Beating the bookmakers at their own game!

A data-science deep dive into predicting football games with machine learning models

Jan Vermeerbergen
Final Work - Bachelor Toegepaste Informatica
Erasmushogeschool Brussel
14/06/2024

Table of Contents

Introduction 4

1. Methodology 5

1.1. Data Collection 5

1.2. Types of Data Collected 6

1.3. Data Cleaning and Preprocessing 6

1.4. Handling Missing Data 7

1.5. Data Engineering and Feature selection 7

1.5.1. Team Form 7

1.5.2. ELO rating 7

1.5.3. Average points per game 8

1.5.4. Average goals scored/conceded per game 8

1.5.5. Rolling averages 9

1.5.6. Expected Goals (XG) 9

1.5.7. Team vs Opponent metric gaps 9

2. Model Development 10

2.1. Model 1: XGBoost Classifier 10

2.2. Model 2: Gradient Boosting Classifier 10

2.3. Model 3: Logistic Regression 11

2.4. Model 4: Random Forest Classifier 12

2.5. Model 5: Adaboost Classifier 13

2.6. Model 6: LGBM Classifier 14

2.7. Model 7: MLP Classifier 14

2.8. Hyperparameter Tuning 16

2.8.1. XGBoost Classifier 17

2.8.2. Random Forest Classifier 17

2.8.3. Logistic Regression 18

2.8.4. Gradient Boosting 18

2.8.5. Randomized Search Cross Validation 18

3. Ensemble Model Construction 20

3.1. Data collection 20

3.2. Data processing 20

3.3. Training the model 21

3.4. Ensemble Techniques 22

3.4.1. Voting 22

3.4.2. Stacking 23
3.4.3. Bagging 23
3.4.4. Boosting 24
4. Model Evaluation 25
4.1. Evaluation Metrics 25
4.1.1. Accuracy 25
4.1.2. Precision 26
4.1.3. Recall 26
4.1.4. F1 26
4.2. Confusion Matrix 27
4.3. Feature Importances 27
4.4. Cross Validation 29
4.5. Performance 30
4.5.1. Performance strategy 30
4.5.2. Classification reports 31
5. Conclusion 32
6. Bibliography 33
7. Addendum 34
7.1. Data collection code 34
7.2. Data engineering code 36
7.3. Model training code 46

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] 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]' 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.

[1] A machine learning approach that combines multiple individual models to improve predictive performance and robustness compared to any single model alone.
[2] The practice of streamlining and automating the deployment, monitoring, and management of machine learning models in production to enhance reliability and efficiency.

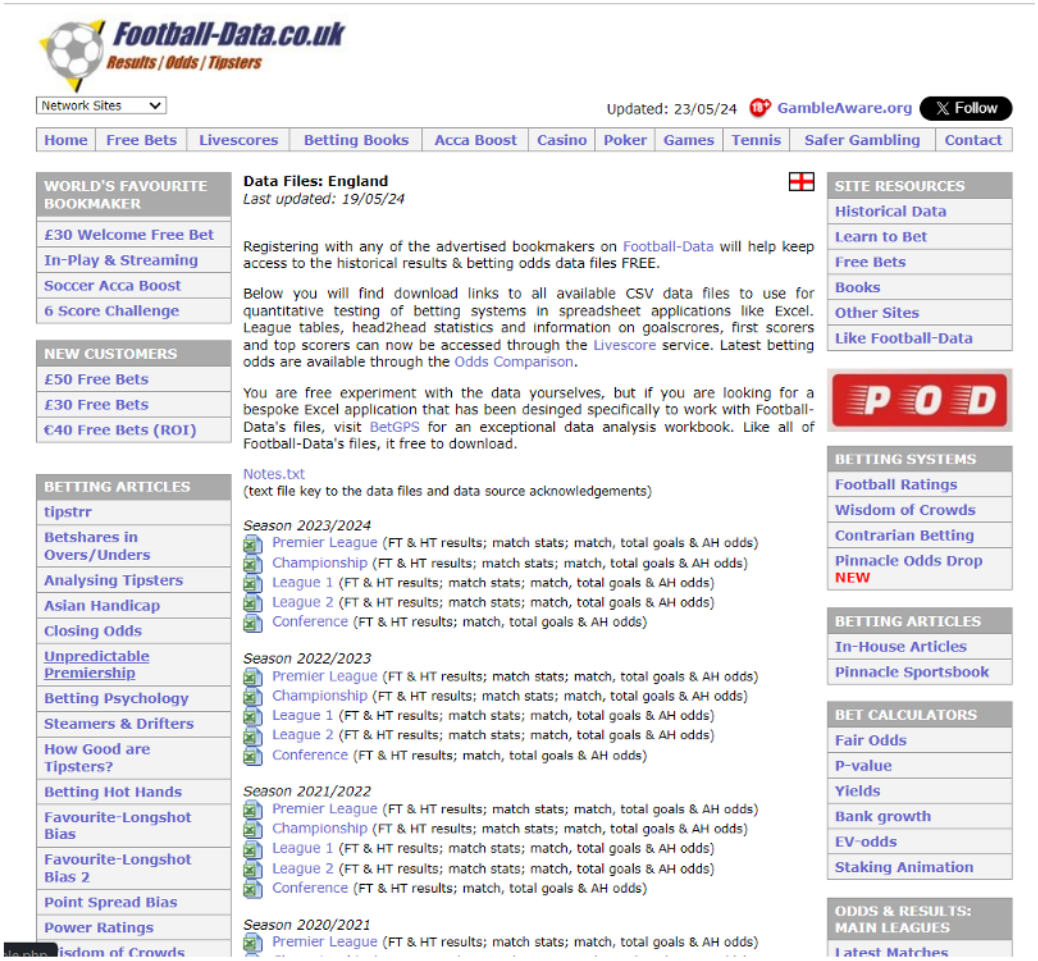
1. Methodology

1.1. 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.

In 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.



1.2. 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

1.3. 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.

```
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
```

1.4. 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.

1.5. 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.

1.5.1. 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.

1.5.2. 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.

Team	ELO	Team	ELO	Team	ELO	Team	ELO
Liverpool	1739	Lyon	1569	Ein Frankfurt	1500	West Ham	1443
Real Madrid	1711	Roma	1567	Luton	1497	Metz	1439
Paris SG	1692	Ath Madrid	1563	Nice	1495	Granada	1438
Barcelona	1690	Strasbourg	1561	Udinese	1495	Nantes	1438
Man City	1672	Brighton	1559	Brest	1494	Wolves	1434
Arsenal	1671	Villarreal	1556	Crystal Palace	1490	Las Palmas	1433
Dortmund	1653	Fiorentina	1551	Sevilla	1489	Augsburg	1431
Inter	1652	Bournemouth	1544	Cagliari	1487	Cadiz	1429
Juventus	1622	Monza	1544	Bologna	1486	Alaves	1425
Lazio	1620	Toulouse	1543	Mallorca	1483	Vallecano	1425
Man United	1620	Fulham	1538	Freiburg	1480	Almeria	1424
Chelsea	1617	Torino	1536	Hoffenheim	1475	FC Koln	1423
Napoli	1608	Stuttgart	1534	Lorient	1475	Werder Bremen	1423
Bayern Munich	1607	Ath Bilbao	1531	Burnley	1474	Bochum	1422
Milan	1607	Atalanta	1529	Montpellier	1474	Getafe	1421
Rennes	1607	Newcastle	1528	Osasuna	1474	Celta	1417
RB Leipzig	1604	Union Berlin	1526	Valencia	1474	Verona	1414
Marseille	1598	Nice	1524	Brentford	1472	Sassuolo	1409
Tottenham	1596	Mainz	1520	Reims	1466	Granada	1408
Sociedad	1595	Aston Villa	1519	Sheffield United	1464	Clermont	1403
Monaco	1582	Frosinone	1517	Lecce	1456	Empoli	1386
Betis	1578	Girona	1515	Le Havre	1454	Salernitana	1314
Lille	1574	M'gladbach	1510	Darmstadt	1453		
Leverkusen	1573	Nott'm Forest	1504	Everton	1450		
Lens	1572	Genoa	1501	Wolfsburg	1447		

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

1.5.3. 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.

1.5.4. 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.

1.5.5. Rolling averages

For several columns in the dataset, including HS, AS, HST, and AST[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.

1.5.6. 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.

1.5.7. 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.

[3] Home/Away team shots, Home/Away team shots on target

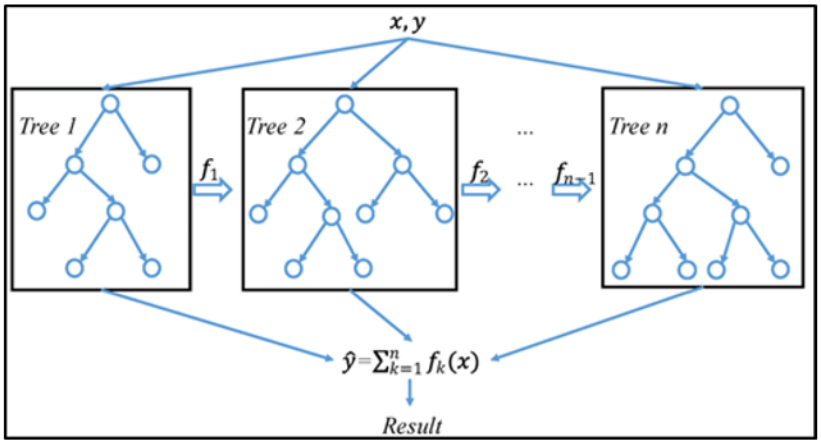
2. Model Development

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.

2.1. 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], usually decision trees[5], and combines them to form a strong predictive model.

One 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.

2.2. Model 2: Gradient Boosting Classifier

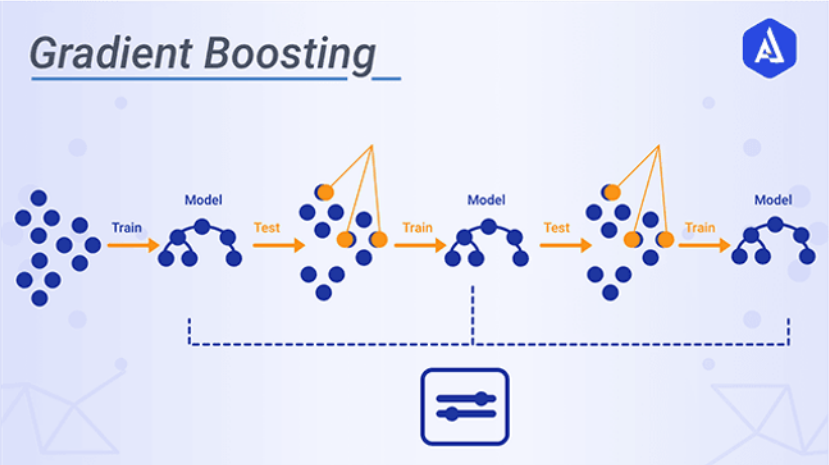
In the realm of sports analytics, particularly when predicting football match outcomes, the GradientBoostingClassifier (GBC) stands out as a powerful tool. This model, part of an

[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.

[5] hierarchical models used for classification and regression, splitting data into branches to make predictions based on feature values.

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 datasets, 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.

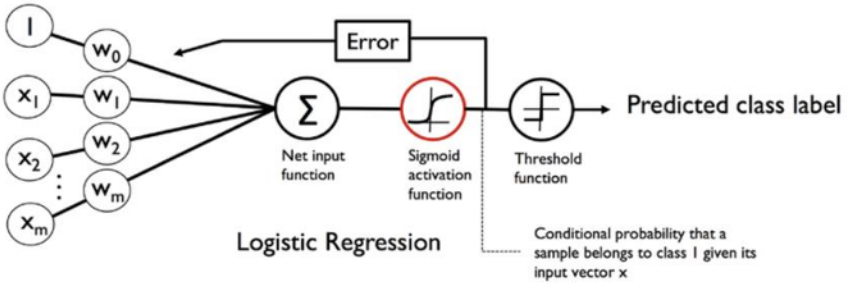
While the GradientBoostingClassifier is undeniably potent, balancing its strengths and weaknesses through meticulous implementation is key to unlocking its full potential in sports analytics.

2.3. 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.

However, 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] and may require careful preprocessing of data to mitigate these effects.

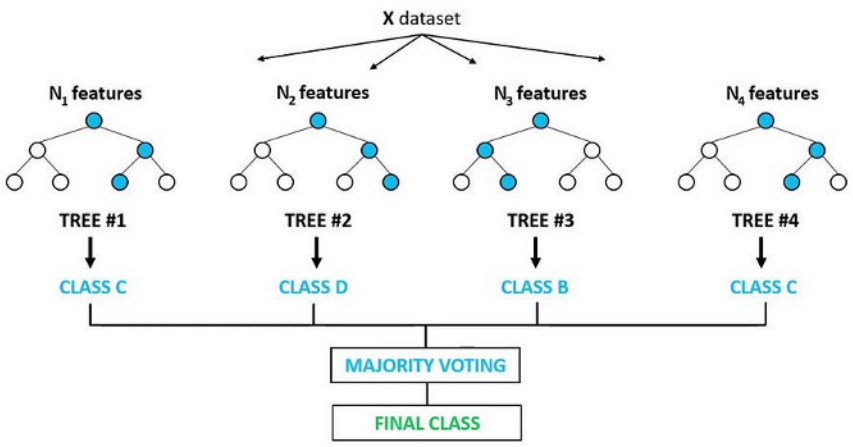
Despite these weaknesses, Logistic Regression's role in ensemble models for predicting football outcomes is significant. When combined with other models, it helps to capture a diverse range of patterns in the data, enhancing overall predictive performance. By leveraging its strengths and compensating 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.

2.4. 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.

However, 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.

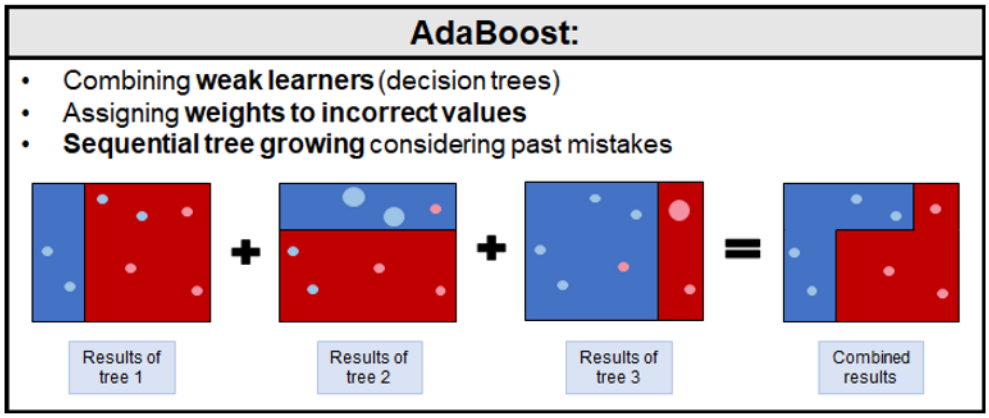
[6] Multicollinearity occurs when predictor variables in a regression model are highly correlated, affecting results.

2.5. 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] and variance[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.

However, AdaBoostClassifier is not without its weaknesses. It is sensitive to noisy data and outliers, which can lead to overfitting[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.



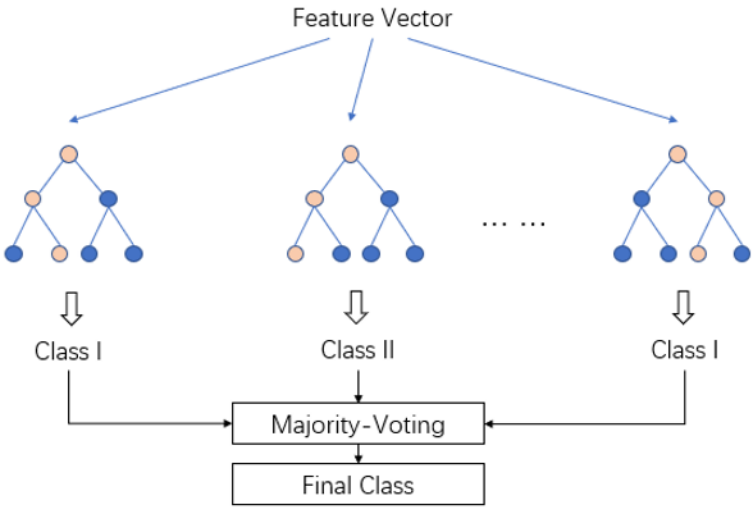
In the context of football match prediction, the AdaBoostClassifier demonstrates notable promise. Its capacity to refine predictions by addressing previous errors makes it a valuable component of ensemble models designed for this purpose. When combined with other models, it can significantly enhance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

[7] Bias refers to the error introduced by overly simplistic models that fail to capture the underlying patterns in the data.
[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.
[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.

2.6. Model 6: LGBM Classifier

When 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.

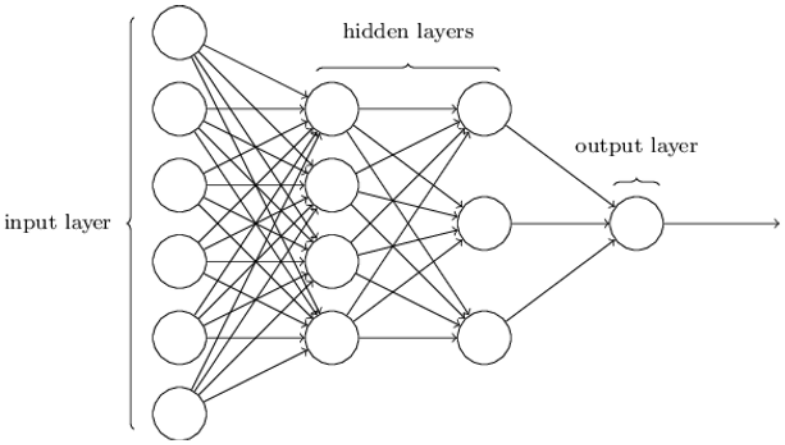
In ensemble settings, combining the LGBMClassifier with other models can mitigate some of its weaknesses, enhancing overall prediction accuracy. By blending the strengths of multiple algorithms, the ensemble approach provides a more balanced and reliable prediction framework, crucial for the dynamic 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.

2.7. 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]Classifier (MLPClassifier). This neural network [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.

[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.
[11] a model inspired by the human brain, consisting of layers of interconnected nodes that learn to make predictions from 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.

2.8. 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.

```
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],

}
```

2.8.1. 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.

2.8.2. 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.

2.8.3. Logistic Regression

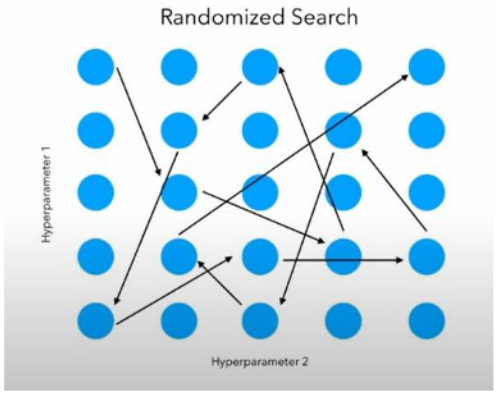
- **C:** This controls regularization strength. Smaller values (0.1) enforce stronger regularization, simplifying the model and reducing overfitting, while larger values (1) make the model more complex by reducing regularization.
- **penalty:** Specifies the norm for penalization. 'l1' promotes sparsity 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.

2.8.4. 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.

2.8.5. Randomized Search Cross Validation

Randomized 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.



```
clf = RandomizedSearchCV(
    estimator=ensemble_clf,
    param_distributions=param_dist,
    n_iter=10,
    scoring=custom_scorer,
    refit='f1_score',
    cv=TimeSeriesSplit(n_splits=10),
    random_state=42,
    n_jobs=-1,
    verbose=0,
    error_score='raise'
)
```

- **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] 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.

---

[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.

3. Ensemble Model Construction

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

3.1. Data collection

The initial phase employs a Jupyter Notebook[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)

3.2. Data 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

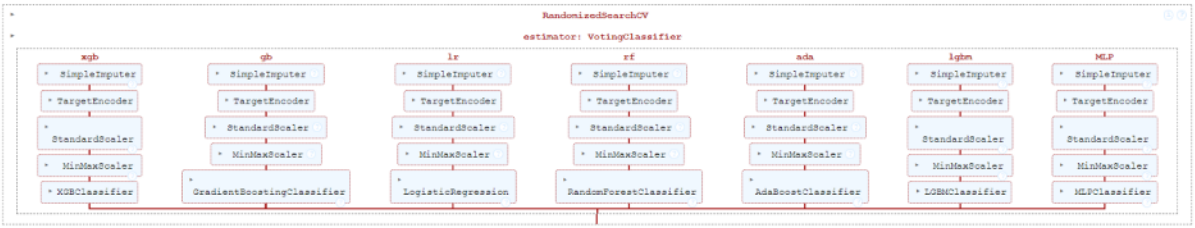
[13] Jupyter is an open-source tool that allows you to create and share documents containing live code, equations, visualizations, and explanatory text for data analysis and machine learning.

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.

3.3. 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.



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.

```
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
```

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.

3.4. Ensemble Techniques

3.4.1. Voting

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

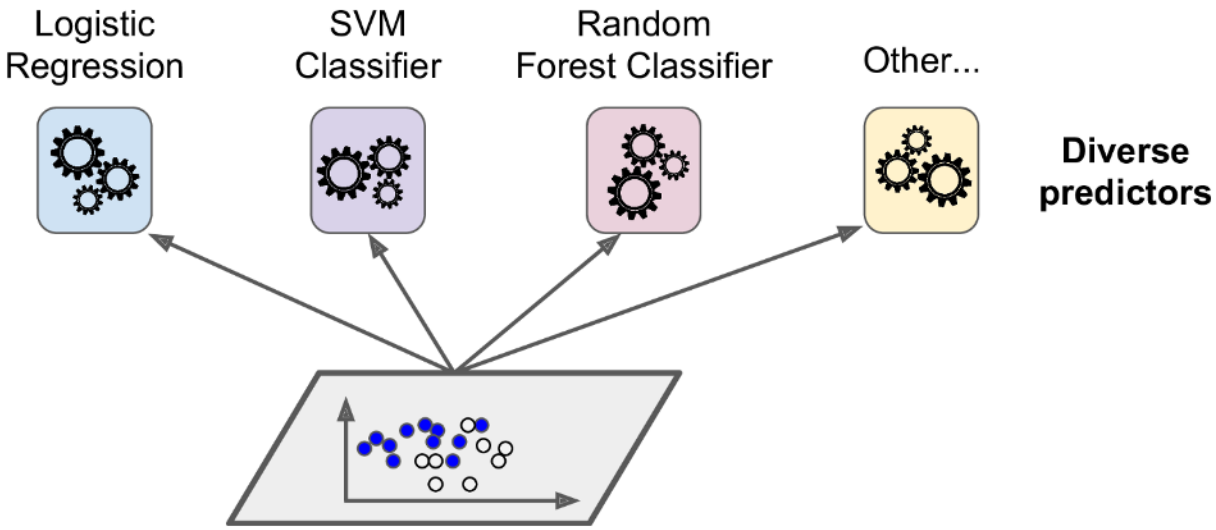
Where Voting is used:

The `VotingClassifier` is created to combine predictions from multiple classifiers:

```
ensemble_clf = VotingClassifier(
    estimators=[(name, pipeline) for name, pipeline in pipelines.items()],
    voting='soft'
)
```

Analysis:

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



3.4.2. Stacking

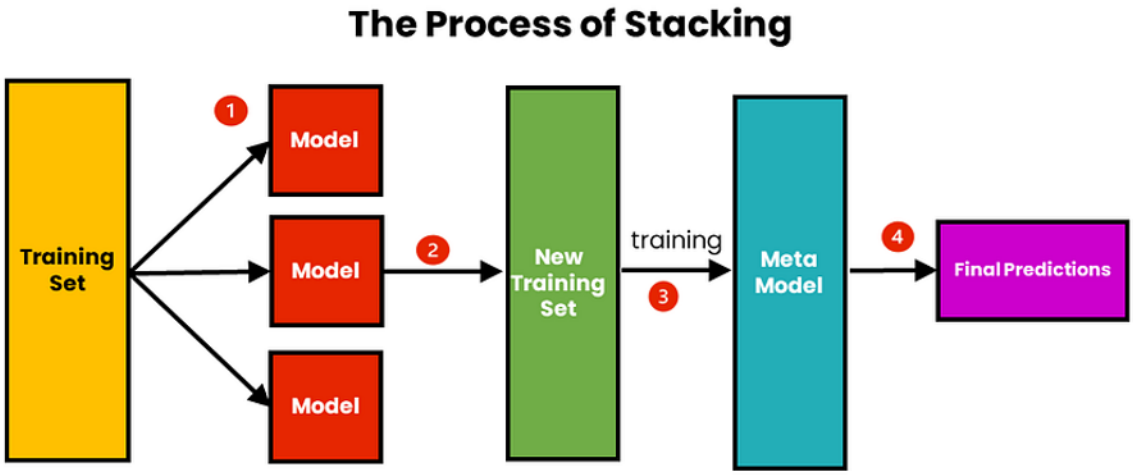
Stacking refers to combining multiple models via a meta-model that makes the final prediction based on the outputs of the base models.

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.



3.4.3. Bagging

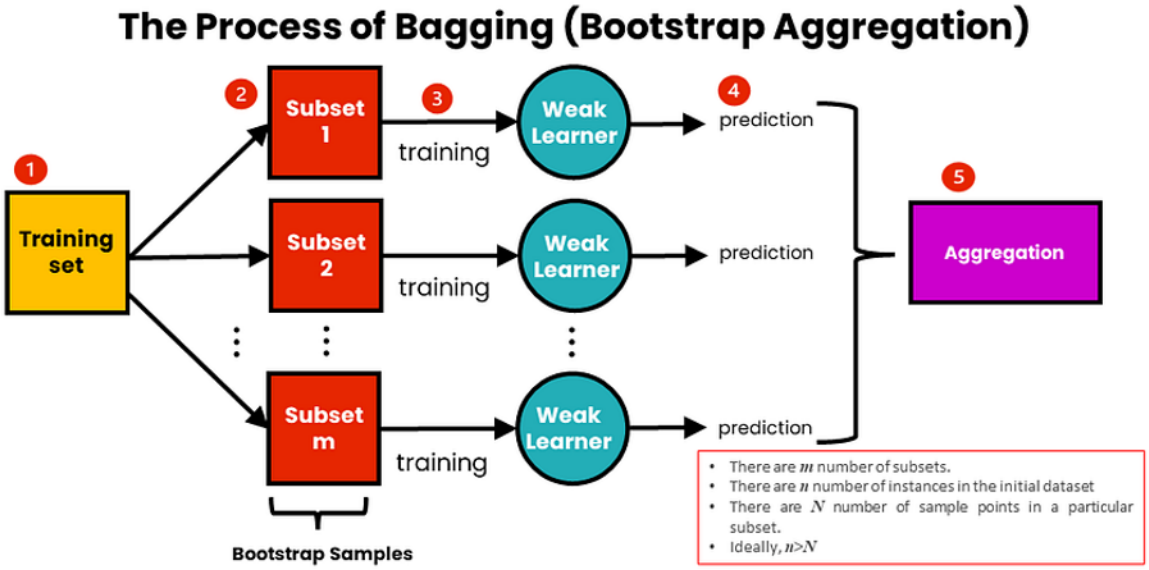
Bagging (Bootstrap Aggregating) involves training multiple models on different random subsets of the training data and then combining their predictions.

Where Bagging is used:

```
classifiers = {
    'rf': RandomForestClassifier(random_state=42, verbose=0),
}
```

Analyis:

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



3.4.4. Boosting

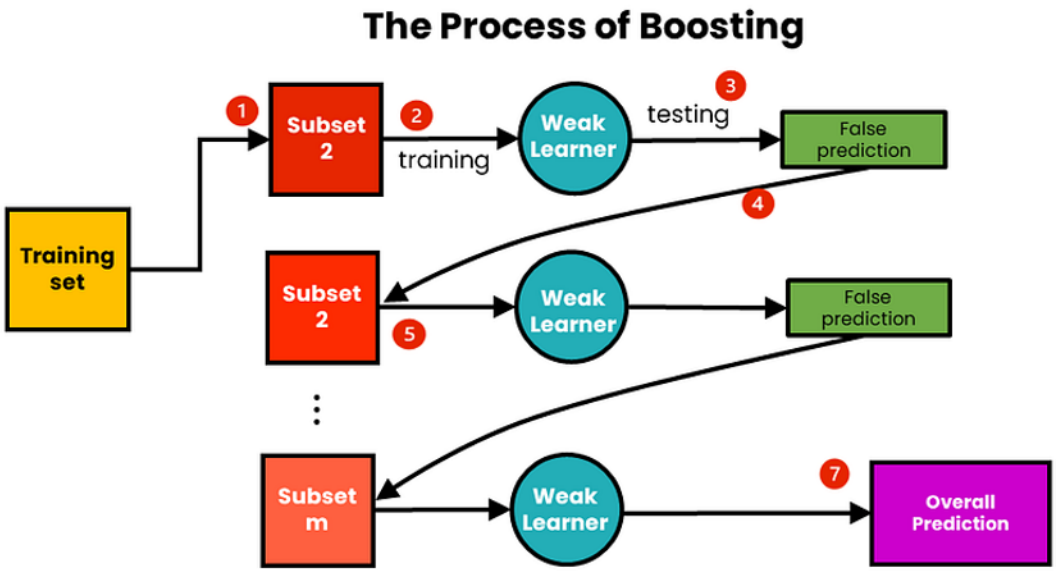
Boosting involves training multiple models sequentially, where each model tries to correct the errors of the previous ones.

Where Boosting is used:

```
classifiers = {
    'xgb': XGBClassifier(random_state=42, verbose=0),
    'gb': GradientBoostingClassifier(random_state=42, verbose=0),
    'ada': AdaBoostClassifier(random_state=42),
    'lgbm': LGBMClassifier(random_state=42, force_col_wise='true', verbose=0),
}
```

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.



4. Model Evaluation

4.1. Evaluation Metrics

4.1.1. Accuracy

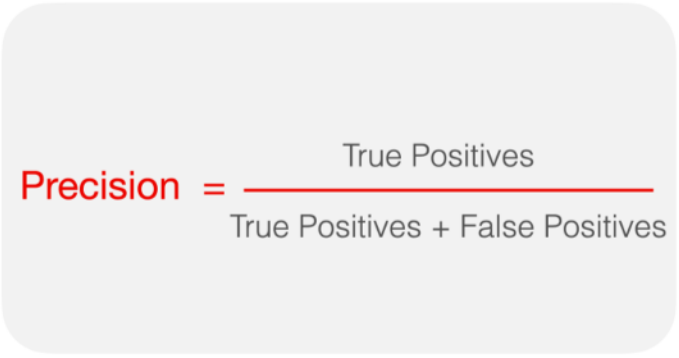
In the intricate domain of machine learning, accuracy stands as a pivotal performance metric, encapsulating the proportion of correctly predicted outcomes to the total number of predictions rendered. This metric, while seemingly straightforward, wields considerable power in the initial evaluation of 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.

$$\text{Accuracy} = \frac{\text{Correct predictions}}{\text{All predictions}}$$

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

4.1.2. 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]. 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.



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

4.1.3. Recall

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]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.

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

4.1.4. 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.

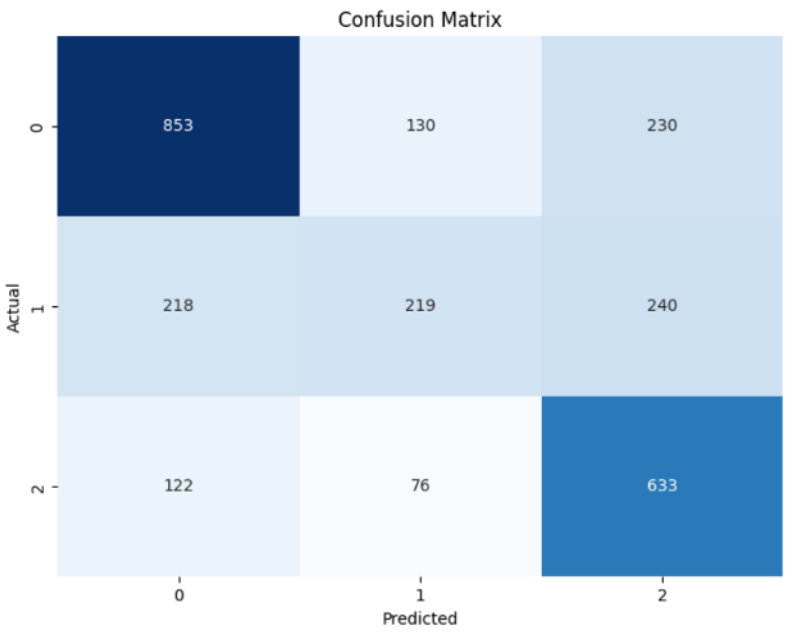
---

[14] In machine learning, "true positives" refer to instances where the model correctly identifies positive cases or events that are actually positive.

[15] False negatives occur when the model incorrectly predicts that a positive instance is negative, missing the correct identification

4.2. Confusion 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.

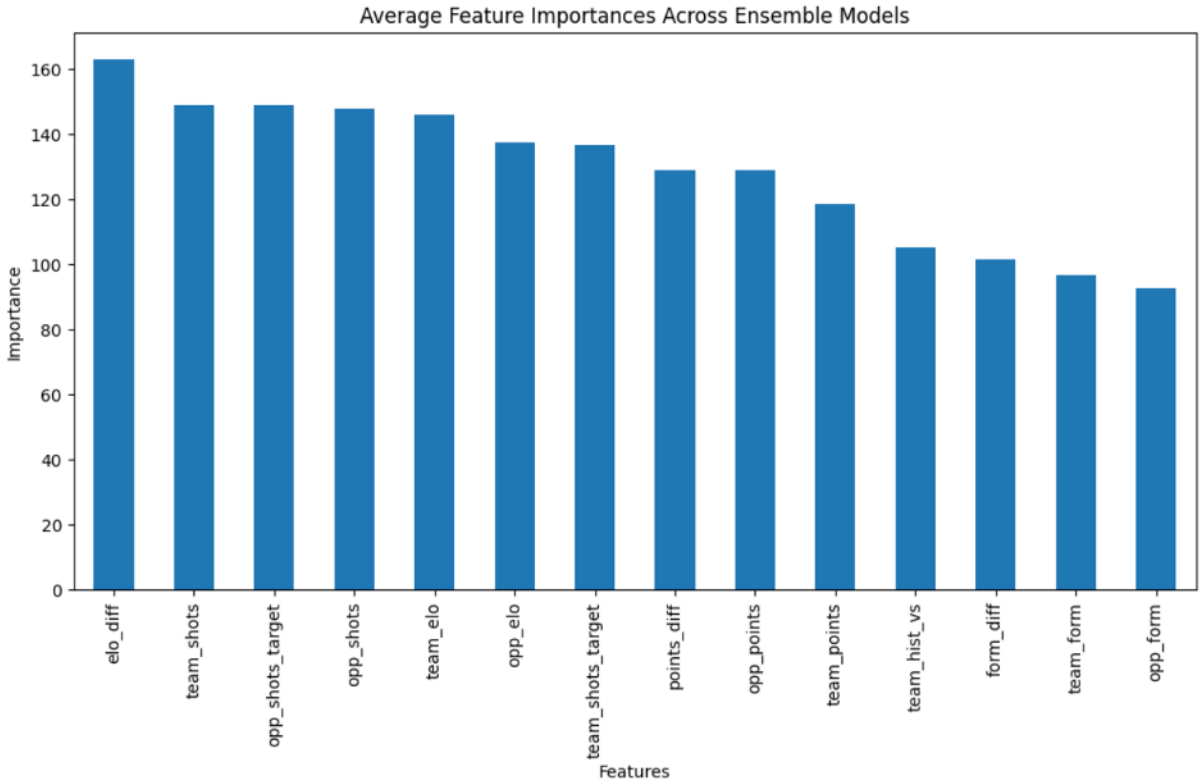
Interpretation of each part of the matrix:
0 = Home team won
1 = Draw
2 = Away team won

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).

4.3. 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.



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.

4.4. 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.

from sklearn.model_selection import TimeSeriesSplit

```
clf = RandomizedSearchCV(
    estimator=ensemble_clf,
    param_distributions=param_dist,
    n_iter=10,
    scoring=custom_scorer,
    refit='f1_score',
    cv=TimeSeriesSplit(n_splits=200),
    random_state=42,
    n_jobs=-1,
    verbose=0,
    error_score='raise'
)
```

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.
clf.fit(X_train, y_train)

During the `fit` [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.

4.5. Performance

4.5.1. 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.

[16] "fitting" refers to the process of training a model on data so that it can make accurate predictions.

Div	Date	Team	Opponent	FTR	AvgH	AvgD	AvgA	1X2	Correct
D1	20240518	Werder Bremen	Bochum	1	2,38	3,67	2,86	1X	TRUE
D1	20240518	Heidenheim	FC Koln	1	2,35	3,97	2,74	1X	TRUE
P1	20240518	Moreirense	Estoril	1	2,31	3,31	3,12	1X	TRUE
SP1	20240518	Alaves	Getafe	1	2,07	3,19	4,04	1X	TRUE
I1	20240519	Udinese	Empoli	X	2,04	3,32	3,9	1X	TRUE
F1	20240519	Toulouse	Brest	2	3,86	3,87	1,89	1X	FALSE

4Example matches identified as value bets

4.5.2. 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.

While 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.

Classification Report for the Test Set:

	precision	recall	f1-score	support
0	0.70	0.70	0.70	1213
1	0.51	0.32	0.39	677
2	0.57	0.74	0.64	831
accuracy			0.62	2721
macro avg	0.59	0.59	0.58	2721
weighted avg	0.61	0.62	0.61	2721

Classification Report for the Validation Set:

	precision	recall	f1-score	support
0	0.68	0.66	0.67	74
1	0.35	0.17	0.23	52
2	0.51	0.74	0.60	61
accuracy			0.55	187
macro avg	0.51	0.52	0.50	187
weighted avg	0.53	0.55	0.53	187

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.

5. 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.

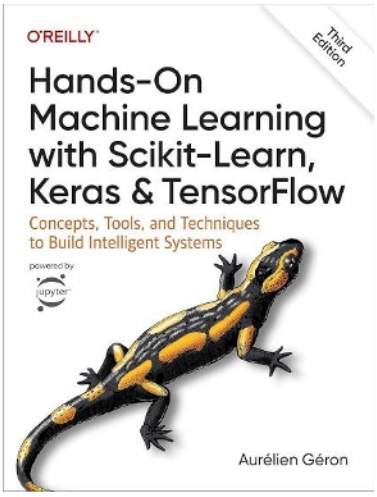
Next to sharpening my Python / Jupyter skills, this project taught me invaluable lessons in data collection, feature engineering, and model development. The practical application of advanced algorithms and strategic betting underscored the model's utility, not just in predicting results but in generating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

Jan Vermeerbergen
05/06/2024

6. Bibliography



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

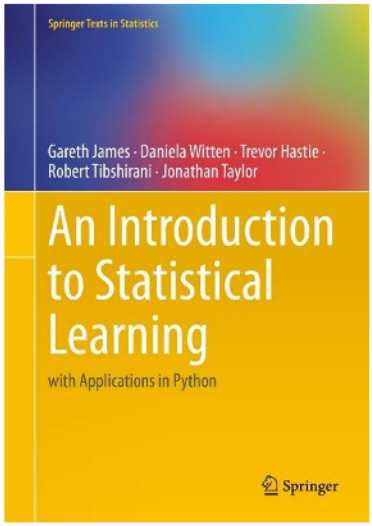
Predicting EPL Football Match Winners Using Machine Learning
https://app.dataquest.io/m/99992/portfolio-project%3A-predicting-epl-football-match-winners-using-machine-learning/1/project-overview

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

7. 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

7.1. 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}')
```

7.2. 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}")
```

7.3. 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],
}
```

```python
# %%
# 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)
```