search Cross Validation 18](#_Toc168608696)

[3. Ensemble Model Construction 20](#_Toc168608697)

[3.1. Data collection 20](#_Toc168608698)

[3.2. Data processing 20](#_Toc168608699)

[3.3. Training the model 21](#_Toc168608700)

[3.4. Ensemble Techniques 22](#_Toc168608701)

[3.4.1. Voting 22](#_Toc168608702)

[3.4.2. Stacking 23](#_Toc168608703)

[3.4.3. Bagging 23](#_Toc168608704)

[3.4.4. Boosting 24](#_Toc168608705)

[4. Model Evaluation 25](#_Toc168608706)

[4.1. Evaluation Metrics 25](#_Toc168608707)

[4.1.1. Accuracy 25](#_Toc168608708)

[4.1.2. Precision 26](#_Toc168608709)

[4.1.3. Recall 26](#_Toc168608710)

[4.1.4. F1 26](#_Toc168608711)

[4.2. Confusion Matrix 27](#_Toc168608712)

[4.3. Feature Importances 27](#_Toc168608713)

[4.4. Cross Validation 29](#_Toc168608714)

[4.5. Performance 30](#_Toc168608715)

[4.5.1. Performance strategy 30](#_Toc168608716)

[4.5.2. Classification reports 31](#_Toc168608717)

[5. Conclusion 32](#_Toc168608718)

[6. Bibliography 33](#_Toc168608719)

[7. Addendum 34](#_Toc168608720)

[7.1. Data collection code 34](#_Toc168608721)

[7.2. Data engineering code 36](#_Toc168608722)

[7.3. Model training code 46](#_Toc168608723)

Introduction

Predicting the outcomes of European football matches, deciding whether a team wins, draws, or loses, by using data science and machine learning, is a complex task.

For my 'final work' at Erasmushogeschool Brussel, studying applied informatics, I took it upon myself to take a deep dive into this daunting issue, aiming on achieving comparable results than existing models. My initial goal was to produce a model that could achieve a steady average accuracy of at least 75%. Besides the model's performance, as an aspiring data scientist, the main priority of my final work was the AI knowledge acquisition itself.

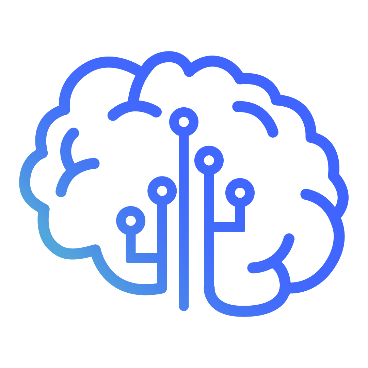
The process of developing a comprehensive model, built in Jupyter Notebook using Python, and performing at a satisfying level towards my preset goals, turned out to be an iterative process of analysing, developing and testing, and proved to be a challenging endeavour at times frustrating and intimidating.

Starting from a baseline model with minimal tuning, and without any feature engineering whatsoever, the progress made by extensive data analysis and thorough feature development was impressive.

The final outcome of my work is an ensemble[[1]](#footnote-1) model, which I named 'football brain', contrived of 7 distinctive classification models, performing in a binary classification strategy, displaying an average accuracy of around 83%, surpassing the expectations of myself and the people I shared my findings with.

An important note to make is that the development of this project will not cease after finishing my degree. The project will continue to be improved in the future. New engineering techniques will be tested, datasets will continue to be analysed, improved and expanded, and the path of moving the whole project to an 'MLOps[[2]](#footnote-2)' environment will be investigated.

Happy to share that during the process of developing the project, interest has grown from an independent investor, who is interested in monetizing the project by packaging it into a marketable online application.



Methodology

Data Collection

During the initial phase of development, the collection of data was a critical task, undertaken by scraping from the football statistics website www.fbref.com. FBref, renowned for its comprehensive tracking of football team and player statistics globally, presented a significant source of valuable data. However, the website did not offer a straightforward method for downloading data in CSV format. Consequently, an intricate web scraper was constructed in Jupyter Notebook, designed to gather match data on a weekly basis. This process involved considerable effort to ensure the accuracy and reliability of the scraped data.

Upon thorough evaluation and deliberation, it became evident that the datasets obtained from FBref.com were insufficient for my needs. The limitations of the data, coupled with the complexities involved in maintaining the scraper, led to the decision to discontinue using these datasets.

A screenshot of a web page

Description automatically generatedIn the search for a more suitable data source, I discovered www.football-data.co.uk, a website that not only mirrored the statistical depth of FBref.com but also offered additional insights. Notably, next to offering ready to download CSV files for competitions of over 25 countries, this alternative source provided bookmakers' odds for both past and upcoming matches. These odds are instrumental in identifying which predictions have the potential to be most profitable, thereby enhancing the predictive models' effectiveness. The availability of such critical data points could significantly influence the accuracy and financial viability of our predictions.

Types of Data Collected

The datasets provided by football-data.co.uk encompass a vast amount of information collected for every match. However, not all of this data is relevant to the project. Therefore, only a selection of the data that is most pertinent to our analysis is used.

Div = League Division

Date = Match Date (dd/mm/yy)

Time = Time of match kick off

HomeTeam = Home Team

AwayTeam = Away Team

FTHG = Full Time Home Team Goals

FTAG = Full Time Away Team Goals

FTR = Full Time Result (H=Home Win, D=Draw, A=Away Win)

HTHG = Half Time Home Team Goals

HTAG = Half Time Away Team Goals

HTR = Half Time Result (H=Home Win, D=Draw, A=Away Win)

HS = Home Team Shots

AS = Away Team Shots

HST = Home Team Shots on Target

AST = Away Team Shots on Target

Data Cleaning and Preprocessing

The ensemble model created in this project is centered around various Gradient Boosting models because of their speed and performance. Internally, gradient boosting models represent all problems as a regression predictive modeling problem that only takes numerical values as input. If the data is in a different form, it must be prepared into the expected format.

Dates are converted to integers through a function, and all categorical data (team names, competion names) are converted to dictionary values so they can be translated back to their original form after training.  
  


Handling Missing Data

The limited subset of data used from the datasets provided by football-data.co.uk has little to no missing information. The only columns with a minimal amount of missing data were HS, AS, HST, and AST. To address these gaps, I employed a technique called imputation, where the empty values in these columns are filled with their respective means. This method ensured that my dataset remained complete and reliable for analysis and training.

Data Engineering and Feature selection

Data engineering is undoubtedly the aspect of my project that demanded the most significant investment of development time. This critical phase involved intricate processes of collecting, cleaning, transforming, and organizing data to ensure its quality and usability for analysis. Given the complexities and challenges associated with managing large datasets in predicting football match outcomes, this stage required meticulous attention to detail, robust problem-solving skills, and a deep understanding of data architectures and tools specific to sports analytics.

From the outset, it was pivotal that the model performed equally well on test data as well as on unseen validation data. Ensuring consistent performance across different datasets was essential for the model’s reliability and accuracy. To achieve this, I deemed it crucial to engineer new features based solely on data available prior to a match starting. This approach prevented any potential data leakage and ensured that the model's predictions were based on realistic and actionable information. By adhering to this principle, I maintained the integrity of the model's performance and provided a robust foundation for its application in real-world scenarios.

Team Form

To evaluate each team's current form, we calculate the average points earned over the past five games. This metric provides a clear, dynamic view of recent performance, highlighting trends in the team's effectiveness. It offers insights into their current momentum and potential future outcomes.

ELO rating

An ELO rating applied to football is a method of ranking teams based on their performance in matches, similar to its original use in chess. This system assigns each team a numerical rating that reflects its overall strength. The ELO rating starts with a base score for all teams and adjusts after each game based on the match's outcome, the ratings of the opponents, and the expected outcome. When a team wins, it gains points, and the losing team loses points. The magnitude of the point change depends on the expected result; beating a higher-rated team yields more points than beating a lower-rated one. Conversely, losing to a lower-rated team results in a more significant point loss. The ELO system is dynamic, updating continuously as teams play more games, which allows it to adapt to changes in team strength over time. By incorporating factors like home-field advantage and goal differentials, the ELO rating provides a nuanced and quantitative measure of team performance, making it a valuable feature for predicting future match outcomes in football.

A table with numbers and names

Description automatically generated

1Calculated ELO rating of teams in the European Top 5 competitions, as of May 20th 2024

Average points per game

To evaluate each team's overall performance in the current season, the average points earned per game is calculated weekly. This metric offers a clear, dynamic view of how each team progresses and performs throughout the season.

Average goals scored/conceded per game

In addition to calculating the average points earned per game in the current season, we also determine the average goals scored and conceded per game. This comprehensive analysis provides a deeper insight into a team's offensive and defensive performance, allowing for a more nuanced understanding of their overall effectiveness on the field. By evaluating both points and goals, we can better assess the strengths and weaknesses of each team throughout the season.History versus an opponent

The weighted points a team earns against another team, calculated over their past five matches, are determined with the most recent match carrying the greatest weight.

Rolling averages

For several columns in the dataset, including HS, AS, HST, and AST[[3]](#footnote-3), rolling averages are calculated over the past five matches. This involves taking the average of a specified number of data points, continuously updated by adding the most recent data point and removing the oldest. By doing so, rolling averages smooth out short-term fluctuations and highlight longer-term trends, providing a clearer view of performance over time.

Expected Goals (XG)

Expected Goals (xG) is a metric used to evaluate the quality of scoring chances for both teams in a match. It estimates the likelihood of a shot resulting in a goal based on factors such as a team’s shots on target and the goals scored in a match. The xG metric provides a more nuanced understanding of a team's offensive effectiveness.

Team vs Opponent metric gaps

During final testing, I discovered that creating additional features to highlight the differences between a team's metrics and those of their opponent, and saving it as a new feature, positively impacted the model's accuracy. These gaps were calculated for key metrics, including ELO, Points, xG, and Form.

The final result is a dataset composed of the aforementioned engineered metrics, supplemented with the average bookmaker's odds for Win/Draw/Lose. During feature selection, only features that positively impacted the model's accuracy were retained, while those with little to no added value were removed. This enriched dataset encapsulates various performance indicators, offering a comprehensive view of each team's strengths, weaknesses, and trends. By integrating these detailed metrics, the dataset serves as a robust foundation for further analysis and predictive modeling.

Model Development

After thorough analysis and experimenting with a variety of model combinations, I ultimately chose to work with seven distinct classification models, as they delivered the most promising results. Interestingly, some models that were anticipated to excel did not meet expectations, while others, unexpectedly, outperformed them.

* 1. Model 1: XGBoost Classifier

In the thrilling world of football match prediction, the XGBoostClassifier has emerged as a powerful tool within ensemble models. XGBoost, short for eXtreme Gradient Boosting, is celebrated for its ability to handle large datasets and deliver high predictive accuracy. Developed by Tianqi Chen, it builds upon the principles of gradient boosting, an ensemble technique that constructs multiple weak learners[[4]](#footnote-4), usually decision trees[[5]](#footnote-5), and combines them to form a strong predictive model.

A diagram of a tree

Description automatically generatedOne of XGBoost’s most compelling strengths is its efficiency and speed. It utilizes advanced regularization techniques to prevent overfitting, ensuring robust performance even with complex data. Furthermore, its scalability allows it to tackle extensive datasets common in football match predictions. This is crucial when considering the myriad of variables such as player statistics, weather conditions, and team strategies.

However, XGBoost is not without its limitations. The model's complexity can be a double-edged sword; it requires careful parameter tuning and can be computationally intensive. Additionally, it demands a significant amount of memory, which can be a drawback for some applications.

In the context of predicting football match outcomes, XGBoostClassifier shines when integrated into an ensemble model. It leverages its strengths to synthesize diverse data points, offering predictions that are both accurate and insightful. Nonetheless, practitioners must be mindful of its resource requirements and the need for meticulous optimization to fully harness its potential. In summary, while XGBoostClassifier is a formidable asset in football analytics, its application requires a blend of computational power and expert tuning.

Model 2: Gradient Boosting Classifier

In the realm of sports analytics, particularly when predicting football match outcomes, the GradientBoostingClassifier (GBC) stands out as a powerful tool. This model, part of an ensemble approach, excels in its ability to enhance predictive accuracy by iteratively correcting errors from weaker models. Its strength lies in its robustness and adaptability, making it particularly effective for complex datasets with diverse features.

However, the GradientBoostingClassifier is not without its drawbacks. Training can be computationally intensive, demanding significant time and resources, which can be a limiting factor for real-time applications. Moreover, it is prone to overfitting, especially with small A diagram of a model boosting

Description automatically generateddatasets, necessitating careful tuning and validation.

In our football match prediction model, GBC plays a crucial role. By leveraging past match data, player statistics, and other relevant factors, it helps create a comprehensive and nuanced predictive framework. This ensemble method, combining multiple GBC models, ensures a higher degree of accuracy and reliability, aiding analysts and enthusiasts alike in making informed predictions.

While the GradientBoostingClassifier is undeniably potent, balancing its strengths and weaknesses through meticulous implementation is key to unlocking its full potential in sports analytics.

Model 3: Logistic Regression

In the realm of predicting football match outcomes, the Logistic Regression model stands as a robust cornerstone within ensemble methods. Originally developed as a binary classifier, Logistic Regression estimates the probability of a categorical outcome, making it particularly adept at scenarios where the result is win or lose, or in some cases, win, lose, or draw.

One of the primary strengths of Logistic Regression lies in its simplicity and interpretability. It offers clear insights into the impact of each feature on the prediction, which is invaluable for understanding the dynamics of football matches. Furthermore, its computational efficiency ensures quick training times, a vital attribute when dealing with large datasets or when integrating real-time data.

A diagram of a error

Description automatically generatedHowever, this model is not without its limitations. Logistic Regression assumes a linear relationship between the independent variables and the log odds of the dependent variable. In the complex world of football, where interactions between variables can be highly non-linear, this assumption can sometimes lead to suboptimal performance. Additionally, Logistic Regression can struggle with multicollinearity[[6]](#footnote-6) and may require careful preprocessing of data to mitigate these effects.

Despite these weaknesses, Logistic Regression's role in ensemble models for predicting football outcomes is significant. When combined with other models, it helps to capture a diverse range of patterns in the data, enhancing overall predictive performance. By leveraging its strengths and compensating for its weaknesses through ensemble techniques, Logistic Regression contributes to more accurate and reliable match predictions, guiding fans and analysts alike in their quest for foresight into the beautiful game.

Model 4: Random Forest Classifier

In the fascinating realm of football match prediction, the RandomForestClassifier shines as a pivotal component of our ensemble model. Originating from the decision tree family, this model excels by constructing multiple decision trees during training and outputting the mode of the classes for classification tasks. The inherent robustness of RandomForestClassifier lies in its ability to handle a vast number of features and its resistance to overfitting. This resilience is achieved through bootstrapping and aggregation, which diversify the trees and enhance generalization.

The strengths of RandomForestClassifier are manifold. It is highly effective with large datasets and complex feature interactions, making it a perfect fit for football match prediction, where numerous variables interplay. Moreover, it provides insights into feature importance, helping us understand which factors significantly influence match outcomes.

A diagram of a tree

Description automatically generatedHowever, the model is not without its weaknesses. It can be computationally intensive and slower to predict compared to simpler models, especially with large forests. Additionally, it might not perform well with data that has little variation or too many irrelevant features, necessitating careful preprocessing.

Despite these limitations, the RandomForestClassifier's performance in our ensemble approach significantly boosts predictive accuracy. By combining it with other models, we harness its strengths while mitigating its weaknesses, leading to a well-rounded, robust predictive system. This amalgamation not only improves match outcome predictions but also provides a deeper understanding of the factors driving these results, ultimately offering a powerful tool for football analytics.

Model 5: Adaboost Classifier

The AdaBoostClassifier, an ensemble learning technique, has proven to be a powerful tool in predicting football match outcomes. This model excels by combining the strengths of multiple weak classifiers to form a robust predictive model. By sequentially focusing on misclassified instances, AdaBoostClassifier effectively reduces bias[[7]](#footnote-7) and variance[[8]](#footnote-8), leading to more accurate predictions. This characteristic is particularly beneficial in the unpredictable domain of sports, where numerous variables influence the results.

One of the key strengths of AdaBoostClassifier lies in its ability to enhance performance through boosting. By iteratively adjusting the weights of misclassified samples, it hones in on the most challenging instances, thereby improving the overall accuracy. Additionally, its flexibility to work with various weak learners makes it adaptable and versatile.

A diagram of a tree

Description automatically generatedHowever, AdaBoostClassifier is not without its weaknesses. It is sensitive to noisy data and outliers, which can lead to overfitting[[9]](#footnote-9). This sensitivity requires careful data preprocessing and sometimes limits its effectiveness in highly variable datasets typical of football matches. Moreover, the performance of AdaBoost can be computationally intensive, especially with large datasets, potentially leading to longer training times.

In the context of football match prediction, the AdaBoostClassifier demonstrates notable promise. Its capacity to refine predictions by addressing previous errors makes it a valuable component of ensemble models designed for this purpose. When combined with other models, it can significantly enhance the accuracy of match outcome predictions, providing valuable insights for analysts and enthusiasts alike.

In summary, while AdaBoostClassifier offers robust performance and adaptability, it must be handled with care due to its sensitivity to noisy data and computational demands. Its strategic use in ensemble models can unlock profound predictive capabilities in the field of football match forecasting.

Model 6: LGBM Classifier

A diagram of a diagram

Description automatically generatedWhen it comes to predicting the outcomes of football matches, the LGBMClassifier (LightGBM Classifier) stands out as a robust choice within ensemble models. Originating from Microsoft’s LightGBM framework, this model leverages gradient boosting techniques to offer superior performance, particularly with large datasets.

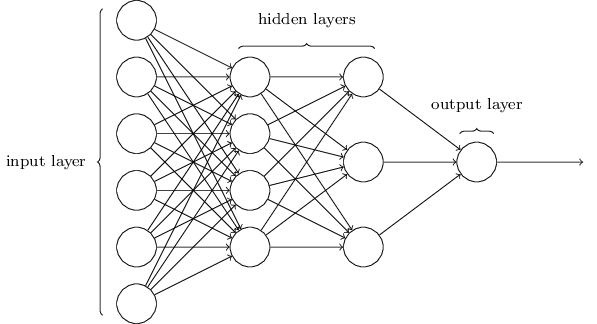
One of the primary strengths of the LGBMClassifier is its speed and efficiency. It outpaces many traditional algorithms thanks to its ability to handle massive data volumes with low memory usage. Additionally, it boasts remarkable accuracy and scalability, making it a favorite for complex predictive tasks like sports outcomes.

However, the LGBMClassifier isn't without its drawbacks. It can be sensitive to overfitting, especially if hyperparameters are not meticulously tuned. Moreover, the model's interpretability is limited compared to simpler algorithms, posing challenges in understanding the rationale behind its predictions.

In ensemble settings, combining the LGBMClassifier with other models can mitigate some of its weaknesses, enhancing overall prediction accuracy. By blending the strengths of multiple algorithms, the ensemble approach provides a more balanced and reliable prediction framework, crucial for the dynamic and unpredictable nature of football matches. Thus, the LGBMClassifier, with its blend of speed and precision, plays a pivotal role in pushing the boundaries of predictive modeling in sports analytics.

Model 7: MLP Classifier

When it comes to predicting football match outcomes, one standout component of our ensemble model is the Multi-Layer Perceptron [[10]](#footnote-10)Classifier (MLPClassifier). This neural network [[11]](#footnote-11)model is particularly adept at capturing non-linear patterns in data due to its architecture, which consists of multiple layers of nodes. Each node, or neuron, in these layers uses a non-linear activation function to learn complex relationships within the input data.

The MLPClassifier excels in scenarios where the relationships between features are intricate and not immediately apparent. Its ability to handle high-dimensional data and learn from a wide array of features makes it invaluable in the dynamic and often unpredictable realm of football match predictions. However, this model is not without its drawbacks. Training an MLPClassifier can be computationally intensive and time-consuming, requiring significant processing power and large datasets to achieve optimal performance. Additionally, it is prone to overfitting, especially if the model is too complex or the training data is not sufficiently large or diverse.

In our ensemble model, the MLPClassifier's strengths are harnessed alongside other algorithms to improve overall predictive accuracy. By combining the MLPClassifier with models that excel in other areas, such as decision trees or logistic regression, we mitigate its weaknesses and capitalize on its unique capabilities. This cooperative approach ensures a more robust and reliable prediction framework, ultimately leading to more accurate football match outcomes.

Thus, the MLPClassifier, with its strengths and occasional limitations, plays a pivotal role in our predictive ensemble, illustrating the power and necessity of diverse algorithmic approaches in machine learning.

Hyperparameter Tuning

In developing the ensemble model, I focused on hyperparameter tuning for four of the seven models: XGBoost, Random Forest, Logistic Regression, and Gradient Boosting. Through extensive testing across various scenarios, this particular configuration emerged as the most effective, consistently yielding the best results.

Hyperparameter tuning is a critical step that can significantly impact the performance of machine learning models. By fine-tuning these parameters, we can enhance the model's accuracy and generalization capabilities. For XGBoost and Gradient Boosting, adjusting parameters like learning rate and max depth helped in capturing complex patterns without overfitting. In the case of Random Forest, parameters such as the number of estimators and max features played a crucial role in balancing bias and variance. Logistic Regression, though simpler, benefited from tuning the regularization strength to prevent overfitting while maintaining interpretability.

Each model contributes uniquely to the ensemble, and optimizing their hyperparameters ensures that we leverage their strengths to the fullest. This approach allows the ensemble to make more robust and reliable predictions. Ultimately, the time and effort invested in this meticulous tuning process paid off, as evidenced by the superior performance of the ensemble model in predicting football match outcomes.



XGBoost Classifier

* **max\_depth:** This sets the maximum depth of each tree. We use values from 1 to 6. Greater depth can capture complex patterns but may lead to overfitting.
* **learning\_rate:** Also known as eta, this parameter controls how much each tree contributes to the model. Values of 0.05, 0.1, and 0.15 help balance accuracy and training speed. Lower rates slow learning for finer adjustments.
* **reg\_lambda:** This is the L2 regularization term, adding a penalty proportional to the square of the weights. We test 0.01 and 0.1. Higher lambda values simplify the model by constraining weights.
* **alpha:** The L1 regularization term adds a penalty equal to the absolute value of weights, promoting sparsity. Values of 0, 0.5, and 1 are tested, with higher values enforcing more sparsity.
* **colsample\_bytree:** This parameter specifies the fraction of features sampled for each tree. We use 0.7 and 0.9 to enhance model robustness by introducing feature diversity.
* **subsample:** This parameter controls the fraction of training data used to fit each tree. Values of 0.75 and 0.85 help reduce overfitting by making the model more robust to data variations.
* **n\_estimators:** This indicates the number of trees in the model. We consider 1, 5, and 10 trees. More trees can improve accuracy but also risk overfitting and increase computational cost.

Random Forest Classifier

* **max\_depth:** This parameter defines the maximum depth of each tree in the forest. We explore values of 1 and 2. A deeper tree can capture more detailed patterns but may overfit the training data. A shallower tree, on the other hand, ensures the model remains simple and generalizes better.
* **min\_samples\_split:** This is the minimum number of samples required to split an internal node. By testing values of 3 and 6, we control the growth of the tree. Higher values prevent the model from learning overly specific patterns, thereby reducing overfitting.
* **min\_samples\_leaf:** This parameter determines the minimum number of samples that a leaf node must have. We consider values of 1 and 3. Setting this parameter helps ensure that leaves have sufficient data, which improves the stability and generalization of the model.
* **n\_estimators:** This denotes the number of trees in the forest. We test 1, 5, and 10 trees. More trees generally lead to better performance, as the model can average out more predictions, but this comes at the cost of increased computational complexity and risk of overfitting.
* **max\_features:** This parameter indicates the number of features to consider when looking for the best split. We use ‘sqrt’ and ‘log2’. ‘Sqrt’ takes the square root of the total number of features, while ‘log2’ takes the logarithm base 2 of the total features. These methods reduce variance and improve the model's robustness.

Logistic Regression

* **C:** This controls regularization strength. Smaller values (0.1) enforce stronger regularization, simplifying the model and reducing overfitting, while larger values (1) make the model more complex by reducing regularization.
* **penalty:** Specifies the norm for penalization. 'l1' promotes sparsity by using absolute values, 'l2' reduces complexity with squared values, and 'elasticnet' combines both, handling correlated features effectively.
* **solver:** The optimization algorithm used. 'saga' supports both L1 and L2 regularization, and is efficient for large, sparse datasets, making it versatile for our needs.
* **l1\_ratio:** Used with 'elasticnet' penalty, it balances L1 and L2 regularization. We set it to 0.5 for an equal mix, allowing fine-tuning between sparsity and smoothness.
* **class\_weight:** Adjusts weights for imbalanced datasets. 'balanced' automatically weights classes inversely proportional to their frequencies, ensuring equal attention to all classes.

Gradient Boosting

* **learning\_rate:** Determines how much each tree contributes to the final model. We test 0.01, 0.1, and 0.15. Lower rates slow learning for finer adjustments, helping to avoid overfitting.
* **n\_estimators:** Indicates the number of boosting stages (trees). We use 1, 5, and 10. More trees can improve accuracy but may increase overfitting and computational cost.
* **max\_depth:** Sets the maximum depth of each tree. We test depths of 3, 5, and 7. Deeper trees capture complex patterns but can overfit if too deep.
* **min\_samples\_split:** Controls the minimum samples required to split a node. We use values of 2 and 5. Higher values prevent splits on noisy data, improving robustness.
* **min\_samples\_leaf:** Defines the minimum samples required at a leaf node. We test 1 and 2. This helps prevent overfitting by ensuring leaves have enough samples.

Randomized Search Cross Validation

A diagram of a randomized search

Description automatically generatedRandomized search is a hyperparameter optimization technique that randomly samples from a specified distribution of parameters. It works by selecting random combinations of hyperparameters from the given distributions, evaluating the performance of each combination, and identifying the best-performing set. I used RandomizedSearchCV to efficiently explore a wide range of hyperparameter combinations for the ensemble model without the exhaustive computation required by grid search. This method randomly samples different parameter values, evaluating each combination to identify the best-performing set. It balances thorough exploration of the hyperparameter space with computational efficiency, making it ideal for finding optimal model settings within a reasonable time frame.



* **estimator:** `ensemble\_clf`  
  The base model or ensemble of models to be optimized.
* **param\_distributions:** `param\_dist`  
  Dictionary specifying the hyperparameters and their possible values to sample from during the search.
* **n\_iter:** `10`  
  Number of different hyperparameter combinations to try.
* **scoring:** `custom\_scorer`  
  Metric(s) used to evaluate the performance of the model during the search.
* **refit:** `'f1\_score'`  
  Metric used to select the best model after the search and refit it on the entire dataset.
* **cv:** `TimeSeriesSplit(n\_splits=10)`  
  Cross-validation strategy, using time series splits to maintain the order of observations.
* **random\_state:** `42`  
  Seed for random number generation to ensure reproducibility of results.
* **n\_jobs:** `-1`  
  Number of CPU cores to use for computation. `-1` means using all available cores.
* **verbose:** `0`  
  Controls the verbosity of the output. `0` means no output, while higher numbers provide more detailed logs.
* **error\_score:** `'raise'`  
  Determines what to do if a model fails to fit. `'raise'` means it will raise an error.

The custom\_scorer function, xgb\_early\_stopping\_score, calculates the F1 score[[12]](#footnote-12) of a model's predictions, facilitating early stopping. Early stopping is a technique where training halts if the model's performance on a validation set doesn't improve after a certain number of iterations. This prevents overfitting, ensuring the model generalizes well to unseen data. Unlike standard scorers like accuracy or F1, early stopping dynamically adjusts the training process, terminating it when optimal performance is reached, thereby improving efficiency and potentially enhancing model performance. This approach is particularly advantageous in iterative algorithms like gradient boosting, where training can be computationally expensive.

Ensemble Model Construction

The journey from raw data collection to making accurate predictions unfolds through a 3-stage process.

Data collection

The initial phase employs a Jupyter Notebook[[13]](#footnote-13), configured for weekly data collection from the extensive repository at football-data.co.uk.

The football-data website offers match data for more than 20 seasons from a wide range of competitions from a large amount of countries, all in csv format. For our analysis, we have honed in on data from the last five seasons, specifically targeting 19 European competitions, focusing on Europe’s most important competitions, supplemented with lower tier competitions where available. This deliberate choice reflects the competitive diversity of European football, offering a dynamic and multifaceted dataset ideal for predictive modeling.

Overview of the competitions used:

**England**

* Premier League (E0)
* Championship (E1)
* League 1 (E2)
* League 2 (E3)

**Scotland**

* Premier League (SC0)
* Division 1 (SC1)

**Germany**

* Bundesliga 1 (D1)
* Bundesliga 2 (D2)

**France**

* Ligue 1 (F1)
* Ligue 2 (F2)

**Italy**

* Serie A (I1)
* Serie B (I2)

**Spain**

* La Liga 1 (SP1)
* La Liga 2 (SP2)

**Belgium**

* Pro League (B1)

**Greece**

* Ethniki Katigoria (G1)

**Netherlands**

* Eredivisie (N1)

**Portugal**

* Primeira Division (P1)

**Turkey**

* Süper Lig (T1)

A computer with a light bulb and various icons

Description automatically generatedData processing

The second phase is the workhorse of the entire project, transforming raw collected data into a polished dataset ready for the training model. This stage involves preprocessing the data collected from football-data.co.uk, cleaning, organizing, and enhancing it to ensure quality and usability.

The data-processing happens in various ways: missing values are imputed, data points are normalized, categorical data is converted into numerical formats suitable for algorithms and new features are engineered uncovering hidden patterns and relationships within the data to boost predictive accuracy. Finally, the processed data is written back to a CSV file, ready to be picked up by the training engine.

In short, this phase refines raw data into a well-structured, insightful dataset, crucial for training an effective and accurate predictive model.

Training the model

With the construction of a new dataset in phase two, we are now poised to feed this data into our ensemble model, enabling it to train and make accurate predictions on football match outcomes. Unlike simpler setups that typically split the full dataset into 80% training and 20% testing, we employ a more sophisticated approach. Recognizing the value of recent match data in reflecting current team performance, we ensure that this data is included in the training set rather than the test set.

A screenshot of a computer screen

Description automatically generated

Our training process utilizes an iterative sliding window approach. This method involves a predetermined window size, currently set to roughly 35% of the total dataset. Each window is divided into a training set and a test set (80/20) and evaluated in every iteration. Subsequently, the window shifts forward by 20% and the training process is repeated. Moreover, the model retrains on data from the previous window where incorrect predictions were made, enhancing its accuracy in subsequent iterations.

To illustrate, our most recent dataset comprises approximately 15,000 matches. With a window size of 5,000 and a step size of 1,000, the sliding window advances 1,000 matches at a time.

Throughout training, the data undergoes a comprehensive pipeline ensuring meticulous preprocessing. This includes handling missing values, encoding categorical variables, scaling features, balancing classes, selecting the most relevant features, and finally fitting the chosen classifier to the prepared data. This rigorous process guarantees that our model is well-equipped to deliver precise and reliable predictions.



**ImbPipeline:** A custom pipeline (from the imbalanced-learn library) designed to handle imbalanced datasets while allowing for easy integration of various preprocessing steps and a classifier.

**'imputer', SimpleImputer(strategy='mean'):** Fills missing values in the dataset with the mean value of the corresponding feature, ensuring no missing data hinders model training.

**'target\_encoder', TargetEncoder():** Encodes categorical features using the target variable, replacing each category with a numerical value based on the mean of the target variable for that category, helping to capture the relationship between categorical features and the target.

**'scaler', StandardScaler():** Standardizes features by removing the mean and scaling to unit variance, ensuring that each feature contributes equally to the model by having similar scales.

**'min\_max\_scaler', MinMaxScaler():** Scales features to a specified range, typically [0, 1], which can be beneficial for models sensitive to the scale of input features, like neural networks.

**'smote', SMOTE(random\_state=random\_state, k\_neighbors=2):** Synthetic Minority Over-sampling Technique (SMOTE) is used to balance the dataset by generating synthetic samples for the minority class, improving model performance on imbalanced datasets.

**'select', SelectKBest(chi2, k=fts):** Selects the top k features based on the Chi-squared statistical test, reducing dimensionality and potentially improving model performance by focusing on the most relevant features. Currently set to 14.

**'clf', clone(base\_estimator):** The classifier to be trained, cloned from a base estimator to ensure a fresh, untrained model instance is used for each training run, preserving the integrity of the original estimator settings.

Ensemble Techniques

Voting

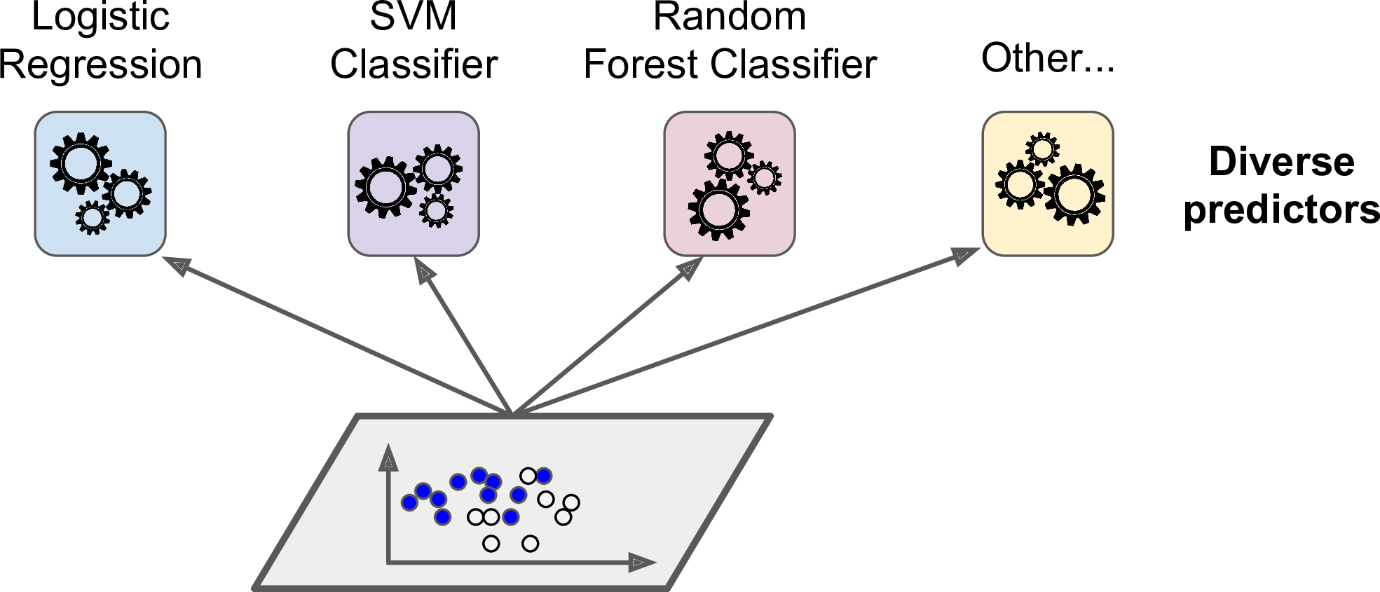
Voting combines the predictions from multiple models by averaging (for regression) or by majority vote (for classification).

**Where Voting is used:**

The VotingClassifier is created to combine predictions from multiple classifiers:



**Analysis:**  
The VotingClassifier combines multiple models and uses a soft voting mechanism, meaning it averages the probabilities predicted by each classifier.



Stacking

Stacking refers to combining multiple models via a meta-model that makes the final prediction based on the outputs of the base models.

**Where Stacking is used:**

A StackingClassifier is not explicitly used in the provided code. While the VotingClassifier is mentioned, it doesn't implement a typical stacking approach.

**Analysis:**  
No explicit stacking is implemented in the code.

A diagram of a model

Description automatically generated

Bagging

Bagging (Bootstrap Aggregating) involves training multiple models on different random subsets of the training data and then combining their predictions.

**Where Bagging is used:**



**Analyis:**

The RandomForestClassifier internally uses bagging by training multiple decision trees on different random subsets of the data and averaging their predictions.

A diagram of steps to a training

Description automatically generated

Boosting

Boosting involves training multiple models sequentially, where each model tries to correct the errors of the previous ones.

**Where Boosting is used:**



**Analysis:**

The GradientBoostingClassifier, AdaBoostClassifier, XGBClassifier, and LGBMClassifier are all boosting algorithms. They train multiple models sequentially, where each new model focuses on the errors made by the previous ones.

A diagram of a process

Description automatically generated

Model Evaluation

Evaluation Metrics

A white background with black text

Description automatically generatedAccuracy

https://www.evidentlyai.com/classification-metrics

In the intricate domain of machine learning, accuracy stands as a pivotal performance metric, encapsulating the proportion of correctly predicted outcomes to the total number of predictions rendered. This metric, while seemingly straightforward, wields considerable power in the initial evaluation of a model's efficacy. For instance, envision a model designed to forecast the results of football matches; accuracy, in this context, would signify the percentage of matches where the model's prediction (whether a team wins, loses, or draws) aligns perfectly with the actual result. However, the utility of accuracy can be deceptive, particularly in scenarios involving imbalanced datasets where certain outcomes disproportionately dominate. Thus, while accuracy provides a foundational understanding, it necessitates supplementary metrics to present a holistic view of a model's true performance.

A white rectangular object with red text

Description automatically generated

2https://www.evidentlyai.com/classification-metrics

Precision

Precision in machine learning is about how often the model is correct when it predicts a specific outcome. For example, if a model predicts that a football team will win, precision tells us how many of those win predictions are actually true positives[[14]](#footnote-14). High precision means the model rarely mistakes a loss or draw for a win, making it very accurate in its win predictions. This is especially important in predicting football match outcomes where getting the exact result right is crucial for accuracy.

A white rectangular object with red text

Description automatically generatedRecall

3https://www.evidentlyai.com/classification-metrics

Recall, in machine learning, is a metric that measures the ability of a model to correctly identify all relevant instances in a dataset. It is defined as the ratio of true positive predictions to the sum of true positive and false negative [[15]](#footnote-15)predictions. High recall means that the model is able to find most of the relevant results. In football terms, if your model predicts the outcomes of matches, recall would indicate how well it correctly identifies all the matches where a team is predicted to win and they actually win.

F1

In machine learning, the F1 score is a crucial metric for evaluating model performance, particularly in classification tasks. It combines precision and recall into a single measure, balancing the trade-off between false positives and false negatives. Specifically, precision measures the accuracy of positive predictions, while recall assesses the model's ability to identify all positive instances. The F1 score is especially valuable in contexts where the cost of false positives and false negatives are significantly different, providing a more nuanced assessment of model effectiveness.

A graph with blue squares and numbers

Description automatically generatedConfusion Matrix

A confusion matrix is a fundamental tool in machine learning used to evaluate the performance of classification algorithms. It presents the outcomes in a matrix format, allowing one to see the true positives, true negatives, false positives, and false negatives. By examining the matrix, you can understand not just the accuracy of your model, but also the types of errors it makes, providing deep insights into its strengths and weaknesses.

In machine learning, the confusion matrix is an indispensable tool for assessing the performance of classification models. By presenting predictions in a matrix format, it delineates the number of true positives, true negatives, false positives, and false negatives. This visualization allows one to interpret not just the model's overall accuracy, but also the specific types of errors, offering a comprehensive understanding of model performance nuances.

**Interpretation of each part of the matrix:**

0 = Home team won

1 = Draw

2 = Away team won

**True Positives (Correct Predictions):**

853 matches where the model correctly predicted the home team would win (0, 0).

219 matches where the model correctly predicted a draw (1, 1).

633 matches where the model correctly predicted the away team would win (2, 2).

**False Positives and False Negatives (Incorrect Predictions):**

130 matches where the model wrongly predicted a home win, but it was a draw (0, 1).

230 matches where the model wrongly predicted a home win, but it was an away win (0, 2).

218 matches where the model wrongly predicted a draw, but the home team won (1, 0).

240 matches where the model wrongly predicted a draw, but the away team won (1, 2).

122 matches where the model wrongly predicted an away win, but the home team won (2, 0).

76 matches where the model wrongly predicted an away win, but it was a draw (2, 1).

Feature Importances

A Feature Importances plot visually represents the significance of each feature in contributing to the predictive power of a model. This plot highlights which variables have the greatest impact, allowing practitioners to interpret and prioritize features accordingly. By examining these importances, one can gain insights into the underlying patterns within the data, potentially leading to more informed decisions and refined models.

A graph of blue bars

Description automatically generated with medium confidence

From the plot, we observe that 'elo\_diff' stands out as the most influential feature. This metric, which represents the difference in Elo ratings between the competing teams, underscores the predictive value of historical performance and relative strength. Following closely are 'team\_shots' and 'opp\_shots\_target,' emphasizing that the number of shots taken by the team and the accuracy of the opponent's shots on target are critical factors in determining match results.

Interestingly, 'team\_elo' and 'opp\_elo' also hold substantial importance, reinforcing the significance of Elo ratings but from individual perspectives of the teams involved. This suggests that not only the difference in strength matters, but the absolute strength of each team plays a pivotal role.

Further down the list, we see 'team\_points\_diff' and 'points\_diff,' which likely represent the difference in points earned by the teams over a season or a series of matches. This feature is crucial as it reflects the recent form and consistency of the teams.

The presence of features such as 'team\_hist\_vs' (which denotes historical performance against specific opponents) and 'form\_diff' (possibly indicating recent form differences) provides additional layers of context, allowing the model to make nuanced predictions based on a variety of performance indicators.

In summary, this feature importance plot serves as a roadmap for understanding which aspects of team performance and historical data are most influential in predicting match outcomes. By focusing on the top-ranked features, one can gain insights into the critical factors that drive football match predictions, enabling more informed decision-making and strategic planning.

Cross Validation

Cross-validation is a critical technique in machine learning used to assess how well a model will generalize to an independent dataset. It involves partitioning the dataset into multiple subsets, training the model on some subsets while validating it on the remaining ones, and then averaging the results to reduce variability. This process ensures that the model's performance is consistent and reliable, avoiding the risk of overfitting to a particular subset of the data.

In the project, cross-validation is prominently implemented using the TimeSeriesSplit method within the RandomizedSearchCV function. Here’s where and how cross-validation is applied:

**Time Series Cross-Validation:**

The cross-validation strategy used in my project is TimeSeriesSplit, which is specifically designed for time series data. This method is critical in time series forecasting as it respects the temporal ordering of data, ensuring that the training data is always from the past and the validation data is from the future, thus preventing data leakage.

A graph with red and green dots

Description automatically generated



In this code snippet, TimeSeriesSplit(n\_splits=200) indicates that the dataset is split into 200 folds. This method trains the model on a growing window of data and validates it on the subsequent data points, thereby simulating a real-world scenario where predictions are made on future unseen data based on past observations.

**Randomized Search with Cross-Validation:**

RandomizedSearchCV is used to tune hyperparameters by randomly sampling from the parameter grid. It integrates cross-validation to evaluate each set of hyperparameters.



During the fit [[16]](#footnote-16)method, the cross-validation process takes place. For each set of hyperparameters, the model is trained and validated across multiple time splits. The performance metrics from each fold are averaged to determine the best hyperparameters.

Performance

Performance strategy

The performance of our ensemble model, crafted to forecast football match outcomes, becomes particularly fascinating when viewed from a betting or bookmaker's perspective. Rather than adhering to traditional accuracy metrics, our evaluation hinges on profitability—a more nuanced and financially driven measure. To this end, we employ a 'Double Chance' betting strategy, which involves placing wagers on two out of the three possible outcomes (1X, X2, or 12). This approach inherently increases the likelihood of a favorable outcome, offering a strategic edge in the betting arena.

In tandem with this betting method, our strategy is meticulously selective, pinpointing matches where the bookmakers' odds significantly underestimate the probabilities projected by our model. These opportunities, which we label as 'value bets,' are the cornerstone of our profitability strategy. By identifying and capitalizing on these discrepancies, we position ourselves to secure bets with optimal returns.

This careful orchestration has yielded noteworthy results. On the validation set, the model's accuracy has impressively climbed to an average of approximately 82%. This substantial increase underscores the potency of our dual approach, which marries advanced predictive modeling with sharp betting practices. It's not merely about predicting match outcomes; it's about transforming these predictions into profitable betting decisions, and ultimately 'beating the bookmakers at their own game'.

In summary, the ensemble model's performance, when integrated with a sophisticated betting strategy, reveals a promising landscape for profitability. The essence of our success lies in our ability to identify and act upon value bets, leveraging the model's insights to outwit traditional bookmaker odds. This refined and dynamic approach not only enhances our accuracy but also ensures a consistent and profitable betting strategy.

A table with numbers and letters

Description automatically generated

4Example matches identified as value bets

Classification reports

Comparing the classification reports for the validation and test sets reveals several encouraging signs of the model's performance. Firstly, the accuracy improves from 55% on the validation set to 62% on the test set, showcasing the model's impressive ability to generalize well to new data. Notably, class 0 (=home team win) consistently achieves high precision and recall across both sets, affirming the model’s reliability in identifying this class accurately. Additionally, class 2 (=away team win) demonstrates a remarkable recall of 74% in both sets, indicating a very strong performance in this category.

A screenshot of a report

Description automatically generatedA screenshot of a test

Description automatically generatedWhile class 1 (= draw) initially presents challenges with lower precision and recall on the validation set, it is heartening to see improvements in the test set, where precision increases to 51% and recall to 32%. This upward trend suggests that with further refinement, the model’s performance for this class can continue to improve. The macro and weighted averages also see enhancements in the test set, highlighting a more balanced and robust overall performance.

In summary, the model exhibits strong generalization capabilities, particularly excelling in classes 0 and 2, and shows potential for further improvement in class 1. These positive trends suggest that with ongoing optimization, the model can achieve even greater accuracy and reliability.

Conclusion

Embarking on the journey to predict football match outcomes has been a profoundly enlightening experience for me. My project culminated in an ensemble model, which I named 'Football Brain,' achieving an impressive 83% accuracy. While I struggled initially with the data scraped from FBref.com, switching to football-data.co.uk, which included bookmakers' odds, transformed my approach.

Feature engineering was pivotal, and made all the difference. Metrics like ELO ratings, team form, and average goals scored/conceded were key. Integrating seven classifiers, including XGBoost and Random Forest, allowed me to harness the strengths of diverse algorithms, significantly boosting predictive performance.

Hyperparameter tuning demanded precision, and cross-validation using TimeSeriesSplit ensured robustness. I employed a 'Double Chance' betting strategy, focusing on profitability. Identifying 'value bets,' where predictions diverged from bookmakers' odds, proved particularly profitable.

Next to sharpening my Python / Jupyter skills, this project taught me invaluable lessons in data collection, feature engineering, and model development. The practical application of advanced algorithms and strategic betting underscored the model's utility, not just in predicting results but in generating profitable betting strategies. The investor interest I garnered hints at future commercialization.

Ultimately, this work exemplifies how combining rigorous data processing, sophisticated algorithms, and strategic application can transform data into actionable insights, promising exciting advancements in sports analytics. This experience has significantly broadened my knowledge, and has seriously increased my interest in the field of data science.

Thank you

A blue scribbles on a white surface

Description automatically generatedJan Vermeerbergen  
05/06/2024

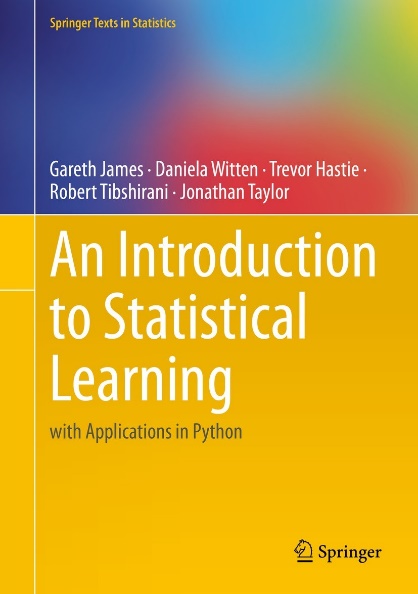
A book cover with a lizard

Description automatically generatedBibliography

Géron, A. (2023). Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow (Third Edition). Sebastopol, Ca USA: O’Reilly

James G., Witten D., Hastie T., Tibshirani R., Taylor J. (2023). An introduction to Statistical Learning with Applications in Python (First printing). Springer

Evaluating actions in football using machine learning  
<https://soccermatics.medium.com/evaluating-actions-in-football-using-machine-learning-69517e376e0c>

Fifa Match Predictor  
*Using Python and machine learning to create a foundation for soccer match predictions using player statistics*  
<https://medium.com/fifa-match-predictor/fifa-match-predictor-10e9985f338a>

Predicting EPL Football Match Winners Using Machine Learning  
<https://app.dataquest.io/m/99992/portfolio-project%3A-predicting-epl-football-match-winners-using-machine-learning/1/project-overview>

Predicting the Winning Team with Machine Learning  
<https://www.youtube.com/watch?v=6tQhoUuQrOw>

In Depth: Parameter tuning for Random Forest  
<https://medium.com/all-things-ai/in-depth-parameter-tuning-for-random-forest-d67bb7e920d>

How to train XGBoost models in Python  
<https://www.youtube.com/watch?v=aLOQD66Sj0g>

Pandas documentation  
<https://pandas.pydata.org/pandas-docs/stable/index.html>

8 Simple Techniques to Prevent Overfitting  
<https://towardsdatascience.com/8-simple-techniques-to-prevent-overfitting-4d443da2ef7d>

A-Z Machine Learning: Random Forest in Time Series Analysis  
*A Deep Understanding and Visualization of Random Forest*  
<https://medium.com/@kaabar-sofien/a-z-machine-learning-random-forest-in-time-series-analysis-28f3ac185666>

A New Horizon in Football Forecasting: The Betting Odds-Based ELO System  
<https://medium.com/@marin11amf11/a-new-horizon-in-football-forecasting-the-betting-odds-based-elo-system-f5032018f546>

Advanced Machine Learning: Enhancing Models with Hyperparameter Tuning  
*Data Science for Everyone — Part 7*  
<https://medium.com/datadriveninvestor/advanced-machine-learning-enhancing-models-with-hyperparameter-tuning-part-7-ba67299cb4dd>

Addendum

All versioned code is available at <https://github.com/aftermathematic/football_brain>

Below is a printout of the 3 phases as how the project is conceived:

* 1: Data collection
* 2: Data engineering
* 3: Model training

Data collection code

# %%

# Standard library imports

import os

import warnings

import requests

import time

# Miscellaneous settings

%matplotlib inline

warnings.filterwarnings('ignore')

# %%

comps = [

    'E0', 'E1', 'E2', 'E3',

    'SC0', 'SC1',

    'D1', 'D2',

    'F1', 'F2',

    'I1', 'I2',

    'SP1', 'SP2',

    'B1',

    'G1',

    'N1',

    'P1',

    'T1',

]

seasons = [

    '2324',

    '2223', '2122', '2021',

    #'1920', '1819', '1718', '1617',

    #'1516', '1415', '1314', '1213',

    #'1112', '1011',

    #'0910', '0809',

    #'0708', '0607', '0506', '0405',

    #'0304', '0203', '0102', '0001',

]

countries = [

    "ARG", "AUT", "BRA", "CHN",

    "DNK", "FIN", "IRL", "JPN",

    "MEX", "NOR", "POL", "ROU",

    "RUS", "SWE", "SWZ", "USA",

]

fixtures = [

    "fixtures",

    "new\_league\_fixtures"

]

# %%

# DOWNLOAD COMPETITION DATA

# Base URL

base\_url = 'https://www.football-data.co.uk/mmz4281/{}/{}.csv'

# Iterate over seasons and competition codes

for season in seasons:

    for comp in comps:

        # Construct file URL

        file\_url = base\_url.format(season, comp)

        # Set the path where the file will be saved

        save\_path = f'data/scraped/{season}/{comp}.csv'

        # Ensure the directory exists

        os.makedirs(os.path.dirname(save\_path), exist\_ok=True)

        try:

            # Download the file

            response = requests.get(file\_url)

            # Check if the response was successful

            if response.status\_code == 200:

                # Write the content to the file, overwriting if it exists

                with open(save\_path, 'wb') as file:

                    file.write(response.content)

                print(f'Successfully downloaded and saved: {save\_path}')

            else:

                print(f'Failed to download {file\_url}. Status code: {response.status\_code}')

            # Wait for 1 second to avoid overwhelming the server

            #time.sleep(1)

        except Exception as e:

            print(f'An error occurred while downloading {file\_url}: {e}')

# %%

# DOWNLOAD COUNTRY DATA

# Base URL

base\_url = 'https://www.football-data.co.uk/new/{}.csv'

for country in countries:

    # Construct file URL

    file\_url = base\_url.format(country)

    # Set the path where the file will be saved

    save\_path = f'data/scraped/other/{country}.csv'

    # Ensure the directory exists

    os.makedirs(os.path.dirname(save\_path), exist\_ok=True)

    try:

        # Download the file

        response = requests.get(file\_url)

        # Check if the response was successful

        if response.status\_code == 200:

            # Write the content to the file, overwriting if it exists

            with open(save\_path, 'wb') as file:

                file.write(response.content)

            print(f'Successfully downloaded and saved: {save\_path}')

        else:

            print(f'Failed to download {file\_url}. Status code: {response.status\_code}')

        # Wait for 1 second to avoid overwhelming the server

        #time.sleep(1)

    except Exception as e:

        print(f'An error occurred while downloading {file\_url}: {e}')

# %%

# DOWNLOAD FIXTURE DATA

# Base URL

base\_url = 'https://www.football-data.co.uk/{}.csv'

for fixture in fixtures:

    # Construct file URL

    file\_url = base\_url.format(fixture)

    print(file\_url)

    # Set the path where the file will be saved

    save\_path = f'data/fixtures/{fixture}.csv'

    # Ensure the directory exists

    os.makedirs(os.path.dirname(save\_path), exist\_ok=True)

    try:

        # Download the file

        response = requests.get(file\_url)

        # Check if the response was successful

        if response.status\_code == 200:

            # Write the content to the file, overwriting if it exists

            with open(save\_path, 'wb') as file:

                file.write(response.content)

            print(f'Successfully downloaded and saved: {save\_path}')

        else:

            print(f'Failed to download {file\_url}. Status code: {response.status\_code}')

        # Wait for 1 second to avoid overwhelming the server

        #time.sleep(1)

    except Exception as e:

        print(f'An error occurred while downloading {file\_url}: {e}')

Data engineering code

# %%

# Standard library imports

import os

import sys

import re

import warnings

import random

import hashlib

# Data manipulation and analysis

import numpy as np

import pandas as pd

# Visualization libraries

import matplotlib.pyplot as plt

import seaborn as sns

# Machine learning and preprocessing

from sklearn.metrics import confusion\_matrix, classification\_report, precision\_score

from sklearn.model\_selection import RandomizedSearchCV, TimeSeriesSplit

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import StandardScaler

# Specific models and tools

from xgboost import XGBClassifier

import xgboost as xgb

# Encoding and feature selection

from category\_encoders import TargetEncoder

from scipy.stats import randint, uniform

# Model persistence

from joblib import dump, load

# Miscellaneous settings

%matplotlib inline

warnings.filterwarnings('ignore')

# %%

comps = [

    'E0', 'E1', 'E2', 'E3',

    'SC0', 'SC1',

    'D1', 'D2',

    'F1', 'F2',

    'I1', 'I2',

    'SP1', 'SP2',

    'B1', 'G1', 'N1', 'P1', 'T1',

]

seasons = [

    '2324', '2223', '2122',

    #'2021', '1920', '1819', '1718', '1617',

    #'1516', '1415', '1314', '1213', '1112', '1011', '0910',

    #'0809', '0708', '0607', '0506', '0405',

]

fixtures = [

    'fixtures',

    # 'new\_league\_fixtures'

]

# %%

# Set the dataprep\_start\_date to the date the data preparation should start

# If None, the data preparation will start from the beginning of the data

# Make sure the file below already exists if you want to start from a specific date

# file should be in the format "processed\_data\_<content>.csv"

content = "allcomps\_3s\_june2024"

dataprep\_start\_date = None

#dataprep\_start\_date = pd.Timestamp(year=2024, month=5, day=1)

# %%

matches\_files = []

fixtures\_files = []

# %%

for season in seasons:

    for comp in comps:

        matches\_files.append('data/scraped/%s/%s.csv' % (season, comp))

        continue

# %%

for fixture in fixtures:

    fixtures\_files.append(f'data/fixtures/{fixture}.csv')

    continue

# %%

fixtures\_files

# %%

# Function to load data from multiple files into a single DataFrame

def load\_data(files):

    df = pd.DataFrame()

    for file in files:

        try:

            print(f'Loading {file}')

            # Try to read with default utf-8 encoding

            try:

                df\_temp = pd.read\_csv(file, encoding='utf-8')

            except UnicodeDecodeError:

                # If utf-8 decoding fails, try reading with ISO-8859-1

                df\_temp = pd.read\_csv(file, encoding='ISO-8859-1')

            match = re.search(r'(\d{4})', file)

            if match:

                year = match.group(1)

                df\_temp['Season'] = year

            else:

                # If the file does not contain the FTR column, it is a fixture file

                df\_temp['Season'] = seasons[0]

            df = pd.concat([df, df\_temp], ignore\_index=True)

        except FileNotFoundError:

            print(f'Error: {file} not found')

        except Exception as e:

            print(f"An error occurred while loading {file}: {e}")

    return df

# %%

# Load data into DataFrames

df = load\_data(matches\_files)

df\_fixtures = load\_data(fixtures\_files)

# %%

len(df), len(df\_fixtures)

# %%

def parse\_date\_to\_int(date\_str):

    # Split the date\_str by the "/" character into day, month, year

    components = date\_str.split('/')

    # If split was successful but not in expected format, try splitting by absence of separator for '%d%m%Y' or '%d%m%y'

    if len(components) == 1:

        if len(date\_str) in [6, 8]:  # Length 6 for '%d%m%y', 8 for '%d%m%Y'

            day, month = int(date\_str[:2]), int(date\_str[2:4])

            year = int(date\_str[4:])

        else:

            return 19000101  # Return default if format does not match expected

    else:

        day, month = int(components[0]), int(components[1])

        year = int(components[2])

    # Adjust the year if it was only 2 characters long

    if year < 100:

        year += 2000

    # Create a date variable by using the day, month, year integers

    # Note: Direct creation of date variable skipped to avoid unnecessary complexity,

    # directly formatting to YYYYMMDD integer format instead.

    date\_int = int(f"{year:04d}{month:02d}{day:02d}")

    return date\_int

# %%

if len(df\_fixtures) > 0:

    # Parse the 'Date' column to a datetime object

    df\_fixtures['Date\_temp'] = pd.to\_datetime(df\_fixtures['Date'], format='%d/%m/%Y')

    # Convert the datetime object to an integer in the format YYYYMMDD

    df\_fixtures['Date\_temp'] = df\_fixtures['Date\_temp'].apply(

        lambda x: int(x.strftime('%Y%m%d')) if pd.notnull(x) else 19000101)

    # Replace all values with -1 in FTR column

    df\_fixtures['FTR'].fillna('X', inplace=True)

    # Find the lowest fixture date

    # This is the date where the data preparation will start

    fixture\_cutoff = df\_fixtures['Date'].min()

    # Remove all the rows in df that are after the fixture\_cutoff date

    #df = df[df['Date'] < fixture\_cutoff]

    # Concatenate the matches and fixtures dataframes

    df = pd.concat([df, df\_fixtures], ignore\_index=True)

# %%

len(df), len(df\_fixtures)

# %%

# Check for duplicate column names

print(df.columns[df.columns.duplicated()])

# %%

# Remove all the rows in the dataframe where the 'Div' is not in the list of comps

df = df[df['Div'].isin(comps)]

# %%

# Create a dictionary for all competitions

file\_path = f"data/comps\_dict\_{content}.txt"

# Check if the file exists

if os.path.exists(file\_path):

    # Load the dictionary from the file

    with open(file\_path, 'r') as file:

        comps\_dict = eval(file.read())  # Using eval to convert string back to dictionary

    # Find the maximum index currently in the dictionary

    max\_index = max(comps\_dict.values())

    print(f"max index: {max\_index}")

else:

    comps\_dict = {}

    max\_index = -1

# Get all unique divisions from DataFrame

all\_comps = df['Div'].dropna().unique()

all\_comps.sort()

# Create a dictionary of new divisions alone

new\_comps = {div: index for index, div in enumerate(all\_comps, start=max\_index + 1) if div not in comps\_dict}

# Update dictionary only with new divisions

comps\_dict.update(new\_comps)

# Save the updated dictionary to a file

with open(file\_path, 'w') as file:

    file.write(str(comps\_dict))

# Add division ID column to DataFrame

df['Div'] = df['Div'].map(comps\_dict)

# %%

# Create a dictionary for all teams

file\_path = f"data/teams\_dict\_{content}.txt"

# Check if the file exists

if os.path.exists(file\_path):

    # Load the dictionary from the file

    with open(file\_path, 'r') as file:

        teams\_dict = eval(file.read())

    max\_index = max(teams\_dict.values())

    print(f"max index: {max\_index}")

else:

    teams\_dict = {}

    max\_index = -1

# Get all teams from DataFrame

all\_teams = pd.concat([df['HomeTeam'], df['AwayTeam']]).dropna().unique()

all\_teams.sort()

# Create a dictionary of new teams alone

new\_teams = {team: index for index, team in enumerate(all\_teams) if team not in teams\_dict}

# Update dictionary only with new teams, starting indices from max\_index + 1

start\_index = max\_index + 1

teams\_dict.update({team: index + start\_index for index, team in enumerate(new\_teams) if team not in teams\_dict})

# Save the updated dictionary to a file

with open(file\_path, 'w') as file:

    file.write(str(teams\_dict))

# Add team ID columns to DataFrame

df['Team\_ID'] = df['HomeTeam'].map(teams\_dict)

df['Opp\_ID'] = df['AwayTeam'].map(teams\_dict)

# %%

def clean\_duplicates(df):

    # Sort the DataFrame so that rows with 'FTR' == -1 come first

    df.sort\_values(by=['Date', 'Team\_ID', 'Opp\_ID', 'FTR'], ascending=[True, True, True, False], inplace=True)

    # Drop duplicates based on 'Date', 'Team\_ID', and 'Opp\_ID' keeping the first occurrence (where 'FTR' is -1)

    df = df.drop\_duplicates(subset=['Date', 'Team\_ID', 'Opp\_ID'], keep='first')

    return df

df = clean\_duplicates(df)

# %% [markdown]

# ### Feature Engineering

# %%

# Calculate ELO ratings for each team

# Initialize ratings dictionary

teams = pd.concat([df['Team\_ID'], df['Opp\_ID']]).unique()

ratings = {team: 1500 for team in teams}

def calculate\_expected\_score(rating\_a, rating\_b):

    return 1 / (1 + 10 \*\* ((rating\_b - rating\_a) / 400))

def update\_elo(rating, actual\_score, expected\_score, k=30):

    rating = rating + k \* (actual\_score - expected\_score)

    # Parse the rating as an integer with no decimal points

    return int(rating)

# Iterate over the DataFrame and update ELO ratings after each match

elo\_team = []

elo\_opp = []

for index, row in df.iterrows():

    home\_team, away\_team, home\_score, away\_score = row['Team\_ID'], row['Opp\_ID'], row['FTHG'], row['FTAG']

    home\_rating = ratings[home\_team]

    away\_rating = ratings[away\_team]

    # Calculate expected scores

    expected\_home = calculate\_expected\_score(home\_rating, away\_rating)

    expected\_away = calculate\_expected\_score(away\_rating, home\_rating)

    # Calculate actual scores

    actual\_home = 1 if home\_score > away\_score else 0.5 if home\_score == away\_score else 0

    actual\_away = 1 - actual\_home

    # Update ratings

    new\_home\_rating = update\_elo(home\_rating, actual\_home, expected\_home)

    new\_away\_rating = update\_elo(away\_rating, actual\_away, expected\_away)

    # Store updated ratings in the ratings dictionary

    ratings[home\_team] = new\_home\_rating

    ratings[away\_team] = new\_away\_rating

    # Append current ratings to list

    elo\_team.append(new\_home\_rating)

    elo\_opp.append(new\_away\_rating)

# %%

# Assign new ELO ratings to the DataFrame

df['team\_elo'] = elo\_team

df['opp\_elo'] = elo\_opp

# %%

df['Date'] = pd.to\_datetime(df['Date'], errors='coerce', dayfirst=True)

# Apply the modified function

df['Date\_temp'] = df['Date'].apply(lambda x: parse\_date\_to\_int(x.strftime('%d/%m/%Y')) if pd.notnull(x) else 19000101)

# Day of the week as an integer

df['DayOTW'] = df['Date'].dt.dayofweek

df['Time'] = df['Time'].fillna('00:00').str.replace(':', '').astype(int)

# Only keep the first 2 digits of the Time column, no decimals

df['Time'] = df['Time'] // 100

# Sort df by Date\_temp and Time

df = df.sort\_values(['Date\_temp', 'Time'])

# %%

df.columns = [re.sub(r'[<]', '\_st\_', str(col)) for col in df.columns]

df.columns = [re.sub(r'[>]', '\_gt\_', str(col)) for col in df.columns]

# %%

# Calculate the average points earend by a team in the current season

def points(df, row, team\_column, last\_n\_matches=None):

    # Season and date of the current match

    current\_season = row['Season']

    current\_date = row['Date']

    # Define the opponent column based on the team column

    opponent\_column = 'Opp\_ID' if team\_column == 'Team\_ID' else 'Team\_ID'

    # Filter DataFrame for matches from the same season before the current date

    past\_matches = df[

        (df['Season'] == current\_season) &

        (df['Date'] < current\_date) &

        ((df[team\_column] == row[team\_column]) | (df[opponent\_column] == row[team\_column]))

    ].copy()

    # If last\_n\_matches is set, filter to the last n matches

    if last\_n\_matches is not None:

        past\_matches = past\_matches.tail(last\_n\_matches)

    # Initialize total points

    total\_points = 0

    # Calculate points for each past match

    for match in past\_matches.itertuples():

        if getattr(match, 'Team\_ID') == row[team\_column]:

            if getattr(match, 'FTR') == 'H':

                total\_points += 3  # Home win

            elif getattr(match, 'FTR') == 'D':

                total\_points += 1  # Draw

        elif getattr(match, 'Opp\_ID') == row[team\_column]:

            if getattr(match, 'FTR') == 'A':

                total\_points += 3  # Away win

            elif getattr(match, 'FTR') == 'D':

                total\_points += 1  # Draw

    # Calculate average points

    matches\_played = len(past\_matches)

    avg\_points = total\_points / matches\_played if matches\_played > 0 else 0

    # Return average points rounded to 3 decimal places

    return round(avg\_points, 3)

# %%

df['team\_points'] = df.apply(lambda x: points(df, x, 'Team\_ID')

    if dataprep\_start\_date is None or x['Date'] >= dataprep\_start\_date else None, axis=1)

df['opp\_points'] = df.apply(lambda x: points(df, x, 'Opp\_ID')

    if dataprep\_start\_date is None or x['Date'] >= dataprep\_start\_date else None, axis=1)

df['team\_form'] = df.apply(lambda x: points(df, x, 'Team\_ID', last\_n\_matches=5)

    if dataprep\_start\_date is None or x['Date'] >= dataprep\_start\_date else None, axis=1)

df['opp\_form'] = df.apply(lambda x: points(df, x, 'Opp\_ID', last\_n\_matches=5)

    if dataprep\_start\_date is None or x['Date'] >= dataprep\_start\_date else None, axis=1)

# %%

# Calculate the the weighted average of the last 5 matches between the two teams

def history\_vs\_opponent\_weighted(df, row, team\_column):

    # Determine opponent column based on team column

    opponent\_column = 'Team\_ID' if team\_column == 'Opp\_ID' else 'Opp\_ID'

    # Combine year, month, and day into an integer 'Date\_temp'

    row\_date\_temp = row['Date'].year \* 10000 + row['Date'].month \* 100 + row['Date'].day

    # Filter for matches between specified teams, excluding current match

    mask = (

        ((df[team\_column] == row[team\_column]) & (df[opponent\_column] == row[opponent\_column])) |

        ((df[team\_column] == row[opponent\_column]) & (df[opponent\_column] == row[team\_column]))

    ) & (df['Date\_temp'] < row\_date\_temp)

    filtered\_matches = df[mask]

    if filtered\_matches.empty:

        return 0  # Return early if no matches found

    # Sort by date and select top 5 recent matches

    recent\_matches = filtered\_matches.sort\_values(by='Date', ascending=False).head(5)

    weights = list(range(len(recent\_matches), 0, -1))

    # Calculate weighted score based on match results

    weighted\_score = sum(

        (3 \* weight if match.FTR == 'H' and match.\_\_getattribute\_\_(team\_column) == match.Team\_ID or

                      match.FTR == 'A' and match.\_\_getattribute\_\_(team\_column) != match.Team\_ID else

         1 \* weight if match.FTR == 'D' else 0)

        for match, weight in zip(recent\_matches.itertuples(), weights)

    )

    # Normalize the weighted score by the sum of weights

    return round(weighted\_score / sum(weights), 3) if weights else 0

# %%

df['team\_hist\_vs'] = df.apply(lambda x: history\_vs\_opponent\_weighted(df, x, 'Team\_ID')

    if dataprep\_start\_date is None or x['Date'] >= dataprep\_start\_date else None, axis=1)

df['opp\_hist\_vs'] = df.apply(lambda x: history\_vs\_opponent\_weighted(df, x, 'Opp\_ID')

    if dataprep\_start\_date is None or x['Date'] >= dataprep\_start\_date else None, axis=1)

# %%

# Calculate rolling averages for the last 5 matches

def rolling\_avgs\_combined(df, row, perspective):

    # Determine the team ID based on the perspective ('Team' or 'Opp')

    if perspective == 'Team':

        team\_id = row['Team\_ID']

    elif perspective == 'Opp':

        team\_id = row['Opp\_ID']

    else:

        raise ValueError("Perspective must be 'Team' or 'Opp'")

    # Get the current match date

    current\_date = row['Date\_temp']

    # Filter past 5 matches for the team

    past\_matches = df[((df['Team\_ID'] == team\_id) | (df['Opp\_ID'] == team\_id)) &

                      (df['Date\_temp'] < current\_date)].sort\_values(by='Date\_temp', ascending=False).head(5)

    # Weights for the matches (most recent match has the highest weight)

    weights = [5, 4, 3, 2, 1]

    # Initialize sums and weighted sums

    shots = []

    shots\_target = []

    # Determine which columns to use and collect the values

    for match in past\_matches.itertuples():

        if match.Team\_ID == team\_id:

            shots.append(getattr(match, 'HS'))  # Home shots

            shots\_target.append(getattr(match, 'HST'))  # Home shots on target

        else:

            shots.append(getattr(match, 'AS'))  # Away shots

            shots\_target.append(getattr(match, 'AST'))  # Away shots on target

    # Calculate the weighted averages of the values

    weighted\_shots = sum(s \* w for s, w in zip(shots, weights))

    weighted\_shots\_target = sum(st \* w for st, w in zip(shots\_target, weights))

    total\_weights = sum(weights[:len(shots)])  # Adjust total weight if there are less than 5 matches

    avg\_shots = weighted\_shots / total\_weights if total\_weights > 0 else 0

    avg\_shots\_target = weighted\_shots\_target / total\_weights if total\_weights > 0 else 0

    # Round the averages to 2 decimal places

    avg\_shots = round(avg\_shots, 2)

    avg\_shots\_target = round(avg\_shots\_target, 2)

    return avg\_shots, avg\_shots\_target

# %%

df['team\_shots'], df['team\_shots\_target'] = zip(\*df.apply(lambda x: rolling\_avgs\_combined(df, x, 'Team')

    if dataprep\_start\_date is None or x['Date'] >= dataprep\_start\_date else (0, 0), axis=1))

df['opp\_shots'], df['opp\_shots\_target'] = zip(\*df.apply(lambda x: rolling\_avgs\_combined(df, x, 'Opp')

    if dataprep\_start\_date is None or x['Date'] >= dataprep\_start\_date else (0, 0), axis=1))

# %%

# Calculate the average goals scored and conceded by a team

def avg\_goals(df, row, team\_column):

    # Season and date of the current match

    current\_season = row['Season']

    current\_date = row['Date']

    # Determine the columns for goals scored and conceded based on perspective

    if team\_column == 'Team\_ID':

        goals\_scored\_column = 'FTHG'  # Assuming FTHG is the column for home team goals

        goals\_conceded\_column = 'FTAG'  # Assuming FTAG is the column for away team goals

    else:

        goals\_scored\_column = 'FTAG'  # Flip the columns if we are looking from the opponent's perspective

        goals\_conceded\_column = 'FTHG'

    # Filter matches from the same season and before the current date

    past\_matches = df[

        (df['Season'] == current\_season) &

        (df['Date'] < current\_date) &

        ((df['Team\_ID'] == row[team\_column]) | (df['Opp\_ID'] == row[team\_column]))

    ]

    # Calculate the average goals scored and conceded

    goals\_scored = 0

    goals\_conceded = 0

    total\_matches = len(past\_matches)

    for match in past\_matches.itertuples():

        if getattr(match, 'Team\_ID') == row[team\_column]:

            goals\_scored += getattr(match, goals\_scored\_column)

            goals\_conceded += getattr(match, goals\_conceded\_column)

        else:  # Team is playing away

            goals\_scored += getattr(match, goals\_scored\_column)

            goals\_conceded += getattr(match, goals\_conceded\_column)

    avg\_goals\_for = goals\_scored / total\_matches if total\_matches > 0 else 0

    avg\_goals\_against = goals\_conceded / total\_matches if total\_matches > 0 else 0

    avg\_goals\_for = round(avg\_goals\_for, 2)

    avg\_goals\_against = round(avg\_goals\_against, 2)

    return avg\_goals\_for, avg\_goals\_against

# %%

# Apply the function and create new columns

df['team\_avg\_goals\_for'], df['team\_avg\_goals\_against'] = zip(\*df.apply(lambda x: avg\_goals(df, x, 'Team\_ID')

    if dataprep\_start\_date is None or x['Date'] >= dataprep\_start\_date else (0, 0), axis=1))

df['opp\_avg\_goals\_for'], df['opp\_avg\_goals\_against'] = zip(\*df.apply(lambda x: avg\_goals(df, x, 'Opp\_ID')

    if dataprep\_start\_date is None or x['Date'] >= dataprep\_start\_date else (0, 0), axis=1))

# %%

# Calculate means only for numeric columns

numeric\_cols = df.select\_dtypes(include=[np.number]).columns

means = df[numeric\_cols].mean()

# Fill missing values in numeric columns with their respective means

df[numeric\_cols] = df[numeric\_cols].fillna(means)

# %%

# Set the FTR to 'X' where the value is currently NaN

df['FTR'] = df['FTR'].fillna('X')

# %%

# Drop every row where 'FTR' is not 'H', 'D', or 'A', or 'X' (if future matches are included)

df = df[df['FTR'].isin(['H', 'D', 'A', 'X'])]

# Map 'H', 'D', and 'A' to 0, 1, and 2 respectively

df['FTR'] = df['FTR'].map({'H': 0, 'D': 1, 'A': 2, 'X': -1}).astype(int)

# %%

# Calculate the expected goals for each team

def calculate\_xg(df, row, team\_column):

    # Initialize the expected goals (xg)

    xg\_total = 0

    count\_matches = 0

    # Season of the current match

    current\_season = row['Season']

    # Date of the current match

    current\_date = pd.to\_datetime(row['Date'], dayfirst=True)  # Ensure the date format is correct

    # Define the opponent column based on the team column

    if team\_column == 'Team\_ID':

        goals\_col = 'FTHG'

        shots\_on\_target\_col = 'HST'

    else:

        goals\_col = 'FTAG'

        shots\_on\_target\_col = 'AST'

    # Filter DataFrame for matches from the same season before the current date

    past\_matches = df[

        (df['Season'] == current\_season) &

        (pd.to\_datetime(df['Date'], dayfirst=True) < current\_date) &

        (df[team\_column] == row[team\_column])

    ]

    # Calculate efficiency and xg

    for match in past\_matches.itertuples():

        goals = getattr(match, goals\_col)

        shots\_on\_target = getattr(match, shots\_on\_target\_col)

        if shots\_on\_target > 0:

            efficiency = goals / shots\_on\_target

            xg\_total += efficiency

            count\_matches += 1

    # Calculate average xg

    if count\_matches > 0:

        avg\_xg = xg\_total / count\_matches

    else:

        avg\_xg = 0

    return avg\_xg

# %%

df['team\_xg'] = df.apply(lambda x: calculate\_xg(df, x, 'Team\_ID')  if dataprep\_start\_date is None or x['Date'] >= dataprep\_start\_date else None, axis=1)

df['opp\_xg'] = df.apply(lambda x: calculate\_xg(df, x, 'Opp\_ID') if dataprep\_start\_date is None or x['Date'] >= dataprep\_start\_date else None, axis=1)

# %%

# Create interaction terms for important features

df['elo\_diff'] = df['team\_elo'] - df['opp\_elo']

df['xg\_diff'] = df['team\_xg'] - df['opp\_xg']

df['points\_diff'] = df['team\_points'] - df['opp\_points']

df['form\_diff'] = df['team\_form'] - df['opp\_form']

# %%

df = df[[

        'Div', 'Season', 'Date\_temp', 'Time', 'DayOTW', 'Team\_ID', 'Opp\_ID', 'FTR',

        'team\_elo', 'opp\_elo',

        'team\_xg', 'opp\_xg',

        'team\_hist\_vs',

        'opp\_hist\_vs',

        'team\_points',

        'opp\_points',

        'elo\_diff',

        'xg\_diff',

        'points\_diff',

        'form\_diff',

        'team\_form',

        'opp\_form',

        'team\_avg\_goals\_for',

        'team\_avg\_goals\_against',

        'opp\_avg\_goals\_for',

        'opp\_avg\_goals\_against',

        'team\_shots', 'opp\_shots',

        'team\_shots\_target', 'opp\_shots\_target',

        'AvgH', 'AvgD', 'AvgA'

         ]]

# %%

# Print the value counts of the Date\_temp column where FTR is -1

print(df[df['FTR'] == -1]['Date\_temp'].value\_counts())

# %%

# Rename 'Date\_temp' to 'Date'

df.rename(columns={'Date\_temp': 'Date'}, inplace=True)

# %%

# Drop duplicate rows based on 'Date', 'Team\_ID', and 'Opp\_ID'

df = df.drop\_duplicates(subset=['Date', 'Team\_ID', 'Opp\_ID'], keep='first')

# %%

import pandas as pd

try:

    if dataprep\_start\_date is not None:

        # Convert date columns to datetime

        df['Date\_temp'] = pd.to\_datetime(df['Date'], format='%Y%m%d')

        # Filter new data based on start date

        df\_new = df[df['Date\_temp'] >= dataprep\_start\_date].copy()

        # Load existing data

        df\_existing = pd.read\_csv(f'data/processed/processed\_data\_{content}.csv')

        df\_existing['Date\_temp'] = pd.to\_datetime(df\_existing['Date'])

        # Filter existing data to remove overlap with new data

        df\_existing = df\_existing[df\_existing['Date\_temp'] < dataprep\_start\_date]

        # Combine and sort data

        df\_final = pd.concat([df\_existing, df\_new], ignore\_index=True)

        df\_final.sort\_values(['Date\_temp', 'Time'], inplace=True)

        # Clean up temporary columns

        df\_final.drop(columns='Date\_temp', inplace=True)

    else:

        df\_final = df.copy()

    # Save the final DataFrame

    df\_final.to\_csv(f'data/processed/processed\_data\_{content}.csv', index=False)

    print(f"Data saved: {df\_final.shape[0]} matches")

except Exception as e:

    print(f"Error: {e}")

Model training code

# %%

# Standard library imports

import os

import sys

import re

import warnings

import random

import hashlib

import ast

# Data manipulation and analysis

import numpy as np

import pandas as pd

# Visualization libraries

import matplotlib.pyplot as plt

import seaborn as sns

%matplotlib inline

# Preprocessing and model selection tools

from sklearn.model\_selection import (train\_test\_split, StratifiedKFold, GridSearchCV,

                                     RandomizedSearchCV, TimeSeriesSplit)

from sklearn.preprocessing import StandardScaler, MinMaxScaler

from sklearn.utils.class\_weight import compute\_class\_weight

# Metrics and scoring

from sklearn.metrics import (balanced\_accuracy\_score, classification\_report, f1\_score,

                             make\_scorer, confusion\_matrix, precision\_score, accuracy\_score)

# Machine learning models

from sklearn.pipeline import Pipeline

from sklearn.ensemble import (RandomForestClassifier, GradientBoostingClassifier, VotingClassifier, StackingClassifier,

                              AdaBoostClassifier, ExtraTreesClassifier, BaggingClassifier, HistGradientBoostingClassifier)

from sklearn.naive\_bayes import GaussianNB

from sklearn.linear\_model import LogisticRegression

from sklearn.utils import check\_random\_state

from sklearn.tree import DecisionTreeClassifier

from sklearn.svm import SVC

# Neural networks

from sklearn.neural\_network import MLPClassifier

# selection

from sklearn.feature\_selection import SelectPercentile, chi2, SelectKBest

from sklearn.base import clone

# Advanced models and ensemble techniques

import xgboost as xgb

from xgboost import XGBClassifier

from lightgbm import LGBMClassifier

from catboost import CatBoostClassifier

# Iputing missing values

from sklearn.impute import SimpleImputer

# Handling imbalanced datasets

from imblearn.pipeline import Pipeline as ImbPipeline

from imblearn.over\_sampling import SMOTE

# Encoding and feature selection

from category\_encoders import TargetEncoder

from scipy.stats import randint, uniform

# Model persistence

from joblib import dump, load

# Miscellaneous settings

warnings.filterwarnings('ignore')

# %%

content = "allcomps\_3s\_june2024"

# %%

# Load the processed data csv into a DataFrame

df = pd.read\_csv(f'data/processed/processed\_data\_{content}.csv')

# %%

# Remove duplicate rows

df.drop\_duplicates(inplace=True)

# %%

# Parse the date\_temp column, which is in YYYYMMDD format, into a datetime object, and store it in a new column 'date\_temporary'

df['date\_temporary'] = pd.to\_datetime(df['Date'], format='%Y%m%d')

# %% [markdown]

# ### Date settings

# %%

# Get the current date dynamically

date\_today = pd.Timestamp.now().normalize()  # .normalize() sets the time to 00:00:00

# Declare a date by setting day, month, and year

date\_specific = pd.Timestamp(year=2024, month=5, day=24)

# Calculate the date 2 weeks ago from the current date

date\_delta = date\_specific - pd.DateOffset(days=10)

# Specific start date

#date\_start = date\_specific - pd.DateOffset(days=1000)

# No filter, all data

date\_start = pd.Timestamp(year=2022, month=7, day=1)

# %%

# Delete all rows where the date\_temporary column is older than date\_start

df = df[df['date\_temporary'] >= date\_start]

# %%

# define df\_validationset as all the rows in df where the date\_temporary column is greater than date\_delta

df\_validationset = df[df['date\_temporary'] > date\_delta]

# define df as all the rows in df where the date\_temporary column is less than or equal to date\_delta

df = df[df['date\_temporary'] <= date\_delta]

# %%

# Drop the date\_temporary column

df.drop(columns=['date\_temporary'], inplace=True)

df\_validationset.drop(columns=['date\_temporary'], inplace=True)

# %%

# Read and parse team and competition data from respective text files into dictionaries

teams\_dict = {}

comps\_dict = {}

with open(f'data/teams\_dict\_{content}.txt', 'r') as file:

    data = file.read()

    teams\_dict = ast.literal\_eval(data)

with open(f'data/comps\_dict\_{content}.txt', 'r') as file:

    data = file.read()

    comps\_dict = ast.literal\_eval(data)

# %%

# Sort the df and df\_validationset DataFrames by the 'Date', 'Div', 'Time' columns

df.sort\_values(['Date', 'Div', 'Time'], inplace=True)

df\_validationset.sort\_values(['Date', 'Div', 'Time'], inplace=True)

# Set the 'Date' and 'FTR' column as the index

df.set\_index(['Date'], inplace=True)

df\_validationset.set\_index(['Date'], inplace=True)

# %%

# Split the data into X and y

X = df.drop(['FTR', 'AvgH', 'AvgD', 'AvgA'], axis=1)

y = df['FTR']

X.columns = [re.sub(r'[<]', '\_st\_', str(col)) for col in X.columns]

X.columns = [re.sub(r'[>]', '\_gt\_', str(col)) for col in X.columns]

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# %% [markdown]

# ### Hyperparameters

# %%

# Declare the hyperparameter tuning grid for the models

param\_dist = {

    'xgb\_\_clf\_\_max\_depth': [1,2,3, 4, 6],

    'xgb\_\_clf\_\_learning\_rate': [0.05, 0.1, 0.15],

    'xgb\_\_clf\_\_reg\_lambda': [0.01, 0.1],

    'xgb\_\_clf\_\_alpha': [0, 0.5, 1],

    'xgb\_\_clf\_\_colsample\_bytree': [0.7, 0.9],

    'xgb\_\_clf\_\_subsample': [0.75, 0.85],

    'xgb\_\_clf\_\_n\_estimators': [1, 5, 10],

    'rf\_\_clf\_\_max\_depth': [1, 2],

    'rf\_\_clf\_\_min\_samples\_split': [3, 6],

    'rf\_\_clf\_\_min\_samples\_leaf': [1, 3],

    'rf\_\_clf\_\_n\_estimators': [1, 5, 10],

    'rf\_\_clf\_\_max\_features': ['sqrt', 'log2'],

    'lr\_\_clf\_\_C': [0.1, 1],

    'lr\_\_clf\_\_penalty': ['l1', 'l2', 'elasticnet'],

    'lr\_\_clf\_\_solver': ['saga'],

    'lr\_\_clf\_\_l1\_ratio': [0.5],

    'lr\_\_clf\_\_class\_weight': ['balanced'],

    'gb\_\_clf\_\_learning\_rate': [0.01, 0.1, 0.15],

    'gb\_\_clf\_\_n\_estimators': [1, 5, 10],

    'gb\_\_clf\_\_max\_depth': [3, 5 , 7],

    'gb\_\_clf\_\_min\_samples\_split': [2, 5],

    'gb\_\_clf\_\_min\_samples\_leaf': [1, 2],

}

# %%

# Function to generate sliding windows for training and testing

def generate\_sliding\_windows\_by\_div(X):

    grouped = X.groupby('Div')  # Group by 'Div' only

    windows = []

    for div, group in grouped:

        indices = group.index.tolist()  # Get indices of each group

        n\_samples = len(indices)

        if n\_samples < 2:

            print(f"Skipping division {div} with insufficient samples: {n\_samples}")

            continue

        # Using 80% of data for training and the rest for testing

        split\_point = int(n\_samples \* 0.8)

        train\_indices = indices[:split\_point]

        test\_indices = indices[split\_point:]

        if train\_indices and test\_indices:  # Ensure both are non-empty

            windows.append((train\_indices, test\_indices))

    return windows

# %%

from sklearn.metrics import make\_scorer, f1\_score

from sklearn.model\_selection import RandomizedSearchCV

from xgboost import XGBClassifier

from sklearn.model\_selection import train\_test\_split

def xgb\_early\_stopping\_score(y\_true, y\_pred):

    """

    Custom scorer that uses early stopping.

    """

    # Return the F1 score or any other relevant metric

    return f1\_score(y\_true, y\_pred, average='macro')

custom\_scorer = make\_scorer(xgb\_early\_stopping\_score, greater\_is\_better=True)

# %%

def create\_pipeline(base\_estimator, fts=14, random\_state=42):

    pipeline = ImbPipeline([

        ('imputer', SimpleImputer(strategy='mean')),

        ('target\_encoder', TargetEncoder()),

        ('scaler', StandardScaler()),

        ('min\_max\_scaler', MinMaxScaler()),

        ('smote', SMOTE(random\_state=random\_state, k\_neighbors=2)),

        ('select', SelectKBest(chi2, k=fts)),

        ('clf', clone(base\_estimator))

    ])

    return pipeline

# %%

classifiers = {

    'xgb': XGBClassifier(random\_state=42, verbose=0),

    'gb': GradientBoostingClassifier(random\_state=42, verbose=0),

    'lr': LogisticRegression(random\_state=42, verbose=0, multi\_class='ovr'),

    'rf': RandomForestClassifier(random\_state=42, verbose=0),

    'ada': AdaBoostClassifier(random\_state=42),

    'lgbm': LGBMClassifier(random\_state=42, force\_col\_wise='true', verbose=0),

    'MLP': MLPClassifier(random\_state=42, verbose=0)

}

# Generate pipelines for each classifier

pipelines = {name: create\_pipeline(clf) for name, clf in classifiers.items()}

# Create the ensemble classifier

ensemble\_clf = VotingClassifier(

    estimators=[(name, pipeline) for name, pipeline in pipelines.items()],

    voting='soft'

)

# %%

scoring = {

    'f1\_score': make\_scorer(f1\_score, average='macro'),

    'accuracy': make\_scorer(balanced\_accuracy\_score),

    'precision': make\_scorer(precision\_score, average='macro'),

}

# %%

def enhanced\_rolling\_window\_ensemble(X, y, window\_size, step\_size):

    num\_samples = len(X)

    start\_index = 0

    additional\_training\_data = pd.DataFrame(columns=['y'])

    counter = 1

    while start\_index + window\_size < num\_samples:

        end\_index = start\_index + window\_size

        X\_train = pd.concat([X.iloc[start\_index:end\_index], additional\_training\_data.drop(columns=['y'], errors='ignore')])

        if not additional\_training\_data.empty:

            y\_train = pd.concat([y.iloc[start\_index:end\_index], additional\_training\_data['y']])

        else:

            y\_train = y.iloc[start\_index:end\_index]

        X\_test = X.iloc[end\_index:end\_index + step\_size]

        y\_test = y.iloc[end\_index:end\_index + step\_size]

        print(f"Iteration {counter}: Training on matches {start\_index} to {end\_index} of {num\_samples}")

        clf = RandomizedSearchCV(

            estimator=ensemble\_clf,

            param\_distributions=param\_dist,

            n\_iter=2,

            scoring=custom\_scorer,

            refit='f1\_score',

            cv=TimeSeriesSplit(n\_splits=2),

            random\_state=42,

            n\_jobs=-1,

            verbose=0,

            error\_score='raise'

        )

        clf.fit(X\_train, y\_train)

        y\_pred = clf.predict(X\_test)

        wrong\_indices = y\_test != y\_pred

        wrong\_data = X\_test[wrong\_indices].copy()

        wrong\_data['y'] = y\_test[wrong\_indices]

        additional\_training\_data = wrong\_data

        print("Accuracy:", accuracy\_score(y\_test, y\_pred))

        start\_index += step\_size

        counter += 1

    return clf

# %% [markdown]

# ### Training ###

# %%

X = X.reset\_index(drop=True)

y = y.reset\_index(drop=True)

print(f"Number of rows: {len(X)} ")

window\_size = 5000

step\_size =  1000

model = enhanced\_rolling\_window\_ensemble(X, y, window\_size, step\_size)

# %%

model

# %%

best\_model = model.best\_estimator\_

# %%

# Print a classification report for the test set

y\_pred = model.predict(X\_test)

print("Classification Report for the Test Set:")

print(classification\_report(y\_test, y\_pred))

# %%

# # Print a classification report for the validation set

# Remove rows where FTR = -1

df\_val\_temp = df\_validationset[df\_validationset['FTR'] != -1]

# Prepare the validation data

X\_val = df\_val\_temp.drop(['FTR', 'AvgH', 'AvgD', 'AvgA'], axis=1)

y\_val = df\_val\_temp['FTR']

# Predict the validation set

y\_val\_pred = best\_model.predict(X\_val)

# Print the classification report for the validation set

print("Classification Report for the Validation Set:")

print(classification\_report(y\_val, y\_val\_pred))

# %%

# Feature Importances

# Initialize a dictionary to store feature importances

feature\_importances = {}

# Loop through each classifier in the ensemble

for clf\_name, clf\_pipeline in best\_model.named\_estimators\_.items():

    if hasattr(clf\_pipeline.named\_steps['clf'], 'feature\_importances\_'):

        # Extract feature importances

        importances = clf\_pipeline.named\_steps['clf'].feature\_importances\_

        # Access feature names via the 'select' step in pipeline if available

        # Assuming feature selection might alter the features passed to the classifier

        if 'select' in clf\_pipeline.named\_steps:

            mask = clf\_pipeline.named\_steps['select'].get\_support()  # Get the boolean mask

            feature\_names = np.array(X.columns)[mask]

        else:

            feature\_names = np.array(X.columns)

        # Combine feature names and their corresponding importance

        feature\_importances[clf\_name] = pd.Series(importances, index=feature\_names)

# Now plot the feature importances

plt.figure(figsize=(12, 6))

avg\_importances = pd.DataFrame(feature\_importances).mean(axis=1).sort\_values(ascending=False)

avg\_importances.plot(kind='bar')

plt.title('Average Feature Importances Across Ensemble Models')

plt.ylabel('Importance')

plt.xlabel('Features')

plt.show()

# %%

# confusion matrix

cm = confusion\_matrix(y\_test, y\_pred)

# Plot the confusion matrix

plt.figure(figsize=(8, 6))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False)

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.title('Confusion Matrix')

plt.show()

# %%

df\_val = df\_validationset.copy()

# Calculate the predicted probabilities for the validation set

y\_val\_proba = best\_model.predict\_proba(df\_val.drop(columns=['FTR', 'AvgH', 'AvgD', 'AvgA']))

# %%

df\_val['Prob1'] = y\_val\_proba[:, 0].round(3)

df\_val['ProbX'] = y\_val\_proba[:, 1].round(3)

df\_val['Prob2'] = y\_val\_proba[:, 2].round(3)

# Get the column index of the y\_val\_proba with the highest probability

df\_val['Prediction'] = y\_val\_proba.argmax(axis=1)

# Map the prediction column to the actual result

df\_val['Prediction'] = df\_val['Prediction'].map({0: '1', 1: 'X', 2: '2'})

# Display all predictions

filtered\_df\_val = df\_val.copy()

filtered\_df\_val.reset\_index(inplace=True)

# Map the 'Team\_ID' and 'Opp\_ID' columns to the actual team names

index\_to\_team = {v: k for k, v in teams\_dict.items()}

filtered\_df\_val['Team'] = filtered\_df\_val['Team\_ID'].map(index\_to\_team)

filtered\_df\_val['Opponent'] = filtered\_df\_val['Opp\_ID'].map(index\_to\_team)

# Map the 'Div' column to the actual competition name

index\_to\_comp = {v: k for k, v in comps\_dict.items()}

filtered\_df\_val['Div'] = filtered\_df\_val['Div'].map(index\_to\_comp)

display\_columns = [

    'Div',

    'Date', 'Time', 'Team', 'Opponent',

    'FTR',

    'team\_points', 'opp\_points',

    'team\_form', 'opp\_form',

    'Prediction',

    'team\_elo',

    'opp\_elo',

    'elo\_diff',

    'team\_xg', 'opp\_xg',

    'Prob1', 'ProbX', 'Prob2',

    'AvgH', 'AvgD', 'AvgA',

]

# %%

output = filtered\_df\_val[display\_columns]

output.sort\_values(['Div', 'Date', 'Team'], inplace=True)

# %%

len(output)

# %%

# Function to make predictions based on probabilities

def make\_prediction(row):

    prob1 = row['Prob1']

    probX = row['ProbX']

    prob2 = row['Prob2']

    # Directly return '1' or '2' if their probabilities are greater than 0.65

    if prob1 > 0.65:

        return '1'

    if prob2 > 0.65:

        return '2'

    # Define the expected value (probability \* bookmaker's odds)

    # Calculate combined probabilities for '1X' and 'X2'

    prob1X = prob1 + probX

    probX2 = probX + prob2

    # Create a dictionary to compare probabilities with bet types

    bets = {

        '1': prob1,

        'X': probX,

        '2': prob2,

        '1X': prob1X,

        'X2': probX2

    }

    # Determine the best bet by finding the maximum probability

    best\_bet = max(bets, key=bets.get)

    return best\_bet

# %%

output['1X2'] = output.apply(make\_prediction, axis=1)

# %%

def is\_bet\_correct(row):

    if row['FTR'] == 0:

        return row['1X2'] in ['1', '1X']

    elif row['FTR'] == 1:

        return row['1X2'] in ['1X', 'X', 'X2']

    elif row['FTR'] == 2:

        return row['1X2'] in ['X2', '2']

    return False

# %%

if output['FTR'] is not None:

    output['Correct'] = output.apply(is\_bet\_correct, axis=1)

else:

    output['Correct'] = None

# %%

def bet\_confidence(row):

    Prob1 = row['Prob1']

    ProbX = row['ProbX']

    Prob2 = row['Prob2']

    # parse the probabilities into floats

    Prob1 = float(Prob1)

    ProbX = float(ProbX)

    Prob2 = float(Prob2)

    conf = 0

    if row['1X2'] == '1':

        conf =  Prob1

    elif row['1X2'] == '1X':

        conf = Prob1 + ProbX

    elif row['1X2'] == 'X':

        conf = ProbX

    elif row['1X2'] == 'X2':

        conf = ProbX + Prob2

    elif row['1X2'] == '2':

        conf =  Prob2

    # round the confidence to 2 decimal places

    conf = round(conf, 2)

    return conf

# %%

output['Confidence'] = output.apply(bet\_confidence, axis=1)

# %%

def value\_bet(row):

    Prob1 = float(row['Prob1'])

    ProbX = float(row['ProbX'])

    Prob2 = float(row['Prob2'])

    AvgH = float(row['AvgH'])

    AvgD = float(row['AvgD'])

    AvgA = float(row['AvgA'])

    # Calculate the probabilities of each outcome

    OddProb1 = 1 / AvgH

    OddProbX = 1 / AvgD

    OddProb2 = 1 / AvgA

    if row['1X2'] in ['1', '1X'] and Prob1 > OddProb1:

        return True

    elif row['1X2'] in ['X2', '2'] and Prob2 > OddProb2:

        return True

    elif row['1X2'] == 'X' and ProbX > OddProbX:

        return True

    else:

        return False

# %%

output['Value'] = output.apply(value\_bet, axis=1)

# %%

# Map the 'FTR' back to the actual result

output['FTR'] = output['FTR'].map({0: '1', 1: 'X', 2: '2'})

# %% [markdown]

# ### Validation

# %%

values = ['1', 'X', '2']

# Filter the DataFrame based on the 'FTR' column and count the rows

total\_rows = output[output['FTR'].isin(values)].shape[0]

# Display the total amount of predictions where the value is True

#total\_correct = output['Correct'].sum()

total\_correct = output[(output['FTR'].isin(values)) & (output['Correct'])].shape[0]

# Calculate the percentage of correct predictions

correct\_percentage = (total\_correct / total\_rows) \* 100

# Display the results

print(f"Total Rows: {total\_rows}")

print(f"Total Correct Predictions: {total\_correct}")

print(f"Percentage of Correct Predictions: {correct\_percentage:.2f}%")

# %%

# Identify the most interesting matches to bet on

# Timestamp

import datetime

# Get the current date and time

now = datetime.datetime.now()

# Format the current date and time as a string

timestamp = now.strftime("%Y%m%d\_%H%M%S")

# Keep only the rows where 'FTR' column is null, '1X2' is 1 or 1X, and 'AvgH' is greater than 2

interesting\_matches\_1 = output[(output['FTR'].isnull()) & (output['1X2'].isin(['1', '1X'])) & (output['AvgH'] > 2)]

# Keep only the rows where 'FTR' column is null, '1X2' is 2 or X2, and 'AvgA' is greater than 3

interesting\_matches\_2 = output[(output['FTR'].isnull()) & (output['1X2'].isin(['X2', '2'])) & (output['AvgA'] > 3)]

# Concatenate the two DataFrames

interesting\_matches = pd.concat([interesting\_matches\_1, interesting\_matches\_2])

interesting\_matches.to\_csv(f'data/predictions/interesting\_matches\_{content}\_{timestamp}.csv', index=False)

# %%

# if AvgH exists

if 'AvgH' in output.columns:

    # Change the decimal sign to a point for AvgH, AvgD, and AvgA columns to avoid parsing issues

    output['AvgH'] = output['AvgH'].apply(lambda x: str(x).replace(',', '.'))

    output['AvgD'] = output['AvgD'].apply(lambda x: str(x).replace(',', '.'))

    output['AvgA'] = output['AvgA'].apply(lambda x: str(x).replace(',', '.'))

    # parse AvgH, AvgD, AvgA columns as float

    output['AvgH'] = output['AvgH'].astype(float)

    output['AvgD'] = output['AvgD'].astype(float)

    output['AvgA'] = output['AvgA'].astype(float)

    # parse team\_xg, opp\_xg, team\_form, opp\_form, team\_points, opp\_points as float and round to 3 decimal places

    output['team\_xg'] = output['team\_xg'].astype(float).round(3)

    output['opp\_xg'] = output['opp\_xg'].astype(float).round(3)

    output['team\_form'] = output['team\_form'].astype(float).round(3)

    output['opp\_form'] = output['opp\_form'].astype(float).round(3)

    output['team\_points'] = output['team\_points'].astype(float).round(3)

    output['opp\_points'] = output['opp\_points'].astype(float).round(3)

    # Change the decimal sign to a comma

    output['team\_xg'] = output['team\_xg'].apply(lambda x: str(x).replace('.', ','))

    output['opp\_xg'] = output['opp\_xg'].apply(lambda x: str(x).replace('.', ','))

    output['team\_form'] = output['team\_form'].apply(lambda x: str(x).replace('.', ','))

    output['opp\_form'] = output['opp\_form'].apply(lambda x: str(x).replace('.', ','))

    output['team\_points'] = output['team\_points'].apply(lambda x: str(x).replace('.', ','))

    output['opp\_points'] = output['opp\_points'].apply(lambda x: str(x).replace('.', ','))

    output['AvgH'] = output['AvgH'].apply(lambda x: str(x).replace('.', ','))

    output['AvgD'] = output['AvgD'].apply(lambda x: str(x).replace('.', ','))

    output['AvgA'] = output['AvgA'].apply(lambda x: str(x).replace('.', ','))

    output['Prob1'] = output['Prob1'].apply(lambda x: str(x).replace('.', ','))

    output['ProbX'] = output['ProbX'].apply(lambda x: str(x).replace('.', ','))

    output['Prob2'] = output['Prob2'].apply(lambda x: str(x).replace('.', ','))

    output['Confidence'] = output['Confidence'].apply(lambda x: str(x).replace('.', ','))

# %%

print("TOTAL ROWS: ", len(output))

# %%

# Timestamp

import datetime

# Get the current date and time

now = datetime.datetime.now()

# Format the current date and time as a string

timestamp = now.strftime("%Y%m%d\_%H%M%S")

# save filtered\_df\_val[display\_columns] to a CSV file

output.to\_csv(f'data/predictions/predictions\_{content}\_{timestamp}.csv', index=False)

1. A machine learning approach that combines multiple individual models to improve predictive performance and robustness compared to any single model alone. [↑](#footnote-ref-1)
2. The practice of streamlining and automating the deployment, monitoring, and management of machine learning models in production to enhance reliability and efficiency. [↑](#footnote-ref-2)
3. Home/Away team shots, Home/Away team shots on target [↑](#footnote-ref-3)
4. In machine learning, "weak learners" are simple models that perform slightly better than random guessing, and they are often combined in ensemble methods to create a more accurate and robust predictive model. [↑](#footnote-ref-4)
5. hierarchical models used for classification and regression, splitting data into branches to make predictions based on feature values. [↑](#footnote-ref-5)
6. Multicollinearity occurs when predictor variables in a regression model are highly correlated, affecting results. [↑](#footnote-ref-6)
7. Bias refers to the error introduced by overly simplistic models that fail to capture the underlying patterns in the data. [↑](#footnote-ref-7)
8. variance refers to the extent to which a model's predictions change when trained on different subsets of the training data, indicating its sensitivity to small fluctuations in the dataset. [↑](#footnote-ref-8)
9. Overfitting in machine learning is when a model learns the training data too well, including noise and details, making it perform poorly on new, unseen data. [↑](#footnote-ref-9)
10. A perceptron is a basic neural network unit that makes predictions by weighing input features and applying an activation function to produce an output. [↑](#footnote-ref-10)
11. a model inspired by the human brain, consisting of layers of interconnected nodes that learn to make predictions from data. [↑](#footnote-ref-11)
12. The F1 score is a metric that combines precision and recall to provide a single measure of a model's accuracy, particularly useful for imbalanced datasets. [↑](#footnote-ref-12)
13. Jupyter is an open-source tool that allows you to create and share documents containing live code, equations, visualizations, and explanatory text for data analysis and machine learning. [↑](#footnote-ref-13)
14. In machine learning, "true positives" refer to instances where the model correctly identifies positive cases or events that are actually positive. [↑](#footnote-ref-14)
15. False negatives occur when the model incorrectly predicts that a positive instance is negative, missing the correct identification [↑](#footnote-ref-15)
16. "fitting" refers to the process of training a model on data so that it can make accurate predictions. [↑](#footnote-ref-16)